Zürcher Hochschule für Angewandte Wissenschaften
School of Management and Law

Bachelorarbeit

# Prototypische Implementation einer oBPM-Ausführungsumgebung

## basierend auf der NoSQL-Datenbank ArangoDB

Autor:

Remo Zumsteg (S12472130), zumstrem@students.zhaw.ch
W.BA.WIN.13HS.VZb)

ZHAW School of Management and Law

Betreuung: David Grünert
Abgabetermin: Winterthur, 26. Mai 2016

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Management and Law

# Wahrheitserklärung

„Ich erkläre hiermit, dass ich die vorliegende Arbeit selbständig, ohne Mithilfe Dritter und nur unter Benützung der angegebenen Quellen verfasst habe und dass ich ohne schriftliche Zustimmung der Studiengangleitung keine Kopien dieser Arbeit an Dritte aushändigen werde."

Gleichzeitig werden sämtliche Rechte am Werk an die Zürcher Hochschule für angewandte Wissenschaften (ZHAW) abgetreten. Das Recht auf Nennung der Urheberschaft bleibt davon unberührt.

Name der / des Studierenden (Druckbuchstaben)

Unterschrift der / des Studierenden

……..

Zürcher Hochschule
für Angewandte Wissenschaften

School of
Management and Law

# Herausgabeerklärung der / des Dozierenden

Herausgabe[1)] der Bachelor-Arbeit „Prototypische Implementation einer oBPM-Ausführungsumgebung
basierend auf der NoSQL-Datenbank ArangoDB"

Die vorliegende Bachelor-Arbeit wird

☐ nicht herausgegeben.
☐ nicht herausgegeben werden bis ins Jahr .
☐ für eine uneingeschränkte Herausgabe freigegeben.

, .....

(Ort, Datum) (Unterschrift der / des Dozierenden)

[1)] Unter ”Herausgabe” wird sowohl die Einsichtnahme im Hause wie auch die Ausleihe bzw. die Abgabe zu Selbstkostenpreisen verstanden.

# Management Summary

Das Konzept von Prozessautomatisierungs-Systemen ist bereits seit Jahrzehnten ein fester Bestandteil von Geschäftsorganisationen. Mit den Jahren haben sich anhand verschiedener Einsatzgebiete unterschiedliche Konzepte entwickelt, wie und auf welche Weise Prozessmodelle definiert und in Automatisierungslösungen implementiert werden.

Neben den traditionellen control-flow-basierten Prozessmodellen haben sich dokument- und artefakt-zentrische Modellierungskonzepte durchgesetzt. Diese stellen die Dokumente und Artefakte eines Prozesses in den Mittelpunkt und fokussieren sich weniger auf den statischen Control-Flow traditioneller Prozessmodelle. Zu den bereits bestehenden dokument-zentrischen Prozessmodellen hat sich das Konzept des Opportunistic Business Process Modeling (oBPM) dazu gesellt.

Im Rahmen dieser Arbeit wird ein Software-Prototyp basierend auf dem Datenbanksystem ArangoDB implementiert, auf dessen Basis oBPM-basierte Prozessmodelle definiert und ausgeführt werden können. Mit Hilfe des umgesetzten Prototypen wird geprüft, inwiefern sich ArangoDB für die Umsetzung eines oBPM-Systems eignet hinsichtlich der Performance, Skalierbarkeit und weiteren nicht-funktionalen Anforderungen.

Dazu werden in dieser Arbeit in einem ersten Schritt die Anforderungen an ein oBPM-Modellierungs- und Ausführungssystem analysiert und zusammengefasst. In einem nächsten Schritt wird der Funktionsumfang und die Einsatzmöglichkeiten von ArangoDB geprüft, um auf dieser Basis die zu implementierende Datenstruktur zu planen. Danach werden verschiedene Varianten von möglichen Systemarchitekturen evaluiert und miteinander verglichen. Nach Abschluss der Analyse wird die Umsetzung der Implementation aufgezeigt, hinsichtlich der Datenstrukturen und Applikationsschnittstellen. Als letzter Teil dieser Arbeit wird aufgezeigt, wie die umgesetzte Implementation bezüglich der funktionalen Anforderungen, der Performance und der Skalierbarkeit getestet wird.

Anhand des in dieser Arbeit implementierten Prototypen kann aufgezeigt werden, dass sich die verwendeten Software-Komponenten, im Speziellen ArangoDB, sehr gut für die Umsetzung eines oBPM-Systems eignen. Alle funktionalen Anforderungen können im Prototypen umgesetzt werden. Vor allem das Multi-Model-Konzept von ArangoDB,

welches dokumenten- und graphen-basierte Datenbankkonzepte vereint, eignet sich gut um die in der Modellierung nach oBPM anfallenden Datenstrukturen zu persistieren.

Mit Hilfe von Performancetests anhand verschiedener Benutzungsszenarien kann aufgezeigt werden, dass die vom implementierten Prototyp erreichte Performance und Skalierbarkeit nicht für den produktiven Betrieb genügend ist. Die Reaktionszeit des Systems unter hoher Last übersteigt die in den Testszenarien definierten Richtwerten von unter 2 Sekunden beträchtlich.

Nichtsdestotrotz kann diese Arbeit aufzeigen, dass die Implementation eines oBPM-basierten Systems zur Modellierung und Ausführung von Prozessen in funktionaler Hinsicht möglich ist und dass sich das Datenbanksystem ArangoDB als zentrale Einheit einer oBPM-Umgebung bewährt.

# Inhaltsverzeichnis

## Tabellenverzeichnis

# Abbildungsverzeichnis



# Code-Verzeichnis

# Auflistungsverzeichnis

# 1 Einleitung

Der Begriff Business-Prozess-Automatisierung tauchte bereits vor 1970 auf, oft unter dem Namen Office-Automation (Stohr & Zhao, 2001, S. 2). Traditionelle arbeiten dabei anhand eines Top-Down-Ansatzes, wie von Grünert, Brucker-Kley, & Keller (o. J.-b) ausgeführt. Wie von Grünert, Brucker-Kley, & Keller (o. J.-b) weiter beschrieben, werden traditionellerweise Prozessautomatisierungen in einem ersten Schritt von der Fachabteilung, bzw. einem Businessanalysten modelliert. In einem zweiten Schritt werden die erstellten Prozessmodelle von einem Techniker in eine Ausführungsumgebung implementiert. Dieses Vorgehen birgt den Nachteil, dass ohne technisches Knowhow keine ausführbare Implementation des Prozesses möglich ist, da dank der Komplexität der meisten Prozesse keine automatisierte Umwandlung des beschreibenden Modells in eine ausführbare Komponente möglich ist (Grünert u. a., o. J.-b).

Dieselbe Problematik von traditionellen Umgebungen zur Prozessautomatisierung konnte von Bandara, Indulska, Chong, & Sadiq (2007, S. 1249) festgestellt werden. Sie konnten mit Hilfe einer Expertenumfrage ableiten, dass es zwischen der Phase der Prozessmodellierung und der Prozessausführung zu erheblichen Übersetzungs-Problemen kommt, da oft für die verschiedenen Phasen verschiedene Software-Umgebungen nötig sind. Von Bandara u. a. (2007, S. 1249) befragte Experten führen weiter aus, dass durch die manuelle Übersetzung von Prozessmodellen zu ausführbaren Prozessumgebungen zusätzliche Arbeit anfällt und die Möglichkeit besteht, dass Information während der Übersetzung verloren geht.

Einen weiteren Nachteil von traditionellen workflow-basierten Automatisierungs-Systemen besteht darin, dass sie nur eine transaktionale Ausführung von Prozessschritten erlaubt (Grünert, Brucker-Kley, & Keller, o. J.-a). Dies zeigt sich als ungenügend, sobald Knowledge-Workers am Prozess beteiligt sind, welche anhand von verfügbarer Information in Form von Dokumenten oder Business-Objects über die Entscheidung verfügen müssen, welche nächsten Prozessschritte getätigt werden müssen (Grünert u. a., o. J.-a).

Um der Problematik von traditionellen Prozessautomatisierungssystemen zu begegnen, setzen Grünert u. a. (o. J.) auf ein neues Konzept zur Modellierung und Ausführung von

automatisierten Geschäftsprozessen. Das sogenannte Opportunistic Business Process Modeling (oBPM) reiht sich in die Kategorie der dokument-zentrierten, bzw. artifact-centric Workflow-Systeme ein. Das Konzept von oBPM schlägt dabei ein Modell vor, welches durch die reduzierte Komplexität von Business-Usern verstanden werden kann und ihnen somit erlaubt, am Design eines Prozesses aktiv mitzuwirken (Grünert u. a., o. J.-b). Dazu ermöglicht oBPM die Ansicht aus zwei Perspektiven. Zum einen die bekannte Top-Down-Perspektive für Prozess-Owner und Prozess-Administratoren zur Modellierung eines Prozesses. Zum anderen eine Bottom-Up-Perspektive, welche es Business-Usern erlaubt das Modell aus Benutzer-Sicht anzupassen. Bei beiden Perspektiven handelt es sich um dasselbe Prozess-Modell, wodurch keine Übersetzung zwischen dem Prozess-Design und der Prozess-Ausführung notwendig ist (Grünert u. a., o. J.-b). Zusätzlich führen die Autoren aus, dass oBPM den Business-Usern erlaubt, selbstständig, bzw. opportunistisch zu entscheiden welche nächsten Modell-Änderungen mit Hilfe von Prozess-Schritten ausgeführt werden sollen.

In dieser Arbeit wird ein Software-Prototyp implementiert, welcher nach dem Konzept von oBPM von Grünert u. a. (o. J.-b) und Grünert u. a. (o. J.-a) das Modellieren und Ausführen von Prozess-Modellen erlaubt. Dabei wird aufgezeigt, wie sich das theoretische Modell von oBPM in der Praxis bewährt und ob die von oBPM beschriebenen Vorteile gegenüber traditionellen Automatisierungssystemen in einem Software-Prototypen umgesetzt werden können. Anhand der Aufgabenstellung der vorliegenden Bachelor-Thesis von Grünert (2015) wird dabei auf das NoSQL-Datenbanksystem ArangoDB gesetzt. Die Verwendung einer NoSQL-basierten Datenbank im Kontext von oBPM ist sinnvoll, da sich die dynamischen Prozessmodelle und Datenstrukturen einfacher abbilden lassen als in einem schema-basierten, bzw. relationalen Datenbank-System.

Das Forschungsziel dieser Arbeit besteht darin herauszufinden, inwiefern sich das System ArangoDB zur Umsetzung einer oBPM-Umgebung eignet, hinsichtlich Performance, Skalierbarkeit und weiterer nicht-funktionaler Anforderungen. Dazu werden nach der Umsetzung des oBPM-Prototypen verschiedene Arten von Test-Szenarios abgewickelt, um den Prototypen auf seine Funktionalität, Skalierbarkeit und Performance zu testen.

## 1.1 Stand der Forschung

Dieses Kapitel gibt Einblick in den aktuellen Stand der Forschung im Kontext von dokumentzentrischen, bzw. artefaktzentrischen Business-Prozess-Automatisierungs-Systemen. Zusätzlich werden die vorgestellten Forschungsprojekte mit der Thematik dieser Thesis verglichen, um die vorliegende Arbeit in die aktuellen Fortschritte in der Forschung im Bereich dokument-zentrischen und artifact-zentrischen Prozesssystemen einzugliedern.

Wang & Kumar (2005) stellten ein Framework für document-zentrische Workflow-Systeme vor, welche ohne expliziten Control-Flow auskommen, und den Ablauf eines Prozesses anhand von Dokumenten und deren Abhängigkeiten zueinander steuern. Dazu wurde zuerst ein konzeptionelles Framework vorgestellt und auf dessen Basis ein Softwareprototyp mit Hilfe des relationalen Datenbank-Systems Microsoft SQL Server 2000 implementiert (Wang & Kumar, 2005). Das von ihnen entwickelte Framework basiert auf Prozessmodellen, welche aus Tasks, Dokumentklassen und RessourcenklassenRessourcen-Klassen besteht. Wird durch einen externen Event eine neue Prozess-Instanz instanziiert, erstellt das Framework einen neuen Prozess-Case, inklusive initialen Dokument-Instanzen. Ein Task wird zu einem Work-Item instanziiert, sobald alle benötigten Dokumente für die Ausführung des Tasks vorhanden sind. Sobald die benötigten Dokumente des Work-Item vorhanden , transformiert es sich zu einer Activity und kann über einen externen oder internen Event ausgeführt werden, wodurch neue Dokumente erstellt werden oder die zur Verfügung gestellten Dokumente manipuliert werden (Wang & Kumar, 2005). Die Veränderung jeglicher Dokumente löst Events aus, welche wiederum von anderen Dokumenten abgefangen werden können und je nach Constraints des Prozess-Modells zu weiteren Veränderungen von Dokumenten führt (Wang & Kumar, 2005). Des Weiteren haben die Autoren den implementierten Prototypen vollständig in TSQL verfasst und betreiben ihn in der Datenbankumgebung des Microsoft SQL 2000 Servers.

Wang & Kumar (2005) sehen den Vorteil ihres Frameworks im Vergleich zu traditionellen control-flow-basierten Systemen darin, dass ihr document-zentrischer Ansatz besser geeignet ist für ad-hoc-Events, da durch einfaches Anpassen von Constraints betreffend Dokument-Abhängigkeiten der Prozess-Flow einfach angepasst

werden kann. Zweitens kann durch die strikte Abhängigkeit von Dokumenten und die klare Definition von Constraints der korrekte Ablauf eines Prozesses einfach verifiziert werden. Jedoch wirft der Ansatz laut Wang & Kumar (2005) auch Nachteile auf. Zum einen ist es im Gegensatz zu control-flow-basierten Prozessen schwierig, die zu modellierenden Abhängigkeiten und Constraints zu visualisieren. Zweitens können falsch gesetzte Abhängigkeiten und Constraints zu Konflikten führen, welche relativ schwer auffindbar und lösbar sind.

Ngamakeur, Yongchareon, & Liu (2012) stellten ein weiteres Framework zur Modellierung und Ausführung von artifact-zentrischen Prozess-Modellen in einer service-orientierten Umgebung vor. Ihr Framework basiert auf drei separaten Konzepten. Erstens erstellten sie eine neue Prozessmodelldefinition, welche die Tasks und Artifact-Typen beinhaltet. Als zweites stellte Ngamakeur u. a. (2012) ein Konzept für eine Ausführungsumgebung auf, welche fähig sein muss, die zuvor definierten Prozessmodelle auszulesen und laufende Prozesse anhand der von Business-Regeln definierten Constraints auszuführen. Das letzte Konzept stellen die Business-Regeln dar. Diese definieren die Constraints, unter welchen Tasks für bestimmte Artifact-Typen ausgeführt werden können.

Ngamakeur u. a. (2012) entwickelten dazu ein eigenes XML-basiertes Datenformat für die Definition eines artifact-centric Process-Model (ACP-Model). Dieses Modell beinhaltet die Artifact-Definitionen, die im Modell definierten Tasks und die Business-Rules. Zusätzlich beinhaltet das Modell mehrere Mapping-Definitionen, anhand derer Daten von externen Client zu Artifacts transformiert werden und die Antwort des Frameworks wieder zurück in das zur Kommunikation verwendete Protokoll transformiert werden kann. Dies erlaubt dem von Ngamakeur u. a. (2012) entwickelten System die automatische Definitionen von API-Schnittstellen zur Ausführung von Tasks über externe Clients. Eine weitere Besonderheit des von Ngamakeur u. a. (2012) entwickelten Frameworks stellt die strikte Trennung zwischen Artefakten, Prozessmodellen und den Business-Rules dar. Dies erlaubt eine einfache Modifikation des Prozessablaufs durch Änderungen der Rules ohne dass das Prozessmodell selbst angepasst werden muss.

Redding, Dumas, ter Hofstede, & Iordachescu (2010) stellten ein weiteres Framework zur Prozessmodellierung und -ausführung vor, basierend auf einem object-zentrischen Ansatz. Um der Inflexibilität von control-flow-basierten Prozess-Automatisierungs-

Systemen zu begegnen, schlagen die Autoren ein System vor, welches eine möglichst hohe Flexibilität während der Laufzeit eines Prozesses ermöglicht. Dazu definierten Redding u. a. (2010) drei Patterns, welche die Flexibilität eines objektzentrischen Systems vorgeben. Das erste Pattern beschreibt die Flexibilität, welche es erlaubt, während der Ausführung eines Prozesses, abhängig von im Modell definierten Constraints, Tasks auszuführen, wobei die Reihenfolge der zu ausführenden Tasks nicht statisch im Modell definiert ist (Redding u. a., 2010). Das zweite Pattern beschreibt das Delegieren einer Ausführung eines Tasks an einen anderen Task, wobei alle Daten und der Ausführungskontext übergeben werden. Somit kann in einem spezifischen Fall die Ausführung von einem generischen Task an einen spezifischen Task weitergereicht werden (Redding u. a., 2010). Das dritte Pattern beschreibt die Flexibilität zur Ausführung von Sub-Prozessen während der Ausführung eines Parent-Prozesses. Dies erlaubt die Verschachtelung von Prozessinstanzen und erhöht die Modularisierung von Prozessdefinitionen (Redding u. a., 2010). Zwar hat das von den Autoren vorgeschlagene Framework vieles gemein mit traditionellen control-flow-basierten Prozesssystemen, doch erlangt es mit Hilfe der vorgestellten Patterns eine starke Flexibilität für die Bildung von ad-hoc Prozessen.

Zwar haben alle in diesem Kapitel untersuchten Frameworks zur artifact- und objektzentrischen Prozessmodellierung und -ausführung ähnliche Forschungsziele formuliert, was die Erhöhung der Flexibilität und Vereinfachung der Erstellung von ad-hoc Prozessen anbelangt. Jedoch unterscheiden sich die Herangehensweisen wie in diesem Kapitel beschrieben stark. Die vorliegende Arbeit verfolgt im grossen Kontext ähnliche Ziele wie die vorgestellten Forschungsarbeiten, folgt aber mit dem Konzept vom Opportunistic Business Process Modeling einen anderen Ansatz.

## 1.2 Aufbau dieser Arbeit

Dieses Kapitel erläutert die Struktur und den Aufbau dieser Thesis. In einem ersten Schritt werden dabei die Anforderungen an den Prototypen des oBPM-Systems aufgezeigt, hinsichtlich funktionaler und nicht-funktionalen Erwartungen. Als nächstes wird das Vorgehen während der Analyse und Implementation des Systems beschrieben. Der Hauptteil der Arbeit startet mit einer Analyse der Systemkomponenten zur Entscheidungsgrundlage der Systemarchitektur. Nach der Begründung der Wahl einer Architektur werden die verwendeten Komponenten und Technologien vorgestellt. Als

nächster Punkt wird die Datenstruktur thematisiert, inklusive einer vorhergehenden Analyse verschiedener Implementationsvarianten und der Beschreibung der tatsächlich implementierten Version der Datenarchitektur. In einem weiteren Teil wird im spezifischen auf die Benutzerauthentifizierung und Autorisierung eingegangen und die gewählte Implementation erläutert. Danach wird das durchgeführte Testing des Systems beschrieben mit Einblick in die Testimplementation und Testresultate. Die letzten Kapitel behandeln die offenen Punkte der Arbeit, die Konklusion dieser Thesis und eine Handlungsempfehlung für zukünftige Forschungsarbeiten im Gebiet von oBPM-Systemen.

Der Appendix dieser Arbeit beinhaltet zusätzliche Informationen zur Installation und Inbetriebsnahme des entwickelten oBPM-Systems und weiteren Informationen zu den gewählten Test-Szenarios.

# 2 Anforderungen an das oBPM-System

Folgendes Kapitel fasst den von Grünert u. a. (o. J.-b) und Grünert, Brucker-Kley, & Keller (o. J.-a) erwarteten Funktionsumfang an ein oBPM-System zusammen und beschreibt die funktionalen Erwartungen von Grünert (2015) an den Prototypen. Als Grundlage dienen dabei die beiden Papiere von Grünert u. a. (o. J.-a, o. J.-b) und die Aufgabenstellung zu dieser Bachelor-Arbeit von Grünert (2015).

## 2.1 Grundsätzliche Erwartung und Funktionsumfang

Dieses Kapitel gibt eine Übersicht an den erwarteten Funktionsumfang eines oBPM-System basierend auf den Definitionen von Grünert u. a., o. (J.-a, o. J.-b). Die erwarteten Funktionalitäten werden aufgeteilt nach den von oBPM definierten Top-Down- und Bottom-Up-Perspektiven und dienen als Grundlage zur Definition der Grunderwartung an den oBPM-Prototypen.

### 2.1.1 Top-Down-Perspektive

Die Top-Down-Perspektive in einem oBPM-System erlaubt es Prozess-Owner die Definition eines beliebigen Prozess-Modells. Dazu ist ein User-Case-Diagramm, ein Klassen-Diagramm und für jeden Typ von Dokument-Artefakt (Dokument) ein separates State-Maschine-Diagramm notwendig.

Das Use-Case-Diagramm definiert dabei die Abhängigkeiten zwischen Benutzer-Rollen, Prozess-Tasks, sogenannten Prozess-Actions, und den Dokumenttypen. Das Diagramm definiert dabei welche Benutzer-Rollen welche Prozess-Actions ausführen können und welche Typen von Dokumenten für die Ausführung der Prozess-Action benötigt werden. Des Weiteren definiert oBPM System-Actions, welche Prozess-Actions darstellen, die automatisiert vom System statt von einem Benutzer ausgeführt werden. Die Abbildung 1 zeigt ein simples Beispiel eines User-Case-Diagramms. Dabei werden zwei Benutzerrollen *Professor* und *Student* definiert. Des Weiteren kommen die drei Prozess-Actions *Create Thesis, Assign Student* und *Upload Document* vor. Schlussendlich werden die vier Dokumenttypen *Case, Thesis, Student* und *Upload* definiert. Die Verbindungen zwischen Benutzerrollen und Actions, bzw. zwischen Actions und Dokumenttypen geben Auskunft welche Rolle welche Prozessaktion ausführen darf und welche Dokumente dazu notwendig sind.

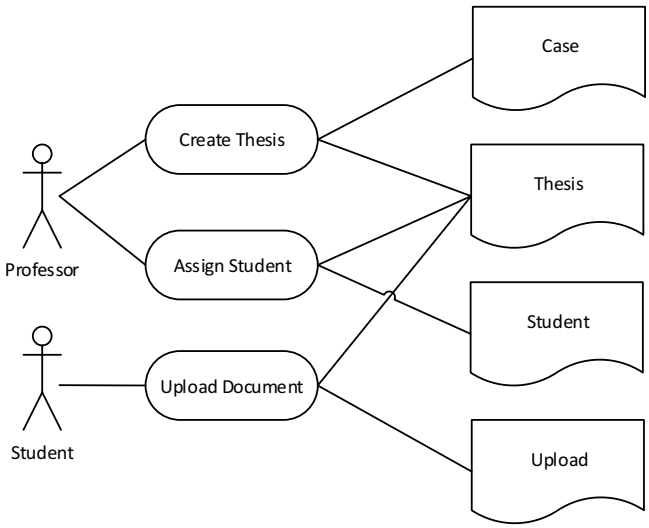


Abbildung 1: Beispiel eines Use-Case-Diagramms

Das Klassendiagramm eines oBPM-Modells definiert die Datenstruktur der Prozessdaten, welche in Form von Dokumentartefakten während der Ausführung von Prozessen anfallen. Das Diagramm definiert dabei die Typen von Dokumenten und ihre Relation zueinander. Abbildung 2 zeigt ein Beispiel eines Klassendiagramms inklusive der Kardinalität der Entitäten-Relationen.

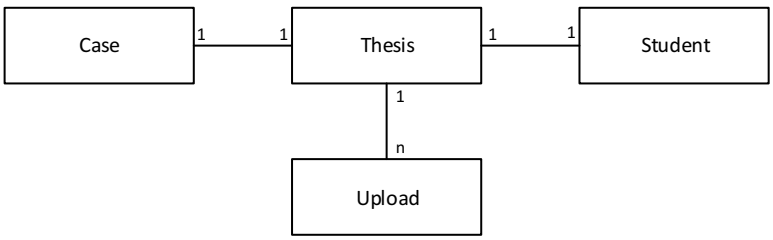


Abbildung 2: Beispiel eines Klassendiagramms

Das letzte benötigte Diagramm, das State-Machine-Diagramm, definiert für jeden Dokumenttyp im Use-Case- und Klassendiagramm alle möglichen Statusübergänge und welche Benutzerrollen einen Statusübergang vornehmen können. Jeder Statusübergang wird dabei von einer im Use-Case definierten Prozess-Action ausgelöst.

Die für den oBPM-Prototypen implementierte Datenstruktur muss fähig sein, alle Informationen, welche mit Hilfe der drei vorgestellten Diagramme produziert werden, zu persistieren. Zusätzlich muss die Datenstruktur erlauben, die Diagramme mit Hilfe der gespeicherten Daten zu rekonstruieren.

### 2.1.2 Bottom-Up-Perspektive

Die Diagramme, welche in Kapitel *2.1.1 Top-Down-* vorgestellt werden, sind zwar sehr hilfreich für die Modellierer eines Prozesses, jedoch nicht für ein Knowledge-Worker, da die Diagramme zum einen ein technisches Knowhow voraussetzen und zum anderen zu detailliert sind für die meisten Benutzerrollen. Für diese Benutzerrollen bietet das oBPM-Modell eine weitere Ansicht an, welche nur die für eine Rolle relevanten Informationen anzeigt. Sie beinhaltet alle vom aktuellen Benutzer ausführbaren Prozess-Actions und die Information, welche Dokumente für die jeweilige Ausführung benötigt werden. Dazu muss das System der Benutzerin oder dem Benutzer zusätzlich erlauben, von einer Bottom-Up-Perspektive in das Prozessmodell eingreifen zu können, indem er/sie die in Auflistung 1 definierten Funktionen im oBPM-System ausführen kann.

- Hinzufügen, editieren und löschen von Prozess-Actions für die aktuelle Benutzerrolle.
- Dokumente von einer Prozess-Action hinzufügen oder entfernen.
- Statusübergänge von Dokumenten hinzufügen, bearbeiten oder löschen.

Auflistung 1: Bottom-Up-Funktionalitäten für Knowledge-Workers

Der zu entwickelnde oBPM-Prototyp muss in der Lage sein, Schnittstellen für Knowledge-Worker anzubieten, welche ihr oder ihm erlauben, mit den zuvor definierten Funktionalitäten arbeiten zu können.

## 2.2 Funktionale Anforderungen an das System

Zusätzlich zu den in Kapitel 2.1 definierten Grunderwartungen erläutert dieser Abschnitt in Tabelle 1 die funktionalen Anforderungen an den oBPM-Prototypen, welche aus der Aufgabenstellung von Grünert (2015) abgeleitet wurden. Die Identifikationsnummer einer Anforderung dient als Referenz an verschiedenen Positionen in diesem Papier.

Tabelle 1: Systemanforderungen an den oBPM-Prototypen

| ID | Beschreibung der Anforderung |
|---|---|
| **001** | **Prozess-Datenmodell:** |
| 001.1 | Das System muss ein Datenmodell bereitstellen, welches erlaubt, ein Prozessmodell nach den Anforderungen von Grünert u. a. (o. J.-b) und Grünert, Brucker-Kley, & Keller (o. J.-a) zu hinterlegen. |
| **002** | **REST-Schnittstelle für Prozessmodell:** |
| 002.1 | Das System muss eine REST-basierte Schnittstelle zur Bearbeitung eines Prozessmodells zur Verfügung stellen. |
| 002.2 | Die REST-Schnittstelle muss eine Benutzer-Athentifizierung durchführen und eine Autorisierung von Benutzerrollen erlauben. |
| 002.3 | Die Prozessdatenschnittstelle muss eine genügende Testabdeckung mithilfe von Modultests besitzen. |
| **003** | **Prozessdatenverwaltung:** |
| 003.1 | Das System muss in der Lage sein, alle Prozessdokumente und deren Zustände in ArangoDB abzulegen. |
| 003.2 | Die Prozessdatenverwaltung muss Dokumente mit strukturiertem Inhalt (JSON) wie auch binärem Inhalt (Office-Dokumente, PDFs, etc.) verwalten können. |
| 003.3 | Die Prozessdatenverwaltung muss jegliche Änderungen in den Zuständen einzelner Dokumente in einer Änderungsgeschichte erfassen. |
| **004** | **REST-Schnittstelle für Prozessdaten:** |
| 004.1 | Das System muss eine REST-Schnittstelle offerieren um Prozessdokumente und deren Zustände auslesen und bearbeiten zu können. |
| 004.2 | Die REST-Schnittstelle für Prozessdaten muss eine Benutzerauthentifizierung durchführen und eine Autorisierung spezifischer Benutzerrollen ermöglichen. |
| 004.3 | Die Prozessdatenschnittstelle muss eine genügende Testabdeckung mithilfe von Modultests besitzen. |
| **005** | **Benchmarking und End-to-End-Tests:** |
| 005.1 | Die Performanz und andere nichtfunktionale Anforderungen müssen mit Hilfe von Benchmarking-Tests an typischen Anwendungsszenarien gemessen werden. |

Zusätzlich zu den grundsätzlichen funktionalen Anforderungen an den oBPM-Prototypen listet Tabelle 2 weitere Funktionalitäten auf, die nach Möglichkeit implementiert werden sollen, um den Funktionsumfang des Prototypen zu erweitern (Grünert, 2015).

Tabelle 2: Erweiterte Systemanforderungen an den oBPM-Prototypen

| ID | Beschreibung der Anforderung |
|---|---|
| **006** | **Implementation eines User-Interface oder Anbindung an eine bestehende Workbench (z.B. Camuda):** |
| 006.1 | Implementation einer graphischen Benutzeroberfläche für die Ausführung von Prozessen. |
| 006.2 | Implementation einer graphischen Benutzeroberfläche zur Modellierung und Visualisierung von Prozessmodellen. |
| **007** | **Versionierung von Prozess-Elementen:** |
| 007.1 | Versionierung von bestehenden Prozessmodellen und ihren Komponenten bei Änderungen des Modells. |
| 007.2 | Versionierung von Prozessdaten bei deren Manipulation durch die Ausführung von Prozess-Tasks. |

# 3 Methodik

Dieses Kapitel beschreibt das Vorgehen bei der Entwicklung des oBPM-Prototypen. Die Vorgehensweise kann dabei grob in eine Analyse-, Planungs- Implementations- und Testphase gegliedert werden. In der Analysephase wurde als erstes eine Recherche unternommen zum Thema dokumentzentrisches Prozessmanagement und -automatisierung, um einen Überblick über den aktuellen Stand der Forschung zu erhalten. Weiter wurden die Anforderungen an ein oBPM-System analysiert um den Scope der Arbeit festlegen zu können. Als letztes wurde das Datenbanksystem ArangoDB und dessen Funktionsumfang analysiert, um auf der Basis der Ergebnisse die zu verwendende Datenstruktur zu planen.

In der Planungsphase wurde anhand der zuvor erhaltenen Analyseresultaten und Scopedefinitionen die Architektur des gesamten oBPM-Systems geplant und die benötigten Systemkomponenten elaboriert. Der nächste Schritt in der Planungsphase

bestand in der Definition der Datenstruktur für die Verwaltung der Prozessmodelle und –Daten. Gleichzeitig wurden die nötigen API-Schnittstellen festgelegt, welche im oBPM-System implementiert werden mussten. Als Hilfe zur Entscheidungsfindung wurde bereits während der Planungsphase mit dem Erstellen eines simplen Prototyps begonnen. Dies unterstützte den Entscheidungsprozess bei der Auswahl der zu verwendenden Systemarchitektur und -technologie.

Während der Implementationsphase wurde als erstes die Basis des Systems umgesetzt, indem die einzelnen elaborierten Systemkomponenten aufgesetzt und entwickelt wurden. Danach wurden die geplanten Datenstrukturen und API-Schnittstellen parallel implementiert.

Die Phase des manuellen Testings fand zum einen parallel während der Implementierung statt. Fertig gestellte API-Schnittstellen und Business-Logik-Komponenten wurden auf ihre Funktionalität überprüft und wenn nötig angepasst. Diese Vorgehensweise erlaubte die frühe Erkennung von fehlerhaften Implementationen und vereinfachte dadurch allfällige Anpassungen am System. Zum anderen wurde nach der Implementationsphase eine separate Phase des Testings durchgeführt, welche sich auf die Implementation von Testszenarios zum automatisierten Testing der API-Schnittstellen und auf die Implementation von Performancetests konzentrierte.

Die gewählte Vorgehensweise ist dem bekannten Wasserfallmodell sehr ähnlich, jedoch besitzt das Wasserfallmodell wie bereits von Arndt, Hermanns, Kuchen, & Poldner (2009, S. 7) beschrieben unter anderem folgende Schwachpunkte, welche in Auflistung 2 zusammengefasst sind.

- Die Phase des Testings findet erst am Ende des Projekts statt. Etwaige auftretende Fehler sind in dieser Phase des Projekts sehr schwierig zu beheben, da möglicherweise ein grosser Teil der Implementation erneut durchgeführt werden muss.
- Durch den sequentiellen Charakter des Modells können Änderungen in den Anforderungen während späteren Phasen des Projekts nur schlecht berücksichtigt werden.

Auflistung 2: Nachteile des Wasserfall-Modells

Den Schwachpunkten des Modells wurde entgegengekommen, indem schon während der Planungsphase mit der Implementation eines ersten Prototyps begonnen wurde, um allfällige Fehler in der Umsetzung von Anforderungen entgegenzuwirken. Des Weiteren

wurde die Phase des Testings bereits während der Implementationsphase gestartet, um allfällige Fehler frühzeitig erkennen zu können.

# 4 Systemübersicht

Dieser Paragraph stellt, unabhängig der zu implementierenden Lösung, verschiedene System-Komponenten vor, welche für den oBPM-Prototypen verwendet werden können. Im Kapitel 5. Systemarchitektur und Technologie wird genauer auf verschiedene mögliche Systemarchitekturen eingegangen und deren Vor- und Nachteile erläutert.

## 4.1 Datenbanksystem

Das Datenbank-System ist für die Persistierung und Verwaltung aller dauerhaft zu speichernden Applikationsdaten verantwortlich. Das Hauptkonzept eines Datenbank-Systems besteht in der Abstraktion der Daten und das Offerieren einer Schnittstelle für den Zugriff und die Manipulation der verwalteten Informationen (Silberschatz, Korth, & Sudarshan, 1997, S. 3). Für die Evaluation eines bestehenden Datenbank-Systems spielt vor allem die Strukturierung der Datenabstraktion eine Rolle. Dabei haben sich verschiedenste Konzepte bewährt, wie beispielsweise relationale, dokumentorientierte oder XML-basierte Datenbanksysteme. Wie in der Aufgabenstellung dieses Papiers erläutert, dient dem oBPM-Prototypen eine Arango-Datenbank als Datenquelle. Sie ist hauptsächlich dafür verantwortlich, alle nötigen Daten zu Prozessmodellen und Prozessdaten zu hinterlegen. Zusätzlich kann sie je nach Architekturwahl auch Businesslogik veralten, wie beispielsweise die Abstrahierung von komplexeren Datenabfragen und –manipulationen.

## 4.2 End-User-Client

Der End-User-Client stellt eine Interaktionsschnittstelle zwischen einem Software-system und einem humanen Benutzer dar. Dadurch unterscheidet er sich zu einem softwarebasierten Client, welcher ähnlich einem Endbenutzer mit einem System interagieren kann. Grundsätzlich haben sich heute Clients mit graphischer Oberfläche durchgesetzt, jedoch schränkt die Definition des Clients die Art der Benutzerinteraktion nicht ein. Nebst der Interaktionsmöglichkeit werden Clients auch anhand der verwendeten Architektur unterschieden, vor allem basierend auf der Komplexität der implementierten Businesslogik. Clients, ähnlich Terminals, welche nur für das Anzeigen

und Entgegennehmen externer Daten verantwortlich sind, werden als Thin-Clients bezeichnet. Clients mit eigener Businesslogik oder Datenverwaltung dagegen als Fat-Clients (Fraternali, Rossi, & Sánchez-Figueroa, 2010). Welche Art von Client-Typ verwendet werden soll hängt grundsätzlich stark von der gesamten Systemarchitektur ab. Stellt ein System beispielsweise mehrere Arten von Clients zur Verfügung (Webapplikation, Desktop-Applikation, Mobile-App, etc.), welche jedoch die gleiche Businesslogik teilen, macht es Sinn diese in einen zentralen Service oder Server auszulagern statt sie mehrmals zu implementieren. Die Clients sind in diesem Fall nur noch für die Eingabe und Ausgabe, bzw. Darstellung von Daten verantwortlich.

Im oBPM-Prototypen besitzt ein Client zwei verschiedene Hauptaufgaben. Zum einen muss er dem Benutzer erlauben, über sein graphisches User-Interface bestehende Prozessmodelle anpassen zu können und neue Modelle erstellen zu können. Zum anderen muss der Client dem User graphisch aufzeigen, welche Ausführungen dem aktuellen Benutzer in einem Prozess zur Verfügung stehen und ihm erlauben, Prozessaktionen vorzunehmen. Prozessaktionen können des Weiteren je nach Prozessmodell einen Dateninput erwarten, welcher beispielsweise über ein vom Client zur Verfügung gestelltes Formular vorgenommen werden können.

### 4.3 Application-Server

Ein Application-Server System (App-Server) stellt eine zentrale Applikation dar, welche intern eine eigene Businesslogik betreibt und gegen aussen Schnittstellen bereitstellt, um diese zu benutzen. Besitzt der App-Server keine eigene Datenverwaltung, ist er gezwungenermassen auf eine externe Verwaltung angewiesen, so beispielsweise ein Datenbanksystem. Des Weiteren sind die angebotenen Schnittstellen auf die Interaktion mit anderen Systemen ausgelegt. Somit wir ein Client jeglicher Art benötigt, um mit dem Application-Server interagieren zu können. Deshalb kann ein Application-Server als zentrale Schnittstelle zwischen Datenquelle (Datenbank) und Benutzerinteraktion (Client) gesehen werden, welcher anhand der internen Businesslogik vorgibt, wie Dateninputs und -abfragen von einem Client verarbeitet werden und wie die Response zu einer Anfrage auszusehen hat.

Im oBPM-System wäre ein Application-Server der Hoster der Businesslogik. Seine Aufgabe besteht darin, zentral Anfragen von Clients für die Manipulation von Modellen oder die Ausführung eines Prozesses abzuwickeln. Er ist dafür verantwortlich, dass

Prozesse anhand der Logik des hinterlegten Modells ausgeführt werden. Dies bedeutet, dass der Application-Server Client-Anfragen auf ihre Validität in Bezug auf das Prozessmodell überprüfen muss und bei Gültigkeit die in der Datenbank hinterlegten Prozessdaten und -modelle anpassen muss. Somit ist der Application-Server für die Integrität der Daten in der Datenbank verantwortlich.

# 5 Systemarchitektur und Technologiewahl

Dieses Kapitel zeigt verschiedene Varianten von möglichen Systemarchitekturen auf und beschreibt deren Vor- und Nachteile hinsichtlich Performance, Entwicklungsfreundlichkeit und Wartbarkeit. Dabei kommen die Komponenten, welche im Kapitel 4 Systemübersicht vorgestellt wurden, zur Anwendung.

Der Kern des oBPM-Prototyps besteht in jedem Fall aus einer Arango-Datenbankinstanz, wie dies in der Aufgabenstellung zur Bachelor-Thesis von Grünert (2015) beschrieben ist. Diese verantwortet die Verwaltung der Prozessdaten, die Verarbeitung von Datenabfragen und Manipulationen und stellt die zuvor definierten Prozessmodelle bereit. Im Unterschied zu konventionellen relationalen Datenbanksystemen stellt ArangoDB zusätzlich seinen eigenen Webserver bereit, wodurch das System per HTTP/REST-Requests kontaktiert werden kann. Des Weiteren bietet ArangoDB eine Serviceschnittstelle an, die es erlaubt, Businesslogik direkt im Datenbanksystem unterzubringen (Arango GmbH, 2016c). Diese Services können über eine REST-Schnittstelle direkt von einem Client oder von einem dazwischengeschalteten Application-Server bezogen werden.

## 5.1 Schnittstellen

Unabhängig der Architektur müssen die Schnittstellen und die Art der Kommunikation zwischen den Komponenten definiert werden. Von Vorteil wird auf standardisierte Protokolle gesetzt, welche nicht an eine proprietäre Softwarelösung oder -technologie gebunden sind. Dies erlaubt eine grössere Unabhängigkeit bei der Wahl der Systemkomponenten.

Wie von Anforderung 002 und Anforderung 003 definiert, muss das System REST-basierte Schnittstellen zur Verfügung stellen. Grundsätzlich ist REST nicht an ein spezifisches Protokoll gebunden (Jakl, 2005, S. 7), doch ist das von Fielding u. a. (2006) definierte HTTP-Protokoll das im World Wide Web meist benutzte Protokoll (Jakl, 2005,

S. 8). Zudem muss beachtet werden, dass durch die Verwendung von REST auf Basis von HTTP auch webbasierte Clients als oBPM-Clients infrage kommen. Zum Beispiel webbasierte Applikationen, welche über einen Web-Browser bedient werden können. Somit fällt der Entscheid auf HTTP-basierte Schnittstellen, welche den Konventionen von REST folgen.

### 5.2 Client-Datenbank-Architektur

Diese Architektur beschreibt ein System bestehend aus einem oder mehreren Clients und einer zentralen Datenbank. Die Datenbank selbst übernimmt nicht nur die Verwaltung der Daten, sondern beinhaltet auch die eigentliche Businesslogik, wie Abbildung 3 schematisch darstellt. Da ArangoDB mit Foxx ein eigenes Service-Framework zur Verfügung stellt (Arango GmbH, 2016c), wäre diese Architektur eine mögliche Variante. Die Foxx-Services sind dabei einzelne Javascript-Module, welche mit Hilfe der V8-Javascript-Engine von Google betrieben werden (Arango GmbH, 2015). Jeder Service ist über eine eigene REST-Schnittstelle von einem Client über HTTP ansprechbar und behandelt die Logik für die Verwaltung einzelner Datensätze oder Applikationsmodule. Da der Service selbst in der Umgebung der Datenbank betrieben wird, steht diesem eine dezidierte Datenbankschnittstelle zur Verfügung, um dessen Verwaltung der Service nicht verantwortlich ist (Arango GmbH, 2016c). Nach Installation von ArangoDB hat sich des Weiteren gezeigt, dass die Administrationsoberfläche von ArangoDB das Erstellen von neuen Services, welche bereits die generischen CRUD-Operationen für bestimmte Dokumenttypen bereitstellen, erlaubt. Diese Architektur verzichtet vollständig auf einen dazwischen geschalteten Application-Server. Somit kann ein potentieller Client eine direkte Verbindung zum Datenbanksystem aufbauen, ohne ein Routing über einen zusätzlichen Server. Dies erhöht die Performance des gesamten Systems und reduziert die Anzahl Komponenten, wodurch sich die Installation und Wartung des Systems merklich verbessert.

Wie bereits erwähnt werden die Foxx-Services von ArangoDB in Datenbankumgebung betrieben. Dies setzt voraus, das die Codebase der Services in einem bestimmten Verzeichnis der Datenbankinstanz abgelegt werden müssen, auch während der Entwicklung. Die Javascript-Laufzeitumgebung basiert zwar wie auch Node.js auf der V8-Engine (Arango GmbH, 2015; Node.js Foundation, 2016), jedoch ergeben sich nach genauer Analyse einige signifikante Unterschiede. Zum einen

unterstützt die Foxx-Umgebung kein Debugging. Die einzige Möglichkeit zur Fehlerbehebung während der Entwicklung und der Wartung besteht im Auslesen von Log-Dateien, welche zuvor aufgezeichnete Programmzustände beinhalten können, wie von Arango GmbH (2016e) erläutert. Ein weiterer wichtiger Unterschied besteht in der Thread-Verwaltung. Während Node.js mit Hilfe von LIBUV auf eine asynchrone und eventbasierte Single-Thread Lösung setzt (Nikhil, 2014), verwendet die Laufzeitumgebung der Foxx-Services einen eigenen Thread-Pool und setzt somit auf eine Multithreading-Lösung (Arango GmbH, 2015). Der letzte grosse Unterschied besteht in der Verwendung von Third-Party-Bibliotheken und -Softwaremodulen. Node.js besitzt eine grosse Anzahl von Softwarekomponenten, welche von Dritten erstellt und der Entwicklungscommunity bereitgestellt werden. Ein grosser Teil dieser Komponenten basiert jedoch auf Node.js-internen Schnittstellen, wodurch diese Softwarepakete abgesehen von einigen Ausnahmen nicht in Foxx-Services eingesetzt werden können (Arango GmbH, 2016f). Somit ist der Entwickler eines Foxx-Services hauptsächlich auf die Module und Schnittstellen angewiesen, welche von ArangoDB oder der Community von ArangoDB selbst zur Verfügung gestellt werden.

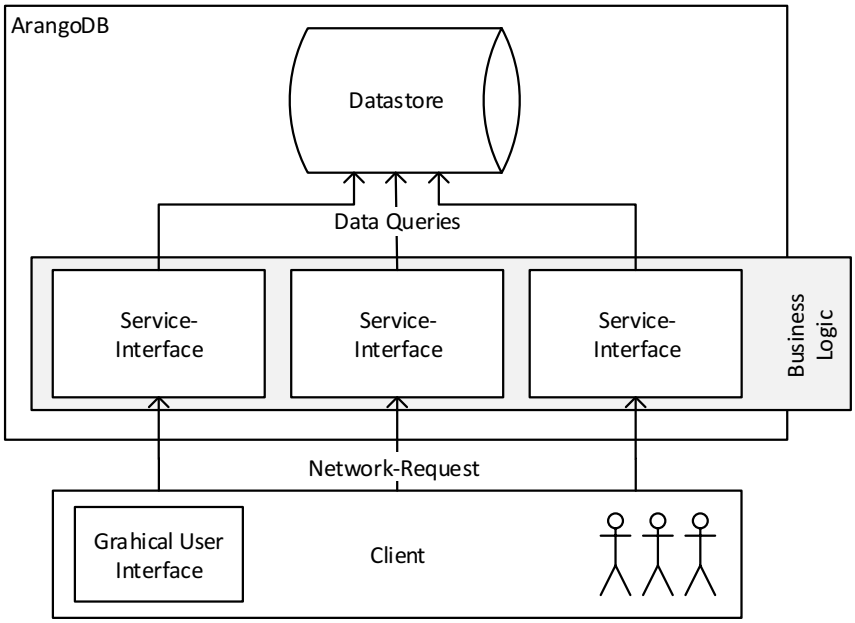


Abbildung 3: Illustration einer Datenbank-Client-Architektur

## 5.3 Application-Server-Architektur

Im Unterschied zur Client-Datenbankarchitektur wird zwischen Client und Datenbank ein zusätzlicher Application-Server eingesetzt. Dieser ist hauptsächlich für die Bereitstellung der Businesslogik verantwortlich, womit das Datenbanksystem alleinig für die Datenverwaltung benötigt wird. Auf der einen Seite stellt der Server die Schnittstellen für den Client bereit, auf der anderen Seite werden die Datenbank-Schnittstellen benutzt, um auf die Datenbasis des Systems zuzugreifen. Da beide Schnittstellen auf standardisierten Protokollen basieren, kann für die Umsetzung des Servers jegliche Art von Technologie eingesetzt werden. Von Vorteil wird dabei auf einen bestehenden Webserver, bzw. ein bestehendes Webframework gesetzt, welches die Verarbeitung von Webbasierten Anfragen von Haus aus verarbeiten kann.

Zwar bietet ArangoDB für jegliche Programmierumgebung bestehende Datenbanktreiber an, jedoch ergibt sich aus der Wahl des Datenbanksystems, dem webbasierten Ansatz und der vorherrschenden Programmiersprache Javascript auf Datenbankebene und Client eine Application-Server Lösung auf Basis von Node.js an. Dies bedeutet, dass von der Datenbank, über den Application-Server, bis zum Client auf dieselbe Technologie gesetzt werden kann. Somit lassen sich externe wie auch interne Codemodule zwischen den verschiedenen Teilsystemen wiederverwenden. In Frage kommen dabei beispielsweise Module zur Datenvalidierung, welche serverseitig wie auch clientseitig eingesetzt werden können. Eine Lösung basierend auf NodeJS bietet zudem einfach umzusetzende Möglichkeiten an, die Nachteile der weiter oben beschriebenen Foxx-Services zu umgehen. So steht der Entwicklung des Servers und damit der  die vollständige Sammlung an Drittherstellerkomponenten bereit, welche für verschiedene Javascript-Umgebungen entwickelt wurden. Auch existiert eine breite Palette an bestehenden Entwicklungs- und Wartungstools, so beispielsweise für das Debugging, Profiling und Monitoring von laufenden Programmen. Die verwendeten Tools werden in Kapitel *5.6 Entwicklungsumgebung* genauer vorgestellt.

NodeJS hat den weiteren Vorteil, dass es plattformunabhängig von seiner asynchronen, eventbasierten Umgebung profitiert, wenn es um Scalability und Performance unter hoher Auslastung geht (Maatouki, Meyer, Szuba, & Streit, 2015).

Schlussendlich setzt der Einsatz eines dezidierten Application-Servers aus eine stärkere Modularisierung, da durch dessen Einsatz die Verwaltung der Daten und der

Businesslogik getrennt wird. Somit können einzelne Komponenten unabhängiger agieren und unabhängig voneinander entwickelt werden. Entscheidet man sich beispielsweise bei der Weiterentwicklung des Projekts ArangoDB durch ein anderes Datenbanksystem zu ersetzen, kann der Application-Server mit minimalen Schnittstellenanpassungen weiterverwendet werden.

Ein Augenmerk bei dieser Lösung muss auf die Request-Performance gesetzt werden, da pro Client-Anfrage mindestens zwei unabhängige Requests ausgelöst werden müssen. Zum einen eine Anfrage vom Client zum Application-Server, zum anderen eine weitere Anfrage von Application-Server zur Datenbank. Auch muss darauf geachtet werden, dass performance-lastige Daten-Queries, wie die mehrfache Filterung von Datensätzen, nicht im Application-Server stattfinden, sondern so nahe wie möglich am Datenbanksystem, da dieses Daten-Queries und -Manipulationen optimierter und damit zeitkritischer ausführen kann. Auch kann es bei komplexeren Datenabfragen unumgänglich sein, eine Filterung in mehrere Queries aufzuteilen. Findet die Aggregation der Daten auf dem Application-Server statt, müssen somit mehrere Requests für eine einzelne Query an das Datenbanksystem gesendet werden, was Auswirkungen auf die Performance hat.

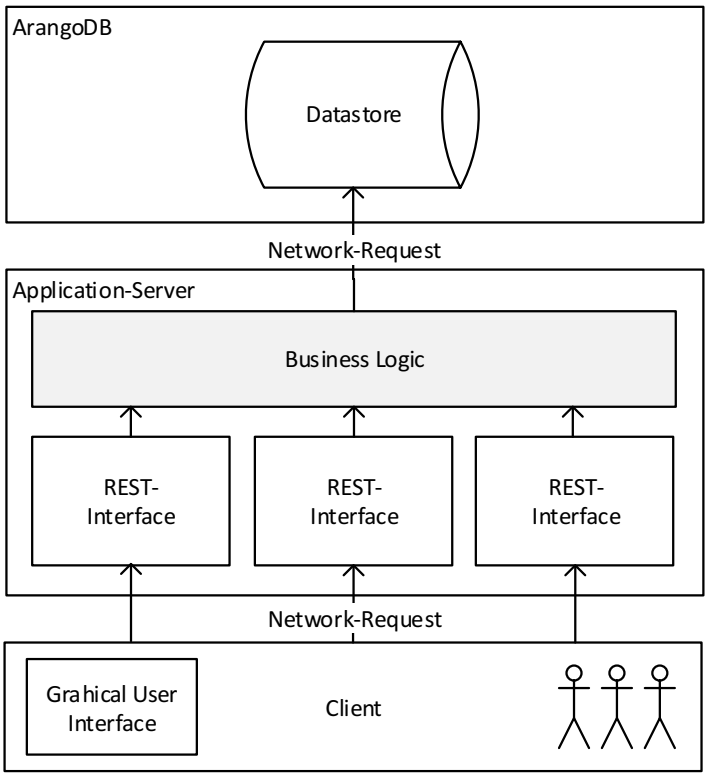


Abbildung 4: Illustration einer Application-Server-Architektur

## 5.4 Architekturentscheid

Beim Vergleich der Vor- und Nachteile der beiden vorgestellten Architekturvarianten hat sich die Architektur basierend auf einem Application-Server klar durchgesetzt. Vor allem der Umstand der schlechten Debugging- und Monitoring-Möglichkeit der Foxx-Services und die Inkompatibilität mit den meisten bestehenden Third-Party Komponenten der Javascript- und NodeJS-Umgebung führen dazu bei, dass der zu entwickelnde Prototyp auf eine Architekturlösung basieren auf einem NodeJS Application-Server setzt. Trotzdem sollen vereinzelt Foxx-Services eingesetzt werden, welche datenbanknahe Datenabfrage- und Manipulations-Algorithmen übernehmen sollen.

## 5.5 Technologie- und Sprachwahl

Die Wahl der Systemkomponenten basierend auf ArangoDB, Node.js und einem bevorzugt webbasierten Client lassen den logischen Schluss zu, bei der Entwicklung auf die Programmiersprache Javascript zu setzen. Jedoch haben sich in den letzten Jahren verschiedene Supersets von Javascript herauskristallisiert. Diese Supersets beschreiben Programmiersprachen, welche zwar ihre eigenen zusätzlichen Features anbieten, jedoch zur Ausführung in Javascript kompiliert werden. Dazu gehört momentan hauptsächlich Coffescript und Typescript. Beide dieser Supersets bieten zusätzliche Sprachfeatures an, welche jedoch nur während der Entwicklung zum Tragen kommen, da grundsätzlich jeder Superset-Code auch Javascript kompatibel sein muss. Die Wahl zwischen Javascript und eines seiner Supersets hat also nur Einfluss auf die Entwicklung selbst, jedoch nicht auf die Art und Weise und die Effizienz der Umsetzung einer programmbasierten Lösung. Coffescript setzt dabei auf eine dezidierte Syntax, inspiriert von Sprachen wie Perl, Phyton oder Ruby und bietet zusätzliche Features an wie List Comprehensions und Pattern-Matching. Typescript dagegen setzt im Gegenzug auf eine starke Typisierung. Es erlaubt die Definition von typisierten Komponenten, wie Variablen, Klassen, Interfaces und Funktionsparametern. Der zur Verfügung gestellte Compiler ist in der Lage falsche oder ungültige Zuweisungen und Codedefinitionen anhand der definierten Typen während der Entwicklung aufzuzeigen. Auch bietet Typescript Sprachfeatures an, welche erst in späteren Versionen von Javascript vorgesehen sind, so beispielsweise Promises, Async/Await-Statements oder Decorators.

Die starke Typisierung erlaubt das Erkennen von Fehlern während der Entwicklungszeit und sorgt im Normalfall für weniger Fehler während der Laufzeit eines

Programms, vor allem dann, wenn es sich um ein grösseres Projekt handelt, bestehend aus einer grossen Zahl interner Komponenten und Modulen. Auch kann die Wartbarkeit der Codebase stark erhöht werden, während gleichzeitig der Dokumentationsaufwand gesenkt werden kann.

Die Wahl der Programmiersprache fällt dabei auf Typescript, da die oben beschriebenen Vorteile gegenüber Javascript klar überwiegen. Die starke Typisierung wird des Weiteren stärker gewichtet als die zusätzlichen Features, welche von Coffee-Script angeboten werden. Weitere Javascript-Supersets werden in dieser Arbeit nicht evaluiert, da ihre Verbreitung zu gering ist und sie sich zuerst als eine stabile Entwicklungsgrundlage bestätigen müssen.

### 5.6 Entwicklungsumgebung

Vor dem Beginn der eigentlichen Entwicklung werden die Tools und Programme definiert, welche für die Programmierung, das Debugging und das Testing verwendet werden sollen. Die gesamte Sammlung der Entwicklungsprogramme wird in diesem Papier unter dem Namen Toolchain zusammengefasst.

Dabei gibt es zwei Grundsätze. Zum einen der Einsatz von vollständigen Entwicklungsumgebungen (IDEs), welche interne Tools und Zusatzprogramme für den gesamten Entwicklungsprozess bereitstellen. Durch deren Einsatz werden meist nur wenige weitere externe Programme benötigt, jedoch ergibt sich eine starke Bindung und die von der IDE zur Verfügung gestellten Tools und eine Einschränkung auf die von der IDE unterstützen Prozesse.

Die zweite Variante besteht im Einsatz einer Lösung einzelner unabhängiger Programme, von welchen jedes eine dezidierte Aufgabe übernimmt. Diese Tools sind dabei auf offene Schnittstellen und Prozesse angewiesen, um miteinander interagieren zu können. Diese Lösung bietet vor allem den Vorteil unabhängiger Module, welche unabhängig vom System und Umgebung eingesetzt, ersetzt und separat verwendet werden können. Für die Umsetzung des Prototypen wird auf folgende Tools zur Entwicklung gesetzt: Der Texteditor Atom mit integriertem Typescript-Package zur Erfassung des Source-Code, der von Microsoft zur Verfügung gestellten Typescript-Compiler zur Übersetzung des Codes zu Javascript, Nodemon als NodeJS-Entwicklungsumgebung, Node-Inspector für das Debugging, Postman für das Testing

und Profiling der REST-Schnittstellen, Mocha für die Umsetzung von Unit-Tests und schlussendlich ein produktives ArangoDB-System.

# 6 Application-Server

Dieser Abschnitt des Papiers beschreibt die Entscheidungsfindung und den Aufbau der NodeJS-basierten Server-Applikation. Zuerst wird in diesem Kapitel auf die Entscheidungsfindung eingegangen und welche Kriterien für die schlussendliche Architektur verantwortlich waren. Anschliessend wird ein Überblick über die gesamte Applikation vorgestellt um danach auf die einzelnen wichtigsten Komponenten genauer einzugehen. Schlussendlich werden die Schnittstellen im Detail beschrieben und wie sie von einem Client benutzt werden können um mit dem oBPM-System zu interagieren.

## 6.1 Entscheidungsgrundlagen zur Architektur

Da es sich beim vorliegenden Prototypen um eine erste Implementation des von Grünert u. a. (o. J.-b) vorgestellten Systems handelt, kann davon ausgegangen werden, dass bei einem positiven Resultat dieser Arbeit der entwickelte Prototyp weiterhin gewartet und zusätzlich ausgebaut wird. Somit wurde bei der Strukturierung der Server-Applikation das Augenmerk auf Wartbarkeit und Erweiterbarkeit gelegt. Um die Wartbarkeit zu erhöhen, wurde auf möglichst viele bereits vorhandene Softwaremodule gesetzt, welche sich zum einen in aktiver Entwicklung befinden und zum anderen eine bereits aktive Community nachweisen können. Diese Faktoren unterstützen und vereinfachen die Wartung und Weiterentwicklung des oBPM-Systems beträchtlich. Als Kennzahl zur aktiven Verwendung wurde dabei unter anderem auf die Anzahl Downloads in der vergangenen Zeit gesetzt, welche auf den meisten Softwareportalen ersichtlich ist. Die aktive Weiterentwicklung eines Moduls wurde anhand der letzten Veränderungen durch die Autoren überprüft. Diese Information ist bei Open-Sourcemodulen in den jeweiligen Code-Repositories verfügbar. Um eine mögliche Übergabe der Projektverantwortlichkeit zur Wartung und Erweiterung des Programms so einfach wie möglich zu gestalten, wurde zusätzlich auf bekannte Best-Practices und standardisierte Application-Patterns gesetzt. Diese sind in Kapitel *6.2 Applikationsufbau*

genauer erläutert. Schlussendlich wurde mit Hilfe von automatisierten API-Tests eine möglichst hohe Testabdeckung erreicht, wie das Kapitel *9.1 API-Tests* genauer erläutert.

Dies unterstützt die zukünftige Erweiterung des Systems, da bei Änderungen des bestehenden Codes die Funktionalität automatisch überprüft werden kann.

Um die Erweiterbarkeit des Systems zu unterstützen, wurde nach einer Struktur gesucht, welche eine möglichst starke Modularisierung des gesamten Systems erlaubt und zusätzlich bekannten Programm-Patterns folgt. Kapitel *6.2 Applikationsufbau*

zeigt die verwendete Programmstruktur auf, welche die zuvor beschrieben Kriterien erfüllt.

## 6.2 Applikationsufbau

Dieses Kapitel beschreibt den internen Aufbau der Businesslogik im Application-Server des oBPM-Systems. Um einen Überblick zu gewährleisten, zeigt Abbildung 5 den schematischen Aufbau des Application-Servers auf. Die darauffolgenden Unterkapitel erläutern jede einzelne Code-Komponente des Schemas im Detail und erklären als Ganzes den Zusammenhang zwischen den einzelnen Modulen des Application-Servers.

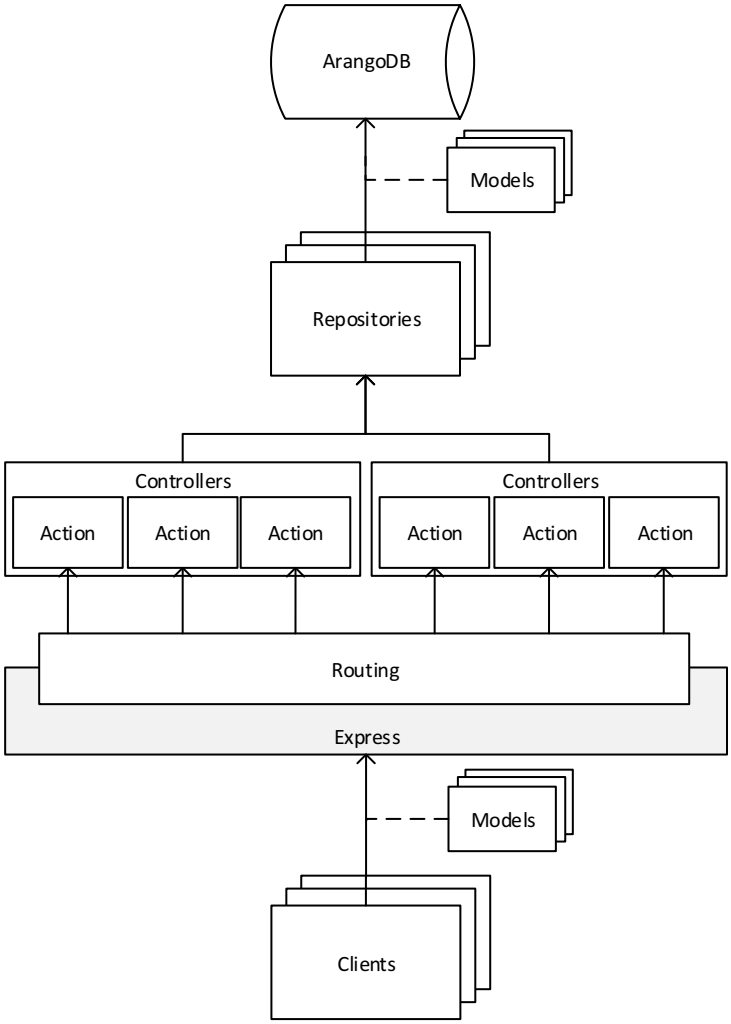


Abbildung 5: Schematische Architektur des Application-Servers

### 6.2.1 Repositories

Die eigentliche Businesslogik zur Abfrage und Manipulation von Daten befindet sich in Repository-Module, bzw. -Klassen. Alle verfügbaren Repository-Module befinden sich im Ordner *api/lib/repositories*. Sie sind hauptsächlich strukturiert nach den in Kapitel *7.2 Implementation* beschriebenen Datenstrukturen. Repositories bieten je nach Datenstruktur verschiedene Funktionen an, welche wiederum von anderen Code-Modulen verwendet werden können, um Daten abzufragen oder zu manipulieren.

### 6.2.2 Controllers

Die Komponenten des Typs Controller bieten die eigentlichen API-Schnittstellen für externe Clients an. Bei jedem Controller-Modul handelt es sich um eine Softwareklasse, welche eine beliebige Anzahl Methoden implementiert, die in dieser Arbeit *Actions*

genannt werden. Diese Actions können mit Hilfe des Request-Routings wie in Kapitel *6.2.5 Routing* beschrieben von einem externen Client über einen HTTP-Request angesprochen werden. Die gewählte Architektur basierend auf Controllern und Actions ist dabei stark angelehnt an das bekannte MVC-Pattern, welches bereits von Selfa, Carrillo, & Boone (2006, S. 4) genauer erläutert wurde. Die Strukturierung der Controller hält sich dabei ähnlich den Repositories stark an die in Kapitel *7.2 Implementation* beschriebenen Datenstrukturen.

Die einzelnen Implementationen der Actions dienen wie bereits erläutert nur als Schnittstelle gegen aussen. Sie beinhalten so wenig Businesslogik wie möglich und benutzen die jeweiligen Repositories um eine Anfrage eines Clients zu verarbeiten. Diese Auslagerung der Logik erhöht die Wiederverwendbarkeit des Codes, da möglicherweise mehrere Actions von verschiedenen Controllern Zugriff auf dieselbe Logik benötigen.

Alle Controller-Klassen befinden sich im Ordner *api/lib/controllers*.

Ein grosser Vorteil der Struktur basierend auf Controllern und Actions liegt in der Möglichkeit der Vererbung von Softwareklassen. Generische Controller-Implementationen, wie beispielsweise die Klasse *RepositoryController,* können ihre Action an weitere Controller-Klassen vererben, wodurch diese nur einmalig implementiert werden müssen. Ein gutes Szenario für diese Art der Implementation stellen die Actions für die gängigen CRUD-Operationen (Create, Read, Update, Delete) dar. Diese werden in den meisten Controllern benötigt und können somit von einem generischen Controller vererbt werden.

### 6.2.3 Models

Die Code-Module des Typs Model definieren den Aufbau von Datenstrukturen, welche zwischen Client und Server ausgetauscht werden können. Jede Modelklasse stellt dabei eine Methode *getSchema* bereit, welche das Objektschema des jeweiligen Model-Typs zurückgibt. Dieses Schema kann verwendet werden, um vom Client gesendete Datenstrukturen nach ihrer Validität zu überprüfen. Die Strukturierung der Model-Klassen, bzw. Model-Typen lehnt sich dabei stark an die in Kapitel *7.2 Implementation* definierten Datenstrukturen an und befinden sich im Ordner *api/lib/models*. Zusätzlich werden im Ordner *api/lib/viewmodels* weitere Model-Typen deklariert, welche unabhängig der verwendeten Datenstruktur im ArangoDB-Datenstore sind, und nur für die Kommunikation zwischen Client und Application-Server verwendet werden.

### 6.2.4 Express-Framework

Das von StrongLoop & IBM (2016) entwickelte Express-Framework dient als Basis einer Webapplikation. Es kümmert sich um die grundlegende Verarbeitung der einkommenden HTTP-Requests und sorgt dafür, dass alle Code-Module, welche für die Weiterverarbeitung eines Requests nötig sind, in der zuvor definierten Reihenfolge im Kontext eines Requests aufgerufen werden. Des Weiteren übernimmt Express die Verarbeitung und das Senden einer HTTP-Antwort an einen Client. Express bietet auch eigene Middleware-Funktionen an, beispielsweise für die Auslieferung von statischen Dateien oder für das Routing von eintreffenden Requests anhand vordefinierten URL-Patterns (StrongLoop & IBM, 2016b, 2016c).

Express verzeichnete im vergangenen Monat über 6.1 Millionen Downloads, davon 234'490 Downloads während dem letzten Tag, verzeichnet am 21.05.2016 (npm Inc., 2016). Des Weiteren kann mit Hilfe der Abbildung 6, welche die Anzahl Contributions in Form von Commits in das Code-Repository von Express über eine Zeitachse von 7 Jahren aufzeigt, abgeleitet werden, dass sich Express in aktiver Entwicklung befindet. Diese zwei Kenntnisse unterstützen klar den Einsatz von Express im oBPM-System.


Abbildung 6: Anzahl Code-Commits zum Express-Framework (Github Inc., 2016)

Folgende Kapitel erläutern einzelne verwendete Funktionalitäten des Express-Frameworks und wie sie für das oBPM-System eingesetzt wurden.

#### 6.2.4.1 Middleware-Funktionen

Dieses Kapitel erläutert den Aufbau und Zweck der Express-Middleware-Funktionen und wie diese im oBPM-System verwendet wurden.

Um die oBPM-Umgebung in Express zu integrieren, wurde auf das Konzept der Middleware-Funktionen gesetzt, auf welchem Express aufbaut (StrongLoop & IBM, 2016d). Middleware-Funktionen sind dabei Funktionen, welche für jeden eintreffenden Request von Express aufgerufen werden und beliebige Aktionen vornehmen können. Bei

ihrer Ausführung erhalten sie Zugriff auf den aktuellen Client-Request und Client-Response. Jede Middleware-Funktion hat nach deren Ausführung die Möglichkeit die Verarbeitung des aktuellen Requests abzubrechen oder den Verarbeitungsprozess an die nächste Middleware-Funktion weiterzureichen. Middleware-Funktionen werden dabei immer in der Reihenfolge aufgerufen, wie sie auch registriert wurden (StrongLoop & IBM, 2016d). Code 1 zeigt als Beispiel die Registrierung der Middleware-Funktion in Express, welche von `morgan()` zur Verfügung gestellt wird.

```
/**
 * Logging-Middleware.
 * Logs each incoming request to stdout and
 * calls the next middleware.
 */
this.app.use(morgan('combined'));
```

Code 1: Beispiel einer Middleware-Registrierung

#### 6.2.4.2 Routing-Middleware

Dieses Kapitel erläutert den Einsatz von Routing-Middleware, was ein Spezialfall von Middleware-Funktionen darstellt. Routing-Middleware-Funktionen werden nicht für jeden eintreffenden Request aufgerufen, sondern nur für Requests, welche einem zuvor definierten Request-Pattern entsprechen. Dazu gehört die URL und die HTTP-Methode. Code 2 zeigt ein Beispiel einer Registrierung einer Routing-Middleware-Funktion auf. Bei dieser Registrierung wird festgelegt, dass die Middleware-Funktion von `this.oauth.grant()` nur für Requests ausgeführt werden, welche über HTTP-POST gesendet wurden und dem URL-Pattern `/oauth/token` entsprechen.

```
// executes this.oauth.grant() for authentication requests:
this.app.post('/oauth/token', this.oauth.grant());
```

Code 2: Beispiel einer Route-Middleware-Registrierung

### 6.2.5 Routing

Das Routing oder genauer Request-Routing ist dafür verantwortlich, dass eintreffende HTTP-Requests von Clients an die richtige Action eines Controllers weitergeleitet wird. Alle dafür verantwortlichen Code-Module befinden sich im Ordner *api/lib/routing*.

Um einen externen Request erfolgreich an die passende Action eines Controllers weiterzuleiten, müssen die Routing-Module feststellen können, welche Action von

welchem Controller angesprochen werden soll. Dazu werden in einem ersten Schritt alle nötigen Routing-Middleware-Funktionen, wie in Kapitel *6.2.4.2 Routing-Middleware* erläutert, erfasst. Tabelle 3 zeigt alle registrierten Request-Routes auf, welche vom oBPM-System verarbeitet werden.

Tabelle 3: Registrierte Request-Routes

| **URL-Pattern** | **Controller-Name** | **Action-Name** |
|---|---|---|
| /user/:id | UserController | HTTP-Methode |
| /user/:action | UserController | Action-Parameter |
| /user | UserController | HTTP-Methode |
| /:process | EnvironmentController | HTTP-Methode |
| /:process/:controller/:id | Controller-Parameter | HTTP-Methode |
| /:process/:controller/:action | Controller-Parameter | Action-Parameter |
| /:process/:controller | Controller-Parameter | HTTP-Methode |

Welcher Controller vom Routing für die Verarbeitung benutzt wird, kann entweder pro Request-Route hard-codiert werden, wie dies bei */user/:id*, */user* und */:tid* der Fall ist, oder wird automatisch aus dem Route-Parameter *:controller* abgeleitet. Die aufzurufende Action wird entweder über den Route-Parameter *:action* oder über die verwendetet HTTP-Methode des Requests abgeleitet.

Alle in Tabelle 3 aufgeführten Request-Routen werden vom oBPM-internen Router verarbeitet, welcher unter *api/lib/routing/Router.ts* abgelegt ist.

## 6.3 API-Schnittstellen

Dieses Kapitel zeigt alle verfügbaren Schnittstellen des Application-Servers auf, welche von einem Client konsumiert werden können. Das erste Unterkapitel beschreibt die Schnittstellen zur Authentifizierung gegen die oBPM-API. Die von der Authentifizierung erhaltenen Identifizierungsdaten müssen bei allen folgenden Schnittstellenanfragen mitgesendet werden. Das Kapitel *6.3.2 Globale* beschreibt dazu globale Request-Headers, welche für alle Anfragen mitgesendet werden müssen, mit Ausnahme von Requests an die Authentifizierungs-API. Alle darauffolgenden Kapitel

zeigen nach Thematik gegliedert die detaillierte Beschreibung jedes verfügbaren API-Endpunkts auf.

Die Schnittstellenbeschreibungen in den folgenden Kapiteln folgen dabei den Konventionen, welche in der Auflistung 3 beschrieben werden.

- URL: Beschreibt die relative URL der Schnittstelle ohne Hostname des Application-Servers.
- Headers: Bezeichnet eine Liste von Header-Namen und ihren jeweiligen Werten.
- Pfad-Parameter: Beschreibt alle notwendigen Parameterwerte des jeweiligen URL-Pfads.
- Body: Beschreibt die jeweiligen zu sendenden Request-Daten. Das Datenformat hängt vom jeweiligen Header *Content-Type* ab.
- Response: Alle API-Antworten sind in JSON formatiert, so auch alle möglichen Fehlermeldungen.
- Werte in eckigen Klammern ([]) beschreiben Variablen, welche mit einem gültigen Wert ersetzt werden müssen.

Auflistung 3: Konventionen für API-Schnittstellen-Beschreibungen

Das Kapitel *15.3 API-Testszenarios* zeigt des Weiteren für die wichtigsten Schnittstellen einen Beispielaufruf auf, um die Verwendung der Schnittstellen zu verdeutlichen.

### 6.3.1 Authentifizierung

Dieses Kapitel zeigt auf, wie der Authentifizierungs-Endpunkt des oBPM-Systems verwendet werden muss, um sich für darauffolgende API-Requests authentifizieren zu können. Die oBPM-API bietet eine einzige Schnittstelle zur Authentifizierung an, welche in Tabelle 4 beschrieben ist.

Tabelle 4: API-Schnittstelle /oauth/token

| URL | HTTP-Methode |
|---|---|
| /ouath/token | POST |
| Headers | |
| Authorization | Basic + base64([client_id]:[client_secret]) |
| Content-Type | application/x-www-form-urlencoded |

| Body | |
|---|---|
| Form-Field: grant_type | password |
| Form-Field: username | [Benutzer_Name] |
| Form-Field: password | [Benutzer_Password] |
| Response | |
| token_type | Besitzt immer den Wert „bearer“ |
| access_token | Der zukünftig zu verwendende Access-Token, wie in Kapitel *6.3.2 Globale* beschrieben. |
| expires_in | Gültigkeitsdauer des Access-Tokens in Sekunden. |
| Errors | |
| invalid_client | Ungültige Client-ID oder Client-Secret |
| invalid_grant | Ungültiger Benutzername oder ungültiges Passwort |

Ein Beispiel zur Benutzung der oben beschriebenen Schnittstelle kann im Anhang unter Kapitel *15.3 API-Testszenarios* gefunden werden.

### 6.3.2 Globale Schnittstellen-Eigenschaften

Dieses Kapitel beschreibt globale Schnittstellen-Eigenschaften, welche auf alle folgenden Schnittstellen zutreffen. Dazu gehören die HTTP-Headers, welche in jedem Request mit Ausnahme von Authentifizierung-Requests gesendet werden müssen. Die Header dienen dabei zum einen der Identifizierung eines authentifizierten Benutzers und zum anderen der Definition des Datenformats des Requests. Des Weiteren definiert dieses Kapitel alle global auftretenden Fehler und deren Beschreibung. Schlussendlich werden in diesem Kapitel Pfadparameter beschrieben, welche in den meisten API-Schnittstellen benötigt werden. Alle Standardeigenschaften der API-Schnittstellen werden in Tabelle 5 zusammengefasst.

Tabelle 5: Standard-Eigenschaften für API-Anfragen

| Headers | |
|---|---|
| Authorization | Beinhaltet einen String zusammengesetzt aus dem String „*Bearer* “ und einem gültigen Access-Token. Beispiel: `„Bearer 40f3f1e1641d05eae5695dd...“` |
| Content-Type | Muss immer den Wert *application/json* besitzen. |
| Pfad-Parameter | |
| :process | Der Name des Prozesses, für den die jeweilige Anfrage ausgeführt werden soll. |
| :id | Die ID des Datensatzes, für dessen die jeweilige Anfrage ausgeführt werden soll. |
| Errors | |
| invalid_token | Es wurde kein gültiger Access Token im Authorization-Header mitgesendet. |
| unknown_error | Ein unbekannter und unbehandelter Fehler ist aufgetreten. |
| not_found | Die gesuchte Ressource konnte nicht gefunden werden. |
| invalid_execution | Allgemeine Fehlermeldung bei der fehlerhaften Ausführung einer Schnittstelle. |
| not_implemented | Die aufgerufene Schnittstelle ist nicht implementiert. |
| not_authorized | Der aktuelle Benutzer ist nicht berechtigt für die Ausführung der Schnittstelle. |
| route_not_found | Die angefragte Schnittstelle konnte nicht gefunden werden. |
| validation_error | Bei der Ausführung sind ein oder mehrere Validierungsfehler aufgetreten. |

### 6.3.3 Schnittstellen für die Benutzer- und Rollenverwaltung

Diese Sektion beschreibt alle verfügbaren Schnittstellen für die Verwaltung von Benutzern und ihren Rollen. Dazu gehört das Erstellen, Ändern und Löschen von Benutzern, aber auch dezidierte Schnittstellen zur Änderung von Passwörtern.

#### 6.3.3.1 Benutzer erstellen

Die Tabelle 6 in diesem Kapitel zeigt die Definition der Schnittstelle zur Erstellung von neuen Benutzern auf.

Tabelle 6: API-Schnittstelle POST /user

| URL | HTTP-Methode |
|---|---|
| /user | POST |
| Benötigte Rollen | admin |
| Body | |
| JSON-Model | View-Model des Typs NewUser |
| Response | |
| _key | Die ID des erstellten Benutzers |
| Errors | |
| dublicate_user | Es existiert bereits ein Benutzer mit demselben Benutzernamen. |

#### 6.3.3.2 Benutzer editieren

Diese Schnittstelle dient zur Editierung von bestehenden Benutzern. Der Benutzernamen und das Passwort können nicht über diese Schnittstelle geändert werden. Der Benutzername ist unveränderbar. Das Passwort eines Benutzers kann jedoch über die API-Schnittstelle *6.3.3.4 Benutzer-Passwort ändern* geändert werden. Die Definition der Schnittstelle ist in Tabelle 7 zusammengefasst.

Tabelle 7: API-Schnittstelle PUT /user/:id

| URL | HTTP-Methode |
|---|---|
| /user/:id | PUT |
| Benötigte Rollen | admin |
| Body | |
| JSON-Model | View-Model des Typs UpdateUser |

#### 6.3.3.3 Benutzer löschen

Dieses Kapitel fasst in Tabelle 8 die Definition der Schnittstelle zur Löschung von bestehenden Benutzern zusammen.

Tabelle 8: API-Schnittstelle DELETE /user/:id

| URL | HTTP-Methode |
|---|---|
| /user/:id | DELETE |
| Benötigte Rollen | admin |

#### 6.3.3.4 Benutzer-Passwort ändern

Um das Passwort eines Benutzers zu ändern, muss die in Tabelle 9 definierte Schnittstelle verwendet werden. Dabei muss wie im Schema unter *7.2.5.3 UpdatePasswort View-Model* definiert nicht nur das neue Passwort, sondern auch das originale Passwort mitgesendet werden, um zu verhindern, dass sich Benutzer gegenseitig die Passwärter ändern können.

Tabelle 9: API-Schnittstelle POST /user/changepassword

| URL | HTTP-Methode |
|---|---|
| /user/changepassword | POST |
| Benötigte Rollen | keine |
| Body | |
| JSON-Model | Model des Typs UpdatePasswordViewModel |

### 6.3.4 Schnittstellen für die Datenmodellierung

Dieses Kapitel erläutert alle verfügbaren Schnittstellen des oBPM-Systems, welche für die Verwaltung eines Prozessmodells verwendet werden können. Beim angegebenen Prozessparameter kann es sich dabei um einen bestehenden wie auch um einen neuen Prozessnamen handeln. Das oBPM-System kümmert sich automatisch um die Erstellung der Umgebung für einen neuen Prozess.

#### 6.3.4.1 Datentyp erstellen

Über die in Tabelle 10 definierte Schnittstelle kann ein neuer Dokumenttyp in einem Prozessmodell definiert werden. Dabei ist zu beachten, dass durch die Datenvalidierung des View-Models *ModelDocument* vorausgesetzt wird, dass in einem neuen Prozessmodell immer als erstes ein Dokumenttyp *Case* erstellt wird. Das Kapitel *7.2.5.4 ModelDocument View-Model* beschreibt das Schema des zu sendenden JSON-Models.

Tabelle 10: API-Schnittstelle POST /:process/datamodel

| URL | HTTP-Methode |
|---|---|
| /:process/datamodel | POST |
| Benötigte Rollen | modeler |
| Body | |
| JSON-Model | Model des Typs ModelDocument |
| Errors | |
| dublicate_case | Dieser Fehler tritt auf, wenn pro Prozess mehrere Datentypen vom Typ Case erstellt werden. |

#### 6.3.4.2 Datentyp editieren

Wenn ein bestehender Dokumenttyp über die Schnittstelle in Tabelle 11 editiert wird muss beachtet werden, dass der Dokumenttyp Case nicht bearbeitet werden kann, da dadurch eine Inkonsistenz der Daten entstehen kann. So wäre es beispielsweise möglich ein Datenmodell zu generieren, welches keinen Dokumenttyp *Case* besitzt. Alle anderen Typen können nach Belieben angepasst werden, jedoch kann kein Dokumenttyp zum Typ *Case* geändert werden.

Tabelle 11: API-Schnittstelle PUT /:process/datamodel/:id

| URL | HTTP-Methode |
|---|---|
| /:process/datamodel/:id | PUT |
| Benötigte Rollen | modeler |
| Body | |
| JSON-Model | Model des Typs ModelDocument |
| Errors | |
| dublicate_case | Dieser Fehler tritt auf, wenn pro Prozess mehrere Datentypen vom Typ Case erstellt werden. |
| edit_case | Es wurde versucht den Dokumenttyp *Case* zu bearbeiten oder ein Dokumenttyp zum Typ *Case* zu ändern. |

#### 6.3.4.3 Datentyp löschen

Wenn über die Schnittstelle in Tabelle 12 ein Dokumenttyp aus dem Prozessmodell gelöscht wird muss beachtet werden, dass alle Entitäten, welche als Child-Entitäten an den zu löschenden Typ angehängt sind, auch gelöscht werden, da ansonsten das Prozessmodell in mehrere Modelle aufgeteilt werden würde.

Tabelle 12: API-Schnittstelle DELETE /:process/datamodel/:id

| URL | HTTP-Methode |
|---|---|
| /:process/datamodel/:id | DELETE |
| Benötigte Rollen | modeler |

#### 6.3.4.4 Datentypen auslesen

Die Schnittstelle in Tabelle 13 erlaubt es einem Client alle Informationen eines einzelnen Dokumenttyps auszulesen. Die Schnittstelle in Tabelle 14 hingegen gibt eine Liste aller definierten Dokumenttypen zurück.

Tabelle 13: API-Schnittstelle GET /:process/datamodel/:id

| URL | HTTP-Methode |
|---|---|
| /:process/datamodel/:id | GET |
| Benötigte Rollen | modeler |

Tabelle 14: API-Schnittstelle GET /:process/datamodel

| URL | HTTP-Methode |
|---|---|
| /:process/datamodel | GET |
| Benötigte Rollen | modeler |

#### 6.3.4.5 Prozessmodell abrufen

Um ein gesamtes Datenmodell in Form eines Trees abzurufen, kann die Schnittstelle in Tabelle 15 verwendet werden.

Tabelle 15: API-Schnittstelle GET /:process/datamodel/tree

| URL | HTTP-Methode |
|---|---|
| /:process/datamodel/tree | GET |
| Benötigte Rollen | modeler |

### 6.3.5 Schnittstellen für die Verwaltung von Prozess-Actions

Dieses Kapitel beschreibt alle Schnittstellen zur Erstellung, Abfrage und Manipulation von Prozess-Action-Definitionen. Die Tabelle 16, Tabelle 17, Tabelle 18, Tabelle 19 und Tabelle 20 fassen alle Schnittstellendefinitionen zur Manipulation von Action-Definitionen zusammen.

#### 6.3.5.1 Action erstellen

Tabelle 16: API-Schnittstelle POST /:process/action

| URL | HTTP-Methode |
|---|---|
| /:process/action | POST |
| Benötigte Rollen | modeler |
| Body | |
| JSON-Model | Model des Typs Action |

#### 6.3.5.2 Action editieren

Tabelle 17: API-Schnittstelle PUT /:process/action/:id

| URL | HTTP-Methode |
|---|---|
| /:process/action/:id | PUT |
| Benötigte Rollen | modeler |
| Body | |
| JSON-Model | Model des Typs Action |

#### 6.3.5.3 Action löschen

Tabelle 18: API-Schnittstelle DELETE /:process/action/:id

| URL | HTTP-Methode |
|---|---|
| /:process/action/:id | DELETE |
| Benötigte Rollen | modeler |

#### 6.3.5.4 Einzelne Action auslesen

Tabelle 19: API-Schnittstelle GET /:process/action/:id

| URL | HTTP-Methode |
|---|---|
| /:process/action/:id | GET |
| Benötigte Rollen | modeler |

#### 6.3.5.5 Alle Actions auslesen

Tabelle 20: API-Schnittstelle GET /:process/action

| URL | HTTP-Methode |
|---|---|
| /:process/action | GET |
| Benötigte Rollen | modeler |

### 6.3.6 Schnittstellen für die Ausführung von Prozess-Actions

Dieses Kapitel beschreibt alle Schnittstellen zur Ausführung von Actions in einem bestehenden Prozess.

#### 6.3.6.1 Abfragen von ausführbaren Actions

Diese Schnittstelle in Tabelle 21 dient dem Auslesen einer Liste aller für den aktuellen Benutzer ausführbaren Prozess-Actions. Das Resultat der Anfrage besteht aus einer Liste aller ausführbarer Actions mit zusätzlichen Informationen zu den erforderlichen Dokumenten für die jeweilige Ausführung. Setzt eine Action ein bestehendes Dokument voraus, werden zudem alle bestehenden Dokumente mitgeliefert, welche den Anforderungen betreffend Status und Typ entsprechen. Des Weiteren kann eine Action in mehrere Cases unterteilt sein, wenn die Ausführung einen Kontext in Form eines Cases voraussetzt.

Tabelle 21: API-Schnittstelle GET /:process/action/executables

| URL | HTTP-Methode |
|---|---|
| /:process/action/executables | GET |
| Benötigte Rollen | Keine |

#### 6.3.6.2 Action ausführen

Die Schnittstelle in Tabelle 22 erlaubt es Benutzern eine spezifische Action auszuführen. Die serverseitige Validierung sorgt dafür, dass eine Ausführung nur dann erfolgreich durchgeführt wird, wenn der aktuelle Benutzer zur Ausführung berechtigt ist und alle gesendeten Dokumentdaten und –IDs der Definition der Action und dem Prozessmodell entsprechen. Das zu sendende Model ist in Kapitel *7.2.5.5 Execution View-Model* genauer spezifiziert.

Tabelle 22: API-Schnittstelle POST /:process/execution

| URL | HTTP-Methode |
|---|---|
| /:process/execution | POST |
| Benötigte Rollen | keine |
| Body | |
| JSON-Model | Model des Typs Execution |
| Errors | |
| invalid_case | Die angegebene Case ID ist ungültig. |
| not_authorized | Der aktuelle Benutzer ist nicht berechtigt die Prozess-Action auszuführen. |
| missing_case | Es wurde keine Case ID übergeben |
| missing_state | Es wurde weder ein state noch ein endState für ein Dokument in der Action-Definition hinterlegt. |
| missing_doc | Ein von der Action erwartetes Dokument wurde nicht übergeben. |
| missing_identifier | Eine erwartete ID eines Dokuments wurde nicht übergeben. |
| wrong_state | Ein übergebenes Dokument befindet sich in einem ungültigen Status. |
| invalid_path | In der Action-Definition wurde für ein Dokument ein ungültiger Pfad hinterlegt. |
| invalid_child | Das Datenmodell verbietet das Erstellen eines Dokuments unter dem in der Action-Definition angegebenen Dokument-Pfad. |

### 6.3.7 Schnittstellen für die Verwaltung von Prozessdaten

Dieses Kapitel fasst in Tabelle 23, Tabelle 24, Tabelle 25 und Tabelle 26 alle Schnittstellen zusammen, welche für die Verwaltung von Prozessdaten verwendet werden können. Dazu gehören Prozessdokumente, welche durch die Ausführung von Prozess-Actions entstanden sind, jedoch auch Dokumenten-Records, welche die Veränderung von Dokumenten aufzeichnen. Prozessdokumente und Dokument-Records

können dabei über die Schnittstellen gelesen werden, jedoch nicht bearbeitet oder gelöscht werden. Dies wird während der Ausführung von Prozess-Actions automatisch vorgenommen. Des Weiteren können die in diesem Kapitel definierten Schnittstellen nur von Benutzern der Benutzer-Rolle *admin* ausgeführt werden, da das manuelle Auslesen von Dokumenten nur aus administrativen Zwecken zugelassen ist. Im Normalfall erhält ein Benutzer alle benötigten Informationen eines Dokuments zur Ausführung einer Action über die Schnittstelle *6.3.6.1 Abfragen von ausführbaren Actions*.

#### 6.3.7.1 Einzelnes Prozessdokument auslesen

Tabelle 23: API-Schnittstelle GET /:process/document/:id

| URL | HTTP-Methode |
|---|---|
| /:process/document/:id | GET |
| Benötigte Rollen | admin |

#### 6.3.7.2 Alle Prozessdokumente auslesen

Tabelle 24: API-Schnittstelle GET /:process/document

| URL | HTTP-Methode |
|---|---|
| /:process/document | GET |
| Benötigte Rollen | admin |

#### 6.3.7.3 Einzelnes Dokument-Records auslesen

Tabelle 25: API-Schnittstelle GET /:process/record/:id

| URL | HTTP-Methode |
|---|---|
| /:process/record/:id | GET |
| Benötigte Rollen | admin |

#### 6.3.7.4 Alle Dokument-Records auslesen

Tabelle 26: API-Schnittstelle GET /:process/record

| URL | HTTP-Methode |
|---|---|
| /:process/record | GET |
| Benötigte Rollen | admin |

# 7 Datenmodellierung

Dieses Kapitel zeigt die verwendete Strukturierung der Applikations-Daten auf, welche im ArangoDB-System hinterlegt werden. Dazu wird zuerst der Funktionalitätsumfang und die Datenabstrahierung von ArangoDB analysiert, um auf dieser Basis eine Entscheidungsfindung und Herleitung zur verwendeten Datenstruktur aufzeigen zu können.

## 7.1 Analyse des ArangoDB-Datenmodells

Der Aufbau der datenbankseitigen Datenstruktur hängt zum einen stark vom Datenbank-System und dessen Möglichkeiten ab und zum anderen von den Abhängigkeiten zwischen den zu speichernden Datensätzen. Folgende Kapitel geben einen Einblick in die Modell-Eigenschaften der verwendeten Datenbank ArangoDB und den darauf basierten Entscheidungsgrundlagen über die Datenstruktur des zu entwickelnden Systems.

### 7.1.1 Datenbanken

Die oberste Struktureinheit im Datenbanksystem bildet die Einheit Datenbank. Sie beinhaltet mit dem Augenmerk auf den Datenkontext Collections und Graphen, auf welche in den folgenden Unterkapiteln genauer eingegangen wird. Um die vom Prototypen benötigten Daten zu organisieren, wurden zwei Möglichkeiten im Kontext der Datenbanken genauer betrachtet. Die erste Lösung beschreibt das Speichern aller oBPM-relevanten Daten in einer einzigen Datenbank. In der zweiten Lösung dagegen wird pro Prozess eine separate Datenbank angelegt, plus eine separate Datenbank für die Benutzer- und Authentifizierungsverwaltung. Für die zweite Lösung spricht eine geringere Datenkomplexität, da keine Verweise zwischen Prozessdaten und Prozessmodell nötig sind. Pro Datenbank gibt es nur ein zentrales Prozessmodell, dem alle Prozessdaten somit

automatisch zugeordnet sind. Des Weiteren wird in der aktuellen Arbeit davon ausgegangen, dass zwischen einzelnen Prozessen keine Daten ausgetauscht werden. Das separate Speichern einzelner Prozesse in Datenbanken vereinfacht deren Verwaltung. Soll beispielsweise ein einzelner Prozess gelöscht oder archiviert werden, kann dies über seine vollständige Datenbank geschehen, statt wie bei der ersten Lösung einzelne Datensätze in einer einzelnen zentralen Datenbank löschen, bzw. archivieren zu müssen.

Jede der Prozess-Datenbanken besitzt intern denselben Aufbau von Collections und Graphen. Beim Erstellen eines neuen Prozesses wird nicht nur dessen Datenbank vom System angelegt, sondern auch alle benötigten Collections und Graphen erstellt.

### 7.1.2 Dokumente und Collections

Wie von Arango GmbH (2016a) beschrieben handelt es sich beim verwendeten Datenbanksystem ArangoDB um eine NoSQL-Datenbank, welche die Datensätze, sogenannte Dokumente, in Collections organisiert. Dokumente besitzen einen eindeutigen Schlüssel und sind schemalos, das heisst, ihr Dateninhalt wird nicht von einem Schema vordefiniert und kann eine beliebige Struktur vorweisen (Arango GmbH, 2016d). Der Prototyp verwendet verschiedene Collections, welche thematisch zusammengehörende Dokumente verwalten. Von wenigen Ausnahmen abgesehen besitzen Dokumente derselben Collection im oBPM-Prototypen die gleiche Struktur. Der Aufbau der Dokumente aller verwendeter Collections kann in *7.2 Implementation* eingesehen werden.

#### 7.1.2.1 Edge-Collections

Edge-Collections stellen einen Spezialfall der Collections dar. Sie beinhalten Edge-Dokumente, welche Relationen zwischen Dokumenten anderer Collections, bzw. Edge-Collections darstellen (Arango GmbH, 2016j). Sie zeichnen sich dadurch aus, dass sie die System-Attribute *_from* und *_to* besitzen, welche die eindeutigen Schlüssel des Source- und Target-Dokuments beinhalten. Edge-Collections werden im oBPM-Prototypen für die Verweise von Modell-Dokumenten, bzw. Prozessinstanz-Dokumenten verwendet, wie in Kapitel *7.2.1 Dokument-Relationen* erläutert.

### 7.1.3 Graphen

Laut Arango GmbH (2016d) unterstützt ArangoDB die Strukturierung der Daten in Form von Graphen. Wie von Mapanga & Kadebu (2013, S. 14) erläutert, speichern

graphen-basierte Datenbanken Informationen in Knotenpunkten ab, welche über Verbindungslinien, sogenannte Edges, miteinander verbunden sind. Die Knotenpunkte stellen die eigentlichen Daten-Entities dar, welche wie auch die Edges beliebige Datenattribute besitzen können. Abbildung 7 zeigt eine schematische Darstellung eines Beispiels eines Graphen. Ersichtlich sind die Entitäten (Firma, Person) mit ihren jeweiligen Attributen. Des Weiteren sind die Edge-Verbindungen (ARBEITET_BEI, KUNDE_VON) aufgezeigt, welche zusätzliche Attribute als Information zu der jeweiligen Relation zwischen den Entitäten zur Verfügung stellen.

ArangoDB bietet laut der Dokumentation von Arango GmbH (2016e) verschiedene Operationen auf bestehende Graphen an, so zum Beispiel GRAPH-EDGES für das Abfragen aller Edges in einem Graphen, GRAPH_VERTICES für das Abfragen aller Entitäten in einem Graphen, COMMON_NEIGHBORS für das Finden gemeinsamer Nachbar-Entitäten in einem Graphen oder GRAPH_SHORTEST_PATH zum Auslesen des kürzesten Pfades zwischen zwei Entitäten.

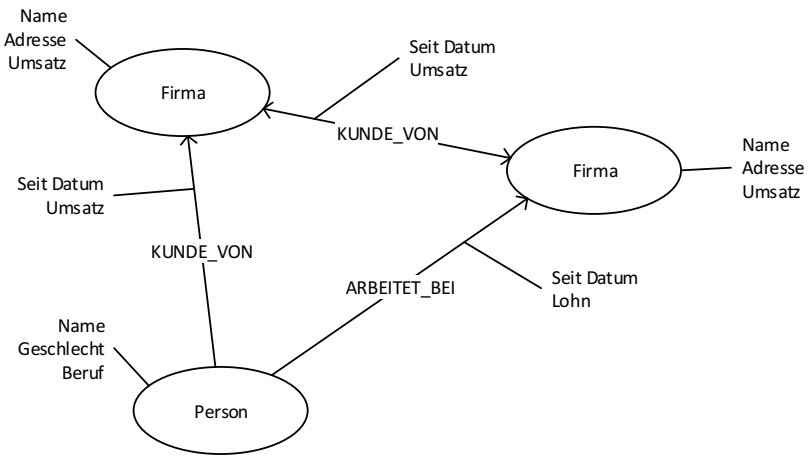


Abbildung 7: Beispiel eines Graphen

Im Zusammenhang zur Unterstützung der Graphen wurde auch ein Augenmerk auf die Performance gelegt. Dazu wurden Performance-Vergleiche zwischen Datenbank-Systemen beigezogen, welche sich auf Graphen-Operationen konzentrieren. Abbildung 8 zeigt einen von Arango GmbH (2016b) produzierten Vergleich mehrerer SQL- und NoSQL-Systemen. Dabei wurden in einer Datenbank mit 1’632'803 vernetzten Dokumenten Graphen-Operationen durchgeführt, um alle direkten Nachbarn aller Nachbar-Entitäten einer Ausgangs-Entität zu finden. Die in Abbildung 8 gezeigte Skala ist geeicht auf die von ArangoDB benötigte Zeit (100%). Interessant ist dabei vor allem

der direkte Vergleich mit den beiden Systemen Neo4j und OrientDB, da es sich bei beiden um reine Graphen-Datenbanken handelt, wie bereits von Miller (2013, S. 1) und OrientDB LTD (o. J.) beschrieben. Die Resultate zeigen, dass ArangoDB mit grossem Abstand die kürzeste Zeitspanne für die Berechnung braucht, im Vergleich zu Neo4j (511%) und OrientDB (1'119%).

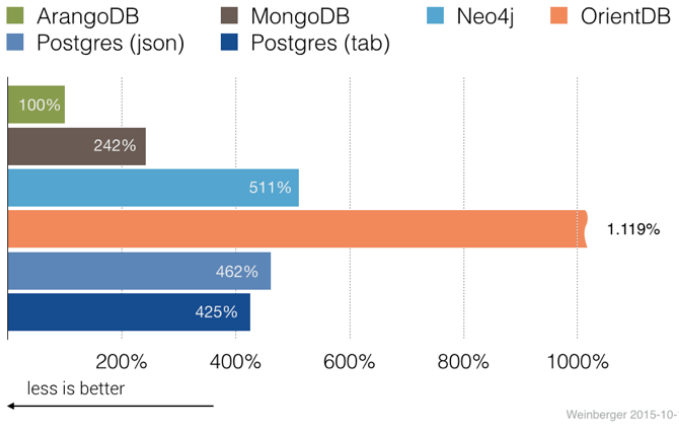


Abbildung 8: Performance-Vergleich von Graphen-Operationen (Arango GmbH, 2016b)

Die von Arango GmbH selbst produzierten Performance-Resultate wurden zur Sicherheit mit externen Performance-Tests abgeglichen, so beispielsweise mit den von Dohmen, Klamma, & Celler (2012, S. 30) produzierten Resultaten. Sie verglichen die Performance von ArangoDB und Neo4j anhand einer Shortest-Path-Operation gegen 500, 1000, 1500 und 2000 Nodes (Entitäten). Abbildung 9 zeigt das Resultat des Performance-Tests. Es zeigt sich hier, dass Neo4j mit Abstand besser abschneidet, vor allem bei ansteigender Anzahl Entitäten.

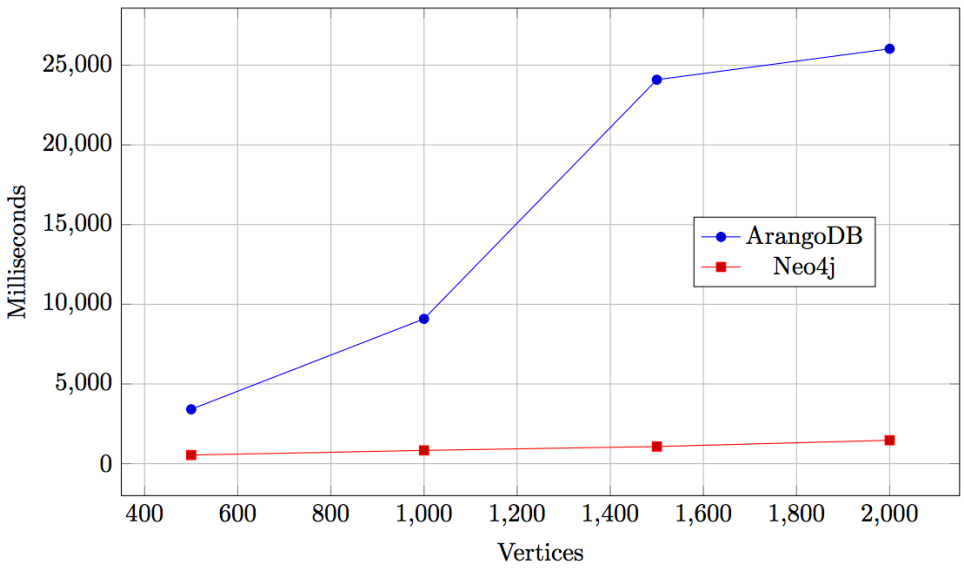


Abbildung 9 Shortest-Path Performance-Tests von Dohmen, Klamma & Celler (2012, S. 30)

## 7.2 Implementation

Dieses Unterkapitel beschreibt die tatsächliche Implementation des Datenmodells auf Basis der in Kapitel *7.1 Analyse des ArangoDB-Datenmodells* beschriebenen ArangoDB-Datenstruktur. Um den grundsätzlichen Aufbau der Daten erläutern zu können, werden als erstes verschiedene Implementationsvarianten von Dokument-Relationen vorgestellt und miteinander verglichen. Als zweites wird die Strukturierung aller relevanten Daten in Form von Schemas vorgestellt. Als letztes werden weitere Datenstrukturen beschrieben, welche nicht für die Persistierung von Daten, sondern für die Kommunikation zwischen Client und Server verwendet werden.

Zum einen muss das System fähig sein, Informationen zu einem Prozessmodell abbilden zu können, wie dies in Kapitel *2.1.1 Top-Down-Perspektive* und in Tabelle 1, Anforderung *001* beschrieben ist. Die benötigte Datenstruktur und Objekt-Schemas sind in Kapitel *7.2.2 Datenstruktur der Prozess-Modelle* erläutert. Zum anderen muss das System Daten, welche während der Laufzeit eines Prozesses anfallen, ablegen können (Tabelle 1, Anforderung *003.1*). Dazu gehören Prozess-Dokumente, wie auch die Aufzeichnung von Dokument-Änderungen (Tabelle 1, Anforderung 003.3). Auf diese Datenstruktur wird in Kapitel *7.2.4 Datenstruktur der Prozess-Daten* eingegangen. Schlussendlich müssen die vom User ausführbaren Prozess-Actions hinterlegt werden können, welche ein Teil vom Prozessmodell darstellen. Die Definition der Prozess-

Actions deckt dabei die in Kapitel *2.1.1 Top-Down-Perspektive* beschriebenen Use-Case- und State-Machine-Diagramme ab. Die Datenstruktur von Actions wird im Kapitel *7.2.3 Prozess-Actions* aufgezeigt.

Bei der Beschreibung aller verwendeten Datenstrukturen werden folgende Konventionen eingehalten:

- `type[]`: Die Klammern beschreiben ein Array des generischen Typs `type`.
- [Attribut-Name]: Die Umklammerung beschreibt ein optionales Attribut..

### 7.2.1 Dokument-Relationen

Sobald in einer Datenstruktur Verweise zwischen einzelnen Dokumenten nötig sind, müssen Relationen zwischen zwei oder mehreren Dokumenten abgebildet werden können. Dies können beispielsweise Child-Parent-Beziehungen in einem Dokumenten-Baum sein, wie sie im oBPM-Prototypen zur Abbildung der Prozessdaten und Modell-Datenstruktur verwendet werden. Um Relationen zwischen zwei oder mehreren Dokumenten abbilden zu können, wurden zwei mögliche Variationen genauer untersucht um eine Relation zwischen einem Source-Dokument und einem Target-Dokument herzustellen, mit dem Ziel, eine spätere Daten-Abfrage oder –Manipulation so einfach und performant wie möglich zu halten.

In der ersten Variation, welche fortführend als direkte Dokumentverweise benannt wird, wird im Source-Dokument als Attribut der einmalige Dokument-Schlüssel des Target-Dokuments hinterlegt. Bei einer Datenabfrage über mehrere Relationen dieses Typs wird dabei auf den FILTER-Operator, bzw. JOIN-Operator bei Collection-übergreifenden Relationen gesetzt (Arango GmbH, 2016g).

Die zweite Variation von möglichen Relationsmechanismen setzt hingegen auf den Graphen-Support von ArangoDB (Arango GmbH, 2016h). Dabei werden Verweise zwischen einzelnen Dokumenten über Edge-Dokumente dargestellt, wodurch es möglich wird, Abfragen statt über FILTER-und JOIN-Operatoren mit Hilfe den von ArangoDB zur Verfügung gestellten Graphen-Funktionen auszuführen.

Beim zu entwickelnden Prototypen wurde auf die zweite Variation gesetzt, um die Graphen-Funktionalität von ArangoDB vollständig auszunutzen. Des Weiteren verringert sich die Komplexität der einzelnen Source- und Target-Dokumente, da im Dokument selbst keine Verweise auf andere Entitäten notwendig sind. Zusätzlich zeigt sich auch,

dass sich die in Kapitel *7.2.2 Datenstruktur der Prozess-Modelle* und *7.2.4 Datenstruktur der Prozess-Daten* beschriebenen Dokumentstrukturen sehr einfach in Graphen abbilden und abfragen lassen. Dazu zeigt Abbildung 10 den Unterschied der Datenstruktur auf, wenn statt direkten Dokumentverweisen Edge-Collections als Dokument-Relationen verwendet werden. Es ist leicht ersichtlich, dass sich Datenabfragen über mehrere Edge-basierte Dokument-Relationen einfacher gestalten. In der ersten Version ist wie in Abbildung 10 ersichtlich eine Relation-Entity nötig, da in der vorliegenden Datenstruktur Informationen auf den jeweiligen Relationen hinterlegt werden können müssen.

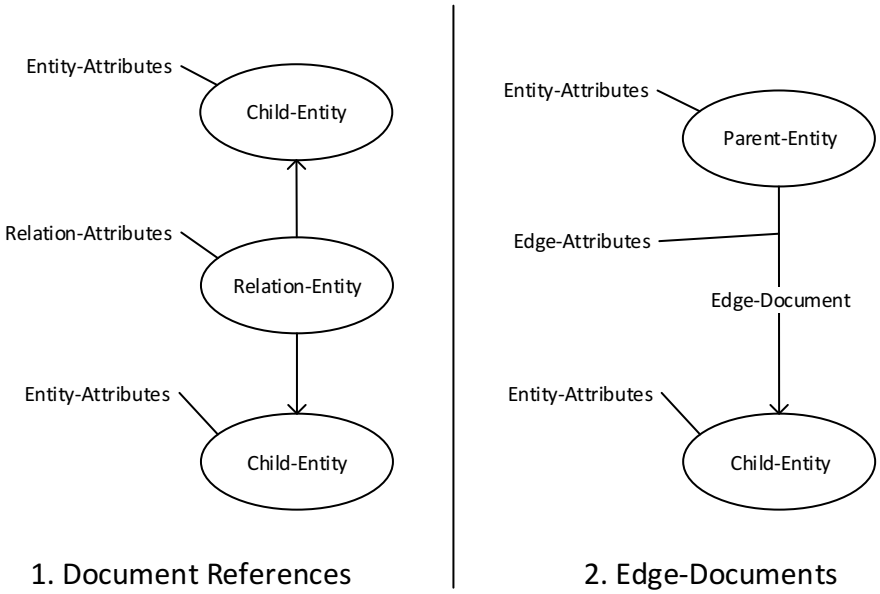


Abbildung 10: Vergleich von Dokument-Relationen

### 7.2.2 Datenstruktur der Prozess-Modelle

Dieses Kapitel zeigt die Datenstruktur und verwendeten Schemas auf, welche zur Abbildung aller Prozessmodell-relevanten Daten nötig sind. Ein Prozess-Modell besteht dabei aus einem Datenmodell, welches die Struktur der Prozessdaten vorgibt. Abbildung 11 zeigt ein Beispiel eines Datenmodells, wie es für ein Prozess-Modell hinterlegt werden könnte. Dabei gibt es zwei wichtige Anmerkungen:

1. Das Datenmodell definiert nur Abhängigkeiten und Kardinalitäten zwischen Entitäten. Attribute einzelner Entitäten werden stattdessen anhand der Daten-Schemas, welche den einzelnen Actions wie in Kapitel *7.2.3 Prozess-Actions* beschrieben hinterlegt werden, definiert.

2. Das Datenmodell besitzt jeweils ein Root-Objekt vom Entität-Typ Case. Dessen Existenz wird von der Business-Logik des Application-Servers sichergestellt.

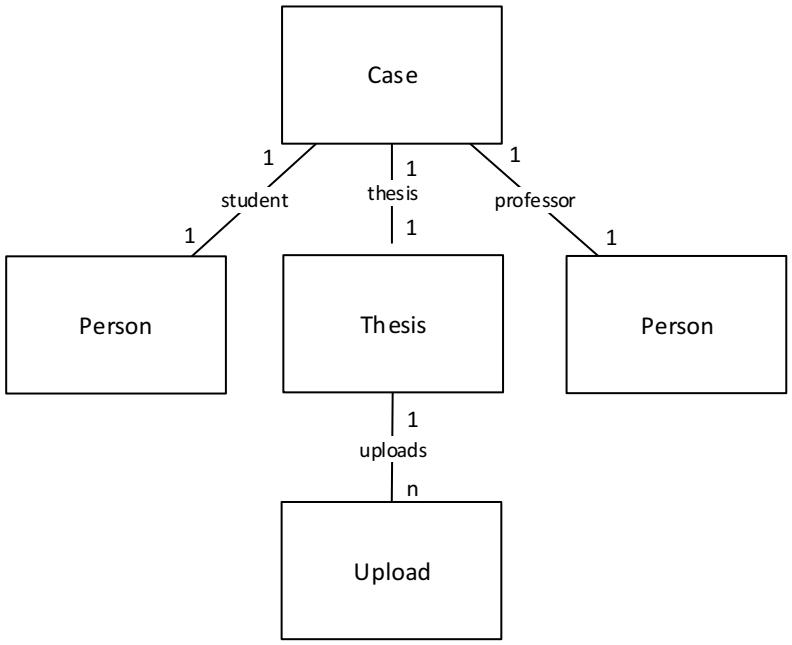


Abbildung 11: Beispiel eines Prozess-Datenmodells

Um ein wie in Abbildung 11 definiertes Schema in ArangoDB sichern zu können, sind die folgend vorgestellten Schemas nötig, welche in separaten Collections als JSON-Dokumente gespeichert werden. Die jeweilige Collection trägt dabei denselben Namen wie das jeweilige Schema.

#### 7.2.2.1 DocumentType

Das Schema *DocumentType* dient zur Beschreibung einer Entität in einem Datenmodell. Das einzige nötige Attribut des Schemas ist *type*, welches den Typ der Entität festlegt, wie Tabelle 27 aufzeigt.

Tabelle 27: Datenstruktur des Typs DocumentType

| Attributname | Datentyp | Beschreibung |
|---|---|---|
| type | `string` | Der Name des Dokument-Typs. |

#### 7.2.2.2 hasModel

Um die im Modell definierten Entitäten miteinander zu verknüpfen, benutzt der oBPM-Prototyp Edge-Dokumente, welche in der Edge-Collection *hasModel* hinterlegt

sind. Dieses Schema erlaubt es zum einen festzulegen, welche Entitätstypen verknüpft werden, zum anderen wird der Property-Name der Relation und die Kardinalität hinterlegt. In der Abbildung 11 werden die Edge-Dokumente des Typs *hasModel* als Verbindung zwischen den Entitäten dargestellt, mit ihrem jeweiligen Property-Namen und ihrer Kardinalität. Tabelle 28 listet alle verfügbaren Attribute des Schemas *hasModel* auf.

Tabelle 28: Datenstruktur des Typs hasModel

| **Attributname** | **Datentyp** | **Beschreibung** |
|---|---|---|
| _from | `DocumentID` | Arango-interner Schlüssel, welcher auf den Source-Dokumenttypen der Verknüpfung zeigt |
| _to | `DocumentID` | Arango-interner Schlüssel, welcher auf den Destination-Dokumenttypen der Verknüpfung zeigt. |
| [max] | `number` | Optionale maximal erlaubte Anzahl Verknüpfungen dieser Art. |
| property | `string` | Name der Property, unter welcher diese Verknüpfung angelegt wird. |

### 7.2.3 Prozess-Actions

Actions definieren die Tasks, welche User auf bestimmte Dokumente ausführen können, um deren Inhalt und Status zu ändern. Die Definitionen der Prozess-Actions ersetzen das von oBPM vorgesehene Use-Case- und State-Machine-Diagramm, welches in Kapitel *2.1.1 Top-Down-Perspektive* beschrieben wird. Eine Definition einer Prozess-Action beinhaltet dazu die Information, welche Benutzer und Benutzer-Rollen für die Ausführung berechtigt sind, welche Dokumente in welchem Zustand zur Ausführung benötigt werden und welche Statusänderung für welches Dokument nach erfolgreicher Ausführung vollzogen wird. Zusätzlich zu den Informationen der Use-Case und State-Machine-Diagrammen erlaubt eine Prozess-Action-Definition das Hinterlegen eines Daten-Schemas, welches Benutzern vorschreibt, welche zusätzlichen Daten pro Dokument für die Ausführung nötig sind. Diese vom Client mitgesendeten Daten werden in ein neu zu erstellendes oder bestehendes Dokument integriert.

Der Grund für die Verwendung von Prozess-Action-Definitionen statt Use-Case- und State-Machine-Diagrammen besteht in der Möglichkeit, alle relevanten Informationen

ohne Redundanz in einer einzelnen Collection ablegen zu können. Des Weiteren hat das Konzept von Prozess-Actions als Datenstruktur den Vorteil, dass bei der Ausführung einer Action oder dem Abfragen aller für den aktuellen Benutzer zur Verfügung stehenden Actions keine Datenaggregationen vollzogen werden müssen. Wären die relevanten Informationen für die genannten Abfragen auf mehrere Diagramme verteilt, erhöht dies die Komplexität der Business- und Abfragelogik beträchtlich.

Eine weitere Eigenschaft einer Action besteht im Kontext, unter welchem sie ausgeführt werden kann. Zum einen gibt es Actions, welche nicht im Kontext eines bestehenden Cases ausgeführt werden. Sie zeichnen sich durch das Attribut *createsNewCase* aus und erstellen bei der Ausführung immer einen neuen Case. Alle anderen Actions müssen im Kontext von einem bestehenden Case ausgeführt werden.

Tabelle 29 zeigt alle notwendigen Attribute einer Prozess-Action auf, um das in der Aufgabenstellung beschriebenen Verhalten gewährleisten zu können.

Tabelle 29: Datenstruktur des Typs Action

| **Attributname** | **Datentyp** | **Beschreibung** |
|---|---|---|
| createsNewCase | `boolean` | Wenn dieses Attribut auf `true` gesetzt wird, wird vor der Ausführung der Action ein neues Case-Dokument erstellt. |
| name | `String` | Ein nicht eindeutiger Name der Aktion. Wird als Anzeigename verwendet. |
| roles | `String[]` | Eine Liste von Benutzern oder Rollen, welche bemächtigt sind die Aktion auszuführen. Wie in Kapitel *8.2.1.2 Prozess-Action-Berechtigung* kann es sich auch um einen Objekt-Pfad handeln. |
| documents | `Object` | Ein Objekt, dessen Attribute alle mit der Aktion verknüpften Dokumente beschreibt. |

Dabei ist das Attribut *documents* speziell zu beachten. Der Typ des Attributs stellt ein dynamisches Objekt dar, welches eine beliebige Anzahl Attribute besitzen kann, jedoch muss der Wert jedes Attributes dem Typ *ActionDocument* entsprechen, dessen Schema in Tabelle 30 definiert ist.

Des Weiteren muss die Kombination der optionalen Attribute im Schema *ActionDocument* beachtet werden. Soll bei der Ausführung der Action, zu welchem das

*ActionDocument* gehört, ein neues Dokument anlegen, muss kein *state* definiert werden. Jedoch muss das Attribut *path* hinterlegt werden, welches definiert, unter welchem Objekt-Pfad das neue Dokument angelegt werden soll. Handelt es sich bei einem *ActionDocument* um eine Referenz zu einem bestehenden Dokument und wird der Status des Dokuments durch die Ausführung der Action nicht verändert, kann das Attribut *endState* weggelassen werden.

Das Attribut *schema* hinterlegt ein optionales Objekt, welches ein JSON-Schema definiert. Die Struktur des Schema-Objekts muss dem JSON-Schema-Standard entsprechen, wie von Galiegue & Zyp (2013) definiert.

Tabelle 30: Datenstruktur des Typs ActionDocument

| **Attributname** | **Datentyp** | **Beschreibung** |
|---|---|---|
| type | `String` | Der verlangte Typ des Dokuments. |
| [state] | `String` | Optionaler verlangter Status des Dokuments. Nicht notwendig wenn ein neues Dokument erstellt werden soll. |
| endState | `String` | Finaler Dokument-Status nach Ausführung der Action. |
| [path] | `String` | Optionaler Pfad unter welchem ein neues Dokument erstellt werden soll. |
| [schema] | `Object` | Ein optionales JSON-Schema mit welchem clientseitige Daten bei der Ausführung der Action validiert werden sollen. |

Code 3 zeigt eine Beispiel-Definition einer Action im JSON-Format, wie es auch in ArangoDB hinterlegt wird. Das Attribut *documents.file* beschreibt dabei ein *ActionDocument*, welches ein neues Dokument vom Typ *Upload* unter dem Objekt-Pfad `thesis.uploads` erstellt. Laut hinterlegtem Schema benötigt die Erstellung des Dokuments die zwei Attribute *content* und *type*, beide vom Typ `string`.

Nach erfolgreicher Erstellung des neuen Dokuments wird ihm der Status *created* zugewiesen, wie im Attribut *endState* definiert.

```
{
  "name": "createUpload",
  "roles": [
    "student"
  ],
  "documents": {
    "thesis": {
      "type": "Thesis",
      "state": "assigned"
    },
    "file": {
      "type": "Upload",
      "endState": "created",
      "path": "thesis.uploads",
      "schema": {
        "type": "object",
        "properties": {
          "content": {
            "type": "string"
          },
          "fileName": {
            "type": "string"
          }
        },
        "required": [
          "content",
          "fileName"
        ]
}}}}
```

Code 3: Beispiel-Definition einer Action

### 7.2.4 Datenstruktur der Prozess-Daten

Dieses Kapitel beschreibt alle Datenstrukturen, welche zur Persistierung von Prozess-Daten verwendet werden.

#### 7.2.4.1 Dokumente

Dokumente beschreiben Datenstrukturen, welche benötigte Informationen eines Prozess-Cases speichern. Sie können Daten in beliebigen Formaten enthalten und befinden sich immer in einem bestimmten Status. Des Weiteren sind sie über die Edge-Collection *hasDocument* untereinander verknüpft, um das im Prozess-Design beschriebene Datenmodell abzubilden. Abbildung 12 zeigt dazu eine Dokumenten-Struktur mit zwei Cases auf, welche dem in Abbildung 11 beschriebenen Datenmodell entspricht. Die Collection *Document* speichert dabei nur die Instanzen der Entitäten, jedoch nicht deren Verbindung zueinander. Diese sind in der Edge-Collection

*hasDocument* abgelegt, welche im Kapitel *7.2.4.2 Dokument-Verknüpfungen* beschrieben wird.

Um die Anforderung 003.2 vollständig zu erfüllen, muss die Datenstruktur erlauben, JSON-basierte wie auch binäre Dokumente ablegen zu können. Um binäre Dateien innerhalb von Prozess-Dokumenten speichern zu können, muss der Inhalt der Datei in einen Base64-codierten String umgewandelt werden. Dadurch kann der gesamte Inhalt der Datei in der JSON-basierten Daten-Struktur des Prozess-Dokuments hinterlegt werden.

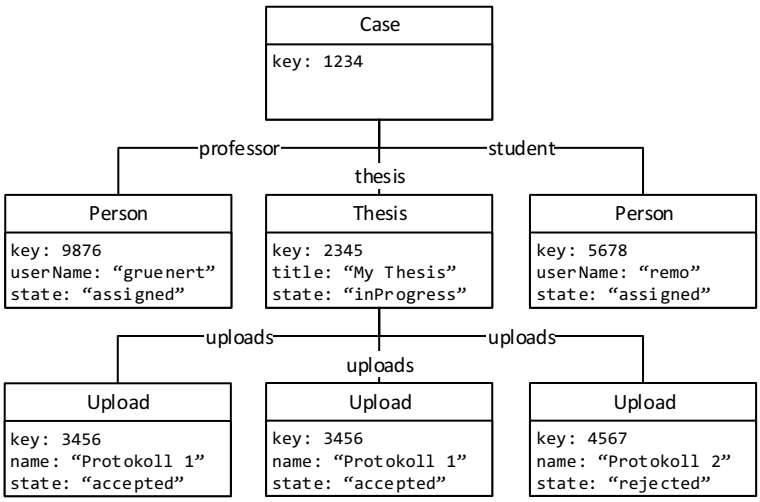


Abbildung 12: Beispiel-Datenstruktur eines Datenmodells

Die Tabelle 32 zeigt das Schema der Dokumente der Collection *Document* auf.

Tabelle 31: Datenstruktur des Typs Document

| Attributname | Datentyp | Beschreibung |
|---|---|---|
| type | `string` | Der Typ des Dokuments |
| state | `string` | Der aktuell hinterlegte Status des Dokuments. |
| data | `object` | Eine beliebige Datenstruktur, welche die zu speichernden Datenobjekte beinhaltet |

### 7.2.4.2 Dokument-Verknüpfungen

Anhand des definierten Datenmodells eines Prozesses wie in *7.2.2 Datenstruktur der Prozess-Modelle* definiert, müssen die in *7.2.4.1 Dokumente* beschriebenen Dokumente

miteinander verknüpft werden. Dazu dienen die Edge-Dokumente der Collection *hasDocument*, welche die Verbindung zwischen einem Parent- und einem Child-Dokument unter einer bestimmten Property darstellen. In Abbildung 12 werden die Edge-Dokumente der Edge-Collection *hasDocument* durch die Verbindungen zwischen den Entitäten dargestellt mit ihrem jeweiligen Property-Namen.

Tabelle 32: Datenstruktur des Typs hasDocument

| **Attributname** | **Datentyp** | **Beschreibung** |
|---|---|---|
| _from | `DocumentID` | Arango-interner Schlüssel, welcher auf das Source-Dokument der Verknüpfung zeigt |
| _to | `DocumentID` | Arango-interner Schlüssel, welcher auf das Destination-Dokument der Verknüpfung zeigt. |
| max | `number` | Optionale maximal erlaubte Anzahl Verknüpfungen dieser Art. |
| property | `string` | Name der Property, unter welcher diese Verknüpfung angelegt wird. |

#### 7.2.4.3 Dokument-Records

Dieses Kapitel erläutert das Schema von Dokument-Records. Um die Anforderung 003.3 zu erfüllen, muss das oBPM-System alle Änderungen eines Dokumentes während eines laufenden Prozesses erfassen. Dies wird im oBPM erfüllt, indem bei jeder Ausführung einer Prozess-Action automatisch ein Dokument-Record erstellt wird, welcher den Zeitpunkt, den ausführenden Benutzer und die Änderungen an den betroffenen Dokumenten festhält, inklusive den Datenmanipulationen und den Änderungen der Dokument-Status. Die Tabelle 33 zeigt das Schema eines Dokument-Records auf.

Tabelle 33: Datenstruktur des Typs Record

| Attributname | Datentyp | Beschreibung |
|---|---|---|
| date | `String` | JSON-formatierter Zeitstempel mit dem aktuellen Datum und der Zeit. |
| user | `Object` | User-Objekt welches den User-Namen und User-ID beinhaltet. |
| action | `Object` | Beinhaltet den Namen und die ID der ausgeführten Action. |
| document | `Object` | Beinhaltet den Typ des Dokuments, die Status-änderung und die veränderten Dokument-Daten. |

### 7.2.5 View-Models

Dieses Kapitel erläutert Datenstrukturen, welche nicht zur Persistierung von Daten in ArangoDB verwendet werden, sondern zur Kommunikation zwischen Clients und Application-Server. Für die Ausführung unterschiedlicher API-Schnittstellen in Kapitel *6.3 API-Schnittstellen* beschrieben sind spezifische Datenstrukturen notwendig, welche in den folgenden Unterkapiteln erläutert werden.

#### 7.2.5.1 NewUser View-Model

Die in diesem Kapitel beschriebene Datenstruktur ist notwendig für das Erstellen eines neuen Benutzers über die API-Schnittstelle *6.3.3.1 Benutzer erstellen*. Tabelle 34 zeigt das Schema des View-Models auf.

Tabelle 34: Datenstruktur des View-Models NewUser

| Attributname | Datentyp | Beschreibung |
|---|---|---|
| userName | `String` | Der Benutzername des zu erstellenden Benutzers. |
| password | `String` | Das clear-text Passwort des zu erstellenden Benutzers. |
| firstName | `String` | Der Vorname des zu erstellenden Benutzers. |
| lastName | `String` | Der Nachname des zu erstellenden Benutzers. |
| email | `String` | Die E-Mail-Adresse des zu erstellenden Benutzers. |
| roles | `String[]` | Eine Liste der Benutzer-Rollen des zu erstellenden Benutzers. |

#### 7.2.5.2 UpdateUser View-Model

Das View-Model *UpdateUser* wird für das Editieren eines bestehenden Benutzers über die API-Schnittstelle *6.3.3.2 Benutzer editieren* verwendet. Tabelle 35 zeigt das Schema des View-Models *UpdateUser* auf.

Tabelle 35: Datenstruktur des View-Models UpdateUser

| Attributname | Datentyp | Beschreibung |
|---|---|---|
| email | `String` | Die E-Mail-Adresse des zu erstellenden Benutzers. |
| firstName | `String` | Der Vorname des zu erstellenden Benutzers. |
| lastName | `String` | Der Nachname des zu erstellenden Benutzers. |
| roles | `String[]` | Eine Liste der Benutzer-Rollen des zu erstellenden Benutzers. |

#### 7.2.5.3 UpdatePasswort View-Model

Dieses hier beschriebene View-Model wird benötigt zum Ändern des Passworts eines Benutzers über die API-Schnittstelle *6.3.3.4 Benutzer-Passwort ändern*. Tabelle 36 zeigt das Schema des View-Models auf.

Tabelle 36: Datenstruktur des View-Models UpdatePasswort

| Attributname | Datentyp | Beschreibung |
|---|---|---|
| email | `String` | Die E-Mail-Adresse des zu erstellenden Benutzers. |
| firstName | `String` | Der Vorname des zu erstellenden Benutzers. |
| lastName | `String` | Der Nachname des zu erstellenden Benutzers. |
| roles | `String[]` | Eine Liste der Benutzer-Rollen des zu erstellenden Benutzers. |

#### 7.2.5.4 ModelDocument View-Model

Das *ModelDocument* View-Model wird für die Erstellung und Bearbeitung von Entitäten eines Prozess-Models über die API-Schnittstelle *6.3.4.1 Datentyp erstellen* und

*6.3.4.2 Datentyp editieren* verwendet. Tabelle 37 beschreibt das Schema des View-Models.

Tabelle 37: Datenstruktur des View-Models ModelDocument

| Attributname | Datentyp | Beschreibung |
|---|---|---|
| type | `String` | Der Typ der Prozessmodell-Entität |
| [parent] | `DocumentID` | Der Key der Parent-Entität. Wird nicht benötigt, wenn der Typ der Entität *Case* ist. |
| [max] | `number` | Optionale Kardinalität. Beschreibt die maximale Anzahl Verbindungen zwischen der Parent-Entität und der aktuellen Entität. |
| [property] | `String` | Der Property-Name der Verbindung zur Parent-Entität. Wird nicht benötigt, wenn der Typ der Entität *Case* ist. |

#### 7.2.5.5 Execution View-Model

Das *Execution* View-Model stellt ein zentrales View-Model des oBPM-Systems dar. Es beinhaltet die Information zur Ausführung einer Prozess-Action und wird zum Aufruf der API-Schnittstelle *6.3.6.2 Action ausführen* benötigt. Das View-Model muss zum einen die Information über die auszuführende Prozess-Action beinhalten, wie auch alle benötigten Informationen zu den einzelnen erwarteten Dokumenten einer Prozess-Action. Tabelle 38 definiert das Schema eines *Execution*-Models. Das Attribut *caseId* beschreibt dabei die Case-ID des Cases, unter dessen Kontext die Action ausgeführt werden soll. Diese Angabe ist nötig, da zum einen Berechtigungen zur Ausführung einer Action wie in Kapitel *8.2.1.2 Prozess-Action-Berechtigung* beschrieben anhand eines Objektpfades im Kontext eines Cases abgeleitet werden müssen. Zum anderen muss bei einer Ausführung einer Action, welche neue Dokumente für einen bestehenden Case generieren soll, mitgeteilt werden, zu welchem Case das neue Dokument hinzugefügt werden soll. Erstellt die Action jedoch einen neuen Case, ist dieses Attribut nicht gestattet. Das Attribut *documents* beschreibt ein dynamisches Objekt, welches die gleichen Attribute besitzen muss wie das von der Prozess-Action hinterlegte *documents*-Attribut. Der Wert jedes Attributs muss dabei dem Schema *ExecutionDocument* von Tabelle 39 entsprechen.

Tabelle 38: Datenstruktur des View-Models Execution

| Attributname | Datentyp | Beschreibung |
|---|---|---|
| actionId | `DocumentID` | Die ID der auszuführenden Action. |
| [caseId] | `DocumentID` | Case-ID, unter welchem die Action ausgeführt werden soll. Wird bei der Ausführung der Action ein neuer Case erstellt, ist dieses Attribut nicht gestattet. |
| documents | `Object` | Ein Objekt, dessen Attribute alle mit der Action verknüpfte Dokument-Daten beschreibt. |

Tabelle 39: Datenstruktur des View-Models ExecutionDocument

| Attributname | Datentyp | Beschreibung |
|---|---|---|
| [id] | `DocumentID` | Muss angegeben werden, wenn sich die Daten auf ein bestehendes Dokument beziehen. |
| [data] | `Object` | Optionale Daten, welche für das verknüpfte Dokument gesendet werde sollen. Müssen dem in der Action hinterlegten Schema entsprechen. |

# 8 Benutzer-Authentifizierung und –Autorisierung

Um die Sicherheit des Systems gewährleisten zu können und die Anforderung 002.2 und Anforderung 004.2 zu erfüllen, muss die API des Prototypen ein Security-Layer bereitstellen welches fähig ist, Anfragen von aussen bestimmten Benutzer zuordnen zu können (Identifikation), die Identität eines Benutzers überprüfen zu können (Authentifizierung) und anhand bestimmter Regeln Benutzeraktionen zuzulassen oder zu verweigern (Autorisierung). Mit Hilfe dieser drei Funktionalitäten ist das System fähig, unerlaubte Zugriffe von unbekannten Usern zu blockieren, bestimmte Bereiche der Applikation nur vordefinierten Usern zu Verfügung zu stellen und schlussendlich vollzogene Aktionen in der Vergangenheit eindeutig einem User zuordnen zu können. Folgende Unterkapitel gehen auf die einzelnen Bereiche der umgesetzten API-Security ein.

## 8.1 Authentifizierung

Da das vom System zur Kommunikation verwendete HTTP-Protokoll wie bereits von Fielding u. a. (2006) beschrieben statuslos ist, schränken sich die Möglichkeiten zur Authentifizierung von Usern grundsätzlich ein. In einer statuslosen Kommunikation werden einzelne Anfragen vom Selben oder mehreren Usern unabhängig voneinander verarbeitet. Kann also eine einzelne Anfrage einem bestimmten User dank mitgesendeter Authentifizierungsinformationen, wie beispielsweise Benutzername und Passwort, zugeordnet werden, hat dies keinen Einfluss auf zukünftige Anfragen desselben Users. Somit ergeben sich grundsätzlich zwei Möglichkeiten. Entweder werden die benötigten Authentifizierungsinformationen mit jeder einzelner API-Anfrage mitgesendet oder die Authentifizierung findet einmalig statt und der User identifiziert sich in zukünftigen Anfragen anhand eines eindeutigen Schlüssels, bzw. Tokens. Die erste Variante wird beispielsweise vom HTTP-eigenen Authentifizierungsmechanismus verwendet (Fielding & Reschke, 2014) und wird in dieser Arbeit nicht zur Anwendung kommen. Die zweite Variante findet hingegen in verschiedenen Authentifizierungsmechanismen ihre Anwendung, so zum Beispiel in einer Session-basierten oder OAuth-basierten Authentifizierung, bzw. Identifizierung. Der grundlegende Ablauf ist dabei immer derselbe: In einer ersten Anfrage authentifiziert sich der User mit den benötigten Informationen beim System und erhält bei Erfolg einen eindeutigen Token, welcher auch auf dem Server-System hinterlegt wird. In allen nachfolgenden Anfragen wird vom Client nur der Token mitgesendet, welches dem Server-System, bzw. API erlaubt den User eindeutig zu identifizieren.

In dieser Arbeit wird vollständig auf eine Authentifizierung basierend auf dem OAuth-Standard der Version 2.0 gesetzt. Dies, da OAuth zum einen eine sehr breite Anwendung im Bereich API-Authentifizierung findet (vgl. (Jacobson, Woods, & Brail, 2011, S. 78)). Weiter beschränkt sich die Anwendung von OAuth im entwickelten Prototypen auf eine simple Benutzername-Passwort-basierte Authentifizierung, jedoch kann dies in der Zukunft ausgebaut werden, inklusive Client- oder Thrid-Party-Authentifizierung. Für die gesamte Umsetzung der Authentifizierung wurde dabei auf das Node-Modul *oauth2-server*[1] gesetzt. Abbildung 13 stellt die Integration des Moduls in das bestehende System dar. Die Schnittstelle zwischen System, bzw. Datenbank und dem *oauth2-server*-Modul

[1] Zugriff auf die aktuelle Version unter: https://www.npmjs.com/package/oauth2-server

stellt dabei das Authentication-Model dar, welches nach den Anforderungen des externen Moduls aufgebaut ist und die benötigten Schnittstelen-Funktionen bereitstellt. Das *oauth2-server*-Modul wiederum bietet zwei Schnittstellen an, jeweils für die Authentifizierung und die Identifizierung eines Users. Diese werden über eine *express*-Route registriert und bei einer User-Anfrage ausgeführt.

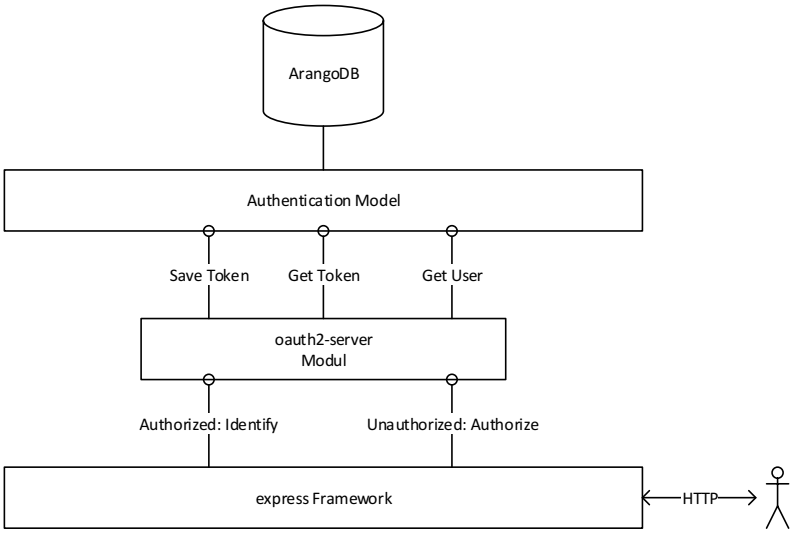


Abbildung 13: oauth2-server Integration

## 8.2 Autorisierung

Um bestimmte Funktionalitäten eines Systems nur bestimmten Usern zur Verfügung zu stellen können, ist ein Autorisierungssystem von Nöten. Dabei gibt es in der Praxis verschiedene Ansätze. Der vorliegende oBPM-Prototyp setzt dabei auf ein simples Rollen-System, wobei jeder registrierte User in einer oder mehreren Benutzerrollen eingegliedert ist. Berechtigungen können nun für einzelne User oder Rollen vergeben werden.

Im Prototypen können Berechtigungen auch zwei verschiedenen Ebenen vergeben werden. Zum einen lassen sich Zugriffe auf bestimmte REST-Schnittstellen einschränken. So können zum Beispiel die Schnittstellen zur Erstellung, Änderung und Entfernung von Prozess-Elementen, wie Actions und Dokument-Modelle, nur Usern der Rolle *modeler* zugänglich gemacht werden. Dazu werden die zu berechtigenden User-Rollen direkt in den Controllern, bzw. Controller-Actions hinterlegt. Findet ein unerlaubter Zugriff auf eine REST-Schnittstelle statt, wird die dazugehörige Controller-Action nicht ausgeführt. Stattdessen wird ein HTTP-Fehler mit dem Status *401 – Not Authorized* an den Client gesendet. Die hinterlegten Rollen in den Controllern und Controller-Actions bestehen aus

einem Set vordefinierter Rollen, welche zur Laufzeit auch nicht geändert werden können. Sie sind somit statisch und unabhängig vom aktuellen Prozess-Modell.

Zum anderen benötigt der Modellierer eines Prozess-Modells die Möglichkeit, den Zugriff auf einzelne Prozess-Actions einzuschränken. Die zugriffsberechtigten User und User-Rollen sind in den einzelnen Prozess-Actions hinterlegt und werden vor der eigentlichen Ausführung einer Action mit den aktuellen Rollen-Mitgliedschaften des aktuellen Benutzers verglichen. Befindet sich der User in keiner der hinterlegten Rollen der auszuführenden Prozess-Action, wird die Anfrage abgebrochen und mit einem HTTP-Fehler *401 – Not Authorized* beantwortet.

### 8.2.1 Implementation

Dieses Kapitel zeigt die technische Implementation der Berechtigungsverwaltung auf Schnittstellen-Ebene und auf Prozess-Action-Ebene auf. Dazu wird mit Hilfe von Code-Ausschnitten erläutert, wie die Anforderungen im System implementiert wurden.

#### 8.2.1.1 Schnittstellen-Berechtigung

Rollen-Berechtigungen auf REST-Schnittstellen werden wie in Kapitel *8.2 Autorisierung* erläutert direkt während der Entwicklung auf einzelnen Controllern, bzw. Controller-Actions hinterlegt. Dazu wird auf dem Controller oder der Action mit Hilfe des Typescript-Decorators `ControllerAuthorization`, bzw. `ActionAuthorization` die zu berechtigenden Rollen als Array hinterlegt, wie Code 4 zeigt. Dabei gelten folgende Regeln:

- Berechtigungen auf Controllern werden auf ihre Controller-Actions vererbt.
- Controller-Actions können vererbte Berechtigungen überschreiben.
- Bei vererbten Controllern werden die Berechtigungen vom Parent-Controller nicht vererbt.
- Eine leere Liste von Rollen erlaubt allen User-Rollen den Zugriff.

Auflistung 4: Berechtigungsregeln für Controllers und Actions

Anhand der Regeln in Auflistung 4 ergeben sich für die in Code 4 definierten Actions folgende Berechtigungen:

- Die Action `executables` überschreibt die vom Controller vererbte Berechtigung mit einer leeren Liste. Somit können alle User-Rollen die Action `executables` ausführen.

- Die Action `get` besitzt keine eigene Berechtigung und erbt somit die Berechtigungsdefinition des Controllers. Somit kann nur die User-Rolle *modeler* die Action `get` ausführen.

Auflistung 5: Ableitung der Berechtigungen anhand der definierten Berechtigungsregeln

```
@CtrlAuth(['modeler'])
export default class ActionController extends
    RepositoryController<ActionRespository, Action> {

    ...

    @ActionAuth([])
    public executables($user): q.Promise<any> {
        return this.repo.getExecutableActions($user);
    }

    public get($id): q.Promise<any> {
        return super.get($id);
    }

    ...
```

Code 4: Benutzung der Autorisierungs-Decorators

#### 8.2.1.2 Prozess-Action-Berechtigung

Die Berechtigung für einzelne Benutzer-Rollen oder Benutzer für die Ausführung einer Prozess-Action wird in der Definition der Action hinterlegt, und zwar im Attribut *roles*, wie in Tabelle 29 definiert. Dabei handelt es sich um ein Array, wodurch das Attribut mehrere Werte besitzen kann. Ein User muss dabei nur Mitglied einer hinterlegten Rolle sein, um zur Ausführung berechtigt zu werden. Statt den Namen einer Rolle in der Action zu hinterlegen, kann stattdessen auch ein Objekt-Pfad angegeben werden, welcher im Kontext des aktuellen Case-Objekts auf ein Attribut zeigt, welches einen zu berechtigenden Benutzer-Namen beinhaltet. Um diese Funktionalität und ihren Zweck genauer zu erläutern, wird folgend ein Beispiel einer Berechtigung per Objekt-Pfad aufgezeigt.

##### 8.2.1.2.1 Beispiel einer Berechtigung mit Hilfe eines Objekt-Pfads

Dieses Kapitel erläutert ein Beispiel für die Verwendung von Objekt-Pfaden zur Berechtigung von Benutzern zur Ausführung einer Prozess-Aktion. Als Daten-Grundlage für das Beispiel dienen die in Abbildung 14 definierten Datenstrukturen eines fiktiven Prozesses. Dabei handelt es sich um zwei separate Datenstrukturen, jeweils einem Case

zugeordnet. Jeder Case besitzt dabei ein Attribut *thesis* vom Typ *Thesis* und ein Attribut *student* vom Typ *Person*. Des Weiteren sind jeder Thesis zwei Dokumente des Typs *Upload* unter dem Attribut *uploads* zugeordnet.

Weiter geht das aktuelle Beispiel davon aus, dass im fiktiven Prozessmodell eine Prozess-Aktion *createUpload* existiert, welche einer bestehenden Thesis ein weiteres Dokument des Typs *Upload* unter dem Attribut *uploads* hinzufügt. Diese Action soll einem Studenten erlauben, einen neuen Upload zu seiner Thesis hinzuzufügen.

Wie Abbildung 14 beschreibt, ist dem ersten Case eine Person mit dem Benutzernamen *remo* (Remo) zugewiesen, dem zweiten Case die Person mit dem Benutzernamen *hans* (Hans). Von beiden Benutzern wird ausgegangen, dass sie der Benutzer-Rolle *student* zugewiesen sind.

Die eigentliche Problematik besteht nun darin, dass das Prozess-Modell die Möglichkeit bieten muss, zu verhindern, dass Hans neue Uploads zur Thesis von Remo hinzufügt und umgekehrt. Zwar kann die Action *createUpload* nur für Benutzer der Rolle *student* freigegeben werden, jedoch hindert dies einen Benutzer nicht, die Action im Kontext eines beliebigen Cases auszuführen.

Um die zuvor beschriebene Problematik zu umgehen, wird statt der Benutzer-Rolle *student* der zu berechtigende Benutzer im Kontext des aktuellen Cases in Form eines Objekt-Pfades hinterlegt. Dieser Objekt-Pfad wird während der Ausführung einer Action aufgelöst und muss als Wert den Benutzernamen des zu berechtigenden Benutzers zurückgeben. In Anlehnung an das Beispiel in Abbildung 14 würde der Objekt-Pfad `student.userName` lauten. Wird die Action *createUpload* vom Benutzer Remo im Kontext des Cases *2345* ausgeführt, gibt der Objektpfad `student.userName` den Wert *hans* zurück. Wird der genannte Objektpfad im Attribut *roles* der Action *createUpload* als einzige Berechtigung hinterlegt, wird dem Benutzer Remo die Ausführung der Action verwehrt. Er kann jedoch die Action im Kontext des Cases 1234 ausführen, da in diesem Fall der Objektpfad *remo* zurückgibt.

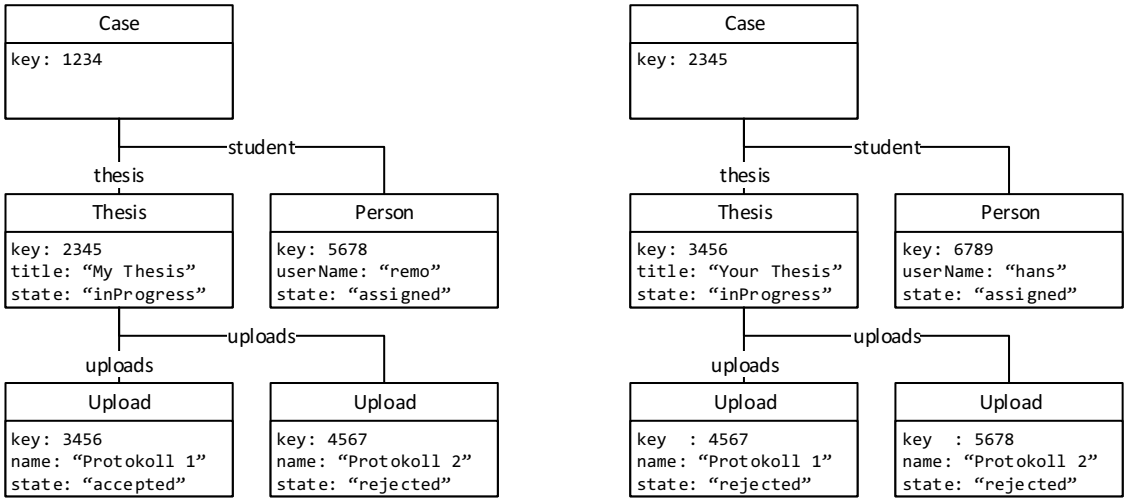


Abbildung 14: Beispiel mehrerer Prozess-Dokument-Strukturen

Code 5 zeigt die Definition der fiktiven Action *createUpload* auf. Dabei ist zu bemerken, dass Objekt-Pfade unter dem Attribut *roles* mit dem Charakter % beginnen müssen, um zu kennzeichnen, dass es sich beim Wert um einen Objekt-Pfad handelt.

```
{
  "name": "createUpload",
  "roles": ["%student.userName"],
  "documents": {
    "thesis": {
      "type": "Thesis",
      "state": "inProgress"
    },
    "file": {
      "type": "Upload",
      "endState": "created",
      "path": "thesis.uploads",
      ...
}}}}
```

Code 5: Definition einer Action mit Objekt-Pfad-Berechtigung

# 9 Testing

Dieses Kapitel erläutert die Test-Umgebung und die Durchführung verschiedener-Test-Szenarios für das vorliegende oBPM-System. Zum einen wurden verschiedene Benutzungs-Szenarios in API-Tests abgebildet, welche die API-Schnittstellen und die

Business-Logik automatisiert auf ihre Funktionalität überprüfen. Die Beschreibung der API-Tests wird in Kapitel *9.1 API-Tests* erläutert.

Um zu testen, wie sich das System unter hoher Last betreffend der Anzahl redundanter Client-Requests und der Grösse der zu verwaltenden Datenmengen verhält, wurden des Weiteren Performance-Tests durchgeführt. Die Performance-Tests decken sich zusätzlich mit der Anforderung 005.1, welche ein Benchmarking des oBPM-Systems voraussetzt.

In einem zusätzlichen Kapitel wird auf Verbesserungsmöglichkeiten eingegangen, um allfälligen Performance-Problemen zu begegnen.

## 9.1 API-Tests

Um die Funktionalität der Business-Logik und der API-Schnittstellen des oBPM-Systems automatisiert auf ihre Funktionalität testen zu können, wurde ein Test-Szenario erstellt, welches gegen ein laufendes oBPM-System abgespielt wurde. Die einzelnen Schritte des Test-Szenarios können im Anhang unter Kapitel *15.3 API-Testszenarios* eingesehen werden.

Durch die Implementation der API-Tests kann zudem nicht nur die Funktionalität des Systems bewiesen werden, sondern es vereinfacht auch die zukünftige Weiterentwicklung des Systems. So kann nach Änderungen im Code des Prototypen mit Hilfe eines Testlaufs festgestellt werden, ob die Änderungen Konsequenzen auf die Funktionalität des Systems haben. Alle definierten API-Tests konnten dabei erfolgreich durchgeführt werden.

## 9.2 Performance-Tests

Um das umgesetzte oBPM-System unter hoher Last zu testen, wurde wie in Anforderung 005.1 gefordert ein Benchmarking in Form von Performance-Tests durchgeführt. Um eine hohe Auslastung des Systems zu simulieren, wurde zum einen eine ArangoDB-Datenbankinstanz verwendet, welche bereits eine grosse Anzahl prozess-relevanten Datensätze persistiert hat. Zum anderen wurde mit Hilfe des von Fernández (2016) entwickelten Load-Testing-Tools loadtest eine hohe Anzahl redundanter Client-Requests auf verschiedene API-Schnittstellen ausgeführt. Als Kennzahlen zur Messung wurde dabei auf die benötigte Dauer der Verarbeitung einzelner Anfragen gesetzt. Als Laufzeitumgebung wurde bei den Performance-Tests auf den von keymetrics Inc. (2015) entwickelten Prozess-Manager pm2 gesetzt.

### 9.2.1 Zielsetzung

Das Ziel des Performance-Testing besteht darin, die Performance und Skalierbarkeit des Systems zu prüfen. Um eine hohe Skalierbarkeit zu erreichen, sollte sich die Reaktionszeit des Systems maximal linear proportional zu der Anzahl Case-Strukturen und gleichzeitiger Requests verhalten. Die Performance dagegen wird in dieser Testdurchführung rein an der Reaktionszeit des Systems gemessen. Dazu wird der Richtwert auf maximal 2 Sekunden gesetzt, welcher vom System nicht überschritten werden sollte.

### 9.2.2 Umgebung

Als Testumgebung diente das Entwicklungsgerät des Autors. Das Gerät besitzt einen Intel i7 2.2 GHz Prozessor mit 4 Kernen. Die maximale Memory-Auslastung liegt bei 16 GB RAM. Sowohl die Testläufe als auch das oBPM-System wurden parallel auf demselben Gerät betrieben.

### 9.2.3 Testkonfiguration und Vorgehen

Um die Performance des oBPM-Systems zu testen, wurden zwei zentrale API-Schnittstellen getestet. Zum einen wurde die Schnittstelle *6.3.6.1 Abfragen von ausführbaren Actions* ausgeführt, welche eine Liste von ausführbaren Actions im Kontext des aktuellen Benutzers zurückgibt. Zum anderen wurde die Schnittstelle *6.3.6.2 Action ausführen* getestet, in dem eine zuvor definierte Action ausgeführt wurde, welche zum einen ein bestehendes Dokument erwartet und ein neues Dokument erstellt.

Alle Tests wurden mit 50 virtuellen Clients ausgeführt, welche parallel während 120 Sekunden wiederholend Anfragen an die Schnittstellen sendeten. Dabei wurden pro Schnittstelle drei Test-Szenarios definiert, welche von einer unterschiedlichen Anzahl offener Cases ausgegangen sind. Im ersten Szenario wurden die Tests auf eine Prozess-Datenbank ausgeführt, welche 10 offene Case-Strukturen persistiert hat. Im zweiten Szenario wurde die Anzahl Cases auf 50 erhöht, um sie dann im dritten Test-Szenario auf 250 zu erhöhen. Alle verwendeten Case-Strukturen weisen dabei dieselbe Struktur auf, welche Abbildung 15 illustriert. Dabei muss betont werden, dass die beiden Upload-Dokumente PDF-Dateien beinhalten, welche eine Originalgrösse von 40KB und 41KB besitzen.

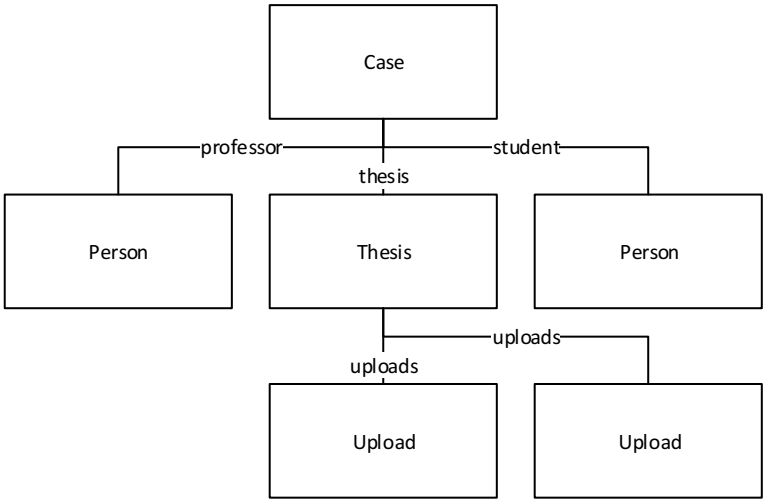


Abbildung 15: Verwendete Case-Struktur während den performance-Tests

### 9.2.4 Test-Resultate

Dieses Kapitel erläutert die Testresultate der Performance-Tests. Die Resultate sind unterteilt nach der getesteten API-Schnittstelle und den durchgeführten Szenarios pro Schnittstelle.

#### 9.2.4.1 Abfrage von ausführbaren Actions

Dieses Kapitel erläutert die Testresultate welche durch das Testen der Schnittstelle *6.3.6.1 Abfragen von ausführbaren Actions* gewonnen wurden. Tabelle 40, Tabelle 41 und Tabelle 42 zeigen die Resultate der durchgeführten Testszenarios.

Tabelle 40: Test-Resultat für ausführbare Actions, 10 Case-Strukturen

| | |
|---|---|
| Anzahl Clients | 50 |
| Testdauer | 120 Sekunden |
| Anzahl ausgeführter Requests | 932 |
| Anzahl Case-Strukturen | 10 |
| **Median der Reaktionszeit:** | **4.7 Sekunden** |

Tabelle 41: Test-Resultat für ausführbare Actions, 50 Case-Strukturen

| | |
|---|---|
| Anzahl Clients | 50 |
| Testdauer | 120 Sekunden |
| Anzahl ausgeführter Requests | 200 |
| Anzahl Case-Strukturen | 50 |
| **Median der Reaktionszeit:** | **13.0 Sekunden** |

Tabelle 42: Test-Resultat für ausführbare Actions, 250 Case-Strukturen

| | |
|---|---|
| Anzahl Clients | 50 |
| Testdauer | 120 Sekunden |
| Anzahl ausgeführter Requests | 15 |
| Anzahl Case-Strukturen | 250 |
| **Median der Reaktionszeit:** | **75.3 Sekunden** |

#### 9.2.4.2 Ausführen einer Action

Dieses Kapitel beschreibt die Testresultate, welche durch das Testen der Schnittstelle *6.3.6.2 Action ausführen* entstanden sind. Tabelle 43, Tabelle 44 und Tabelle 45 zeigen die Resultate der durchgeführten Testszenarios.

Tabelle 43: Test-Resultat für Ausführen einer Action, 10 Case-Strukturen

| | |
|---|---|
| Anzahl Clients | 50 |
| Testdauer | 120 Sekunden |
| Anzahl ausgeführter Requests | 450 |
| Anzahl Case-Strukturen | 10 |
| **Median der Reaktionszeit:** | **12.0 Sekunden** |

Tabelle 44: Test-Resultat für Ausführen einer Action, 50 Case-Strukturen

| | |
|---|---|
| Anzahl Clients | 50 |
| Testdauer | 120 Sekunden |
| Anzahl ausgeführter Requests | 450 |
| Anzahl Case-Strukturen | 50 |
| **Median der Reaktionszeit:** | **11.5 Sekunden** |

Tabelle 45: Test-Resultat für Ausführen einer Action, 250 Case-Strukturen

| | |
|---|---|
| Anzahl Clients | 50 |
| Testdauer | 120 Sekunden |
| Anzahl ausgeführter Requests | 15 |
| Anzahl Case-Strukturen | 250 |
| **Median der Reaktionszeit:** | **18.2 Sekunden** |

### 9.2.5 Auswertung der Resultate

Die Auswertung der Resultate vergleicht die erhaltenen Testwerte von *Kapitel 9.2.4 Test-Resultate* mit der definierten Zielsetzung von *Kapitel 9.2.1 Zielsetzung*. Dabei zeigt sich, dass in allen Test-Szenarios der Richtwert von 2 Sekunden für die maximale Reaktionszeit des Systems massiv überschritten wurden. Was die Skalierbarkeit anbelangt, konnte festgestellt werden, dass sich die Schnittstelle zur Ausführung von Actions sehr gut skalieren lässt. Obwohl in den drei Testszenarios die Anzahl Case-Instanzen zweimal verfünffacht wurde, blieb die Relationszeit auf einem ähnlichen Niveau. Im Gegensatz dazu erhöhte sich die Reaktionszeit bei der Schnittstelle zur Abfrage von ausführbaren Actions von Testszenario 1 zu Testszenario 3 um ungefähr den Faktor 15, obwohl die Anzahl verbundener Clients gleich geblieben ist.

# 10 Offene Punkte

Dieses Kapitel beschreibt Anforderungen an das oBPM-System, welche zwar Teil der Aufgabenstellung dieser Arbeit waren, jedoch während der Umsetzung nicht abgedeckt wurden oder konnten.

## 10.1 Umsetzung eines User-Interfaces

Wie in der Anforderung 006 beschrieben, war die Umsetzung eines User-Interfaces auf Basis eines End-User-Clients zur Modellierung und Ausführung von Prozessen Teil der erweiterten Erwartungen an diese Arbeit. Durch den begrenzten Zeitraum war die Umsetzung eines funktionierenden User-Interfaces nicht möglich. Zwar besitzt eine Benutzeroberfläche keine hohe Komplexität, was die Umsetzung sehr vereinfacht. Jedoch nimmt die Implementation eines gut zu bedienenden Interfaces überproportional viel Zeit in Anspruch, weshalb in dieser Arbeit auf die Umsetzung verzichtet wurde.

## 10.2 Versionierung von Prozess-Elementen

Wie unter der Anforderung 007 erfasst, wurde als erweiterte Erwartung die Versionierung von Prozess-Elementen definiert, inklusive Prozess-Modell-Elemente wie auch Prozess-Daten-Elemente. Ähnlich wie bei der Umsetzung des User-Interfaces verunmöglichte der beschränkte Zeitraum die korrekte Implementation der Versionierung von Prozess-Elementen. Zusätzlich erhöht die Möglichkeit der Versionierung die Komplexität des oBPM-Systems beträchtlich. Auch muss festgestellt werden, dass mit dem aktuellen Wissenstand noch keine Version der Implementierung bekannt ist, welche eine Versionierung der Prozess-Elementen erlaubt. So müssen vor einer allfälligen Implementation zuerst die Anforderungen und Auswirkungen der Implementation analysiert werden, was aus zeitlichen Gründen nicht möglich war.

## 10.3 Korrekte Implementation des Bottom-Up-Ansatzes

Das Kapitel *2.1.2 Bottom-Up-Perspektive* in der Auflistung 1 beschreibt, muss es einem Knowledge-Worker möglich sein, Prozess-Actions seiner Rolle, benötigte Dokumente einer Prozess-Action und Status-Übergänge hinzuzufügen, zu bearbeiten oder zu löschen. Dies erfordert dezidierte API-Schnittstellen, welche die partielle Bearbeitung von Prozess-Actions erlaubt. Im umgesetzten oBPM-System ist es nur Benutzern der Benutzer-Rolle *modeler* erlaubt, Prozess-Actions zu manipulieren, was es

Knowledge-Workern und anderen Benutzern, welche nicht der Rolle *modeler* zugewiesen sind, verunmöglicht, die in Auflistung 1 definierten Funktionalitäten vorzunehmen.

# 11 Resultat und Konklusion

Dieses Kapitel fasst die Resultate dieser Arbeit zusammen, welche während der Entwicklung des oBPM-Prototypen erlangt werden konnten. Dazu gehört die Analyse der Anforderungen, des verwendeten Datenbank-Systems und die Erfahrungen, welche während der Umsetzung des Prototyps gemacht werden konnten.

Die Analyse des ArangoDB Datenbank-Systems hat aufgezeigt, dass der Multi-Model-Ansatz von Arango GmbH (2016h) ausgezeichnet geeignet ist, um den in den Anforderungen von Grünert u. a. (o. J.-a, o. J.-b) und Grünert (2015) definierten Funktionsumfang eines oBPM-Systems zu implementieren. Dazu konnten die JSON-basierten Collection- und Dokumentstrukturen von ArangoDB eingesetzt werden, um die dynamischen Prozessmodelle und Prozessdaten-Strukturen zu persistieren. Zusätzlich wurde der Support von Graphen von ArangoDB zu Hilfe genommen, um Referenzen innerhalb von Datenstrukturen abbilden zu können. ArangoDB erlaubt eine vollumfängliche Umsetzung der benötigten Datenstrukturen, welche für ein oBPM-System benötigt werden.

Nach der Analyse verschiedener Implementationsvarianten von Systemarchitekturen konnte abgeleitet werden, dass sich Architektur basierend auf einem separaten Application-Server am besten eignet um die vom oBPM-System geforderten API-Schnittstellen zu implementieren. Dank des Application-Servers konnten alle relevanten Schnittstellen umgesetzt werden, um mit dem oBPM-System interagieren zu können. Die Node.js-Umgebung hat sich des Weiteren gut bewährt für die Implantation eines Security-Layers, um den Anforderungen an die Rechtevergabe und Authentifizierung und Autorisierung von Benutzern gerecht zu werden.

Der letzte Teil der in dieser Thesis gestellten Forschungsfrage betrifft die Performance und Skalierbarkeit eines oBPM-Systems basierend auf dem Datenbank-System ArangoDB. Es konnte dank ausführlichen Performance-Tests aufgezeigt werden, dass die zu erreichenden Richtwerte nicht erreicht werden konnten. Zwar erwies ein Teil der

Implementation als gut skalierbar, jedoch lag die gemessene Reaktionszeit in allen durchgeführten Tests über dem Richtwert von zwei Sekunden.

Die Konklusion dieser Thesis hält fest, dass die verwendeten Komponenten ArangoDB und der Node.js-basierte Application-Server sehr gut geeignet sind für die Umsetzung eines oBPM-basierten Prozess-Management-Systems. Das während dieser Arbeit entwickelte System befindet sich jedoch klar im Zustand eines Prototyps. Die gemessene Performance und Skalierbarkeit des Systems zeigt auf, dass das System noch nicht für den produktiven Einsatz bereit ist. Bevor der Prototyp in der Praxis eingesetzt werden kann, müssen die Vorgänge, welche für die Abfrage und Manipulation von Daten zuständig sind, optimiert werden.

# 12 Handlungsempfehlung

Dieses Kapitel fasst mögliche nächste Schritte zusammen, welche in Zukunft umgesetzt werden können, um die Umsetzung eines oBPM-basierten Prozessautomatisierungs-Systems auf Basis von ArangoDB vorwärts zu treiben.

Bei der Weiterentwicklung des oBPM-Prototypen muss vor allem auf die Performance und Skalierbarkeit geachtet werden, um ihn für den praktischen Einsatz in der Industrie vorzubereiten. Eine Möglichkeit besteht dabei in der Umverteilung der Business-Logik, welche für die Abfrage und Manipulation von Daten verantwortlich ist. Auf Basis der in Kapitel *5.4 Architekturentscheid*

getroffenen Entscheidungen wurde der grösste Teil der Geschäftslogik des Prototypen im Application-Server implementiert. In einer zukünftigen Weiterentwicklung kann versucht werden, Die Logik zur Verwaltung der Daten näher an das Datenbank-System zu rücken, in dem die Logik in Form von Foxx-Services wie in Kapitel *5.2 Client-Datenbank-Architektur* implementiert wird.

Schlussendlich könnte die zukünftige Implementation eines End-User-Clients aufzeigen, wie gut sich die implementierten API-Schnittstellen für den praktischen Einsatz des oBPM-Systems eignen. Die Umsetzung eines zum entwickelten Prototypen kompatiblen Clients kann aufzeigen, welche weiteren API-Schnittstellen notwendig sind, um eine effiziente und effektive Kommunikation zwischen Client uns System zu gewährleisten.

# 14 Literaturverzeichnis


Arango GmbH. (2015). Running V8 isolates in a multi-threaded ArangoDB database - ArangoDB. Abgerufen 15. Mai 2016, von https://www.arangodb.com/2015/08/running-v8-isolates-in-a-multi-threaded-arangodb-database/

Arango GmbH. (2016a). ArangoDB - the multi-model NoSQL DB. Abgerufen 5. Mai 2016, von https://www.arangodb.com

Arango GmbH. (2016b). Benchmark: PostgreSQL, MongoDB, Neo4j, OrientDB and ArangoDB. Abgerufen 5. Mai 2016, von https://www.arangodb.com/2015/10/benchmark-postgresql-mongodb-arangodb/

Arango GmbH. (2016c). Built in JavaScript Extensibility Framework Foxx. Abgerufen 8. Mai 2016, von https://www.arangodb.com/foxx/

Arango GmbH. (2016d). Collections | ArangoDB 2.8.6 Documentation. Abgerufen 4. Mai 2016, von https://docs.arangodb.com/FirstSteps/CollectionsAndDocuments.html

Arango GmbH. (2016e). Debugging | ArangoDB 2.8.6 Documentation. Abgerufen 15. Mai 2016, von https://docs.arangodb.com/Foxx/Develop/Debugging.html

Arango GmbH. (2016f). JavaScript Modules | ArangoDB 2.8.6 Documentation. Abgerufen 15. Mai 2016, von https://docs.arangodb.com/ModuleJavaScript/

Arango GmbH. (2016g). Key Features | Joins. Abgerufen 5. Mai 2016, von https://www.arangodb.com/key-features/#joins

Arango GmbH. (2016h). Key Features | Multi-Model. Abgerufen 4. Mai 2016, von https://www.arangodb.com/key-features/#multi-model

Arango GmbH. (2016i). Named Operations | ArangoDB 2.8.6 Documentation. Abgerufen 5. Mai 2016, von https://docs.arangodb.com/Aql/GraphOperations.html

Arango GmbH. (2016j). Working with Edges | ArangoDB 2.8.6 Documentation. Abgerufen 5. Mai 2016, von https://docs.arangodb.com/Edges/

Arndt, C., Hermanns, C., Kuchen, H., & Poldner, M. (2009). *Best Practices in der Softwareentwicklung*. F{ö}rderkreis der Angewandten Informatik an der Westf{ä}lischen Wilhelms-Universit{ä}t M{ü}nster.

Bandara, W., Indulska, M., Chong, S., & Sadiq, S. (2007). Major issues in business process management: an expert perspective.

Dohmen, L., Klamma, P. D. R., & Celler, F. (2012). *Algorithms for large networks in the nosql database arangodb*. Bachelors thesis, RWTH Aachen, Aachen.

Fernández, A. (2016). loadtest. Abgerufen von https://www.npmjs.com/package/loadtest

Fielding, R., Gettys, J., Mogul, J., Frystyk, H., Masinter, L., Leach, P., & Berners-Lee, T. (2006). Hypertext transfer protocol–HTTP/1.1, 1999. *RFC2616*. http://doi.org/https://dx.doi.org/10.17487/rfc2616

Fielding, R., & Reschke, J. (2014). Hypertext Transfer Protocol (HTTP/1.1): Authentication. http://doi.org/https://dx.doi.org/10.17487/rfc7235

Fraternali, P., Rossi, G., & Sánchez-Figueroa, F. (2010). Rich internet applications. *Internet Computing, IEEE*, *14*(3), 9–12.

Galiegue, F., & Zyp, K. (2013). JSON Schema: Core definitions and terminology. *Internet Engineering Task Force (IETF)*.

Github Inc. (2016). Contributors to expressjs/express. Abgerufen 21. Mai 2016, von https://github.com/expressjs/express/graphs/contributors

Grünert, D. (2015). *Aufgabenstellung Bachelor-Thesis - oBPM Implementation*. Winterthur, Switzerland.

Grünert, D., Brucker-Kley, E., & Keller, T. (o. J.-a). Artifact-Centric Modeling of Business Processes Using UML Diagrams.

Grünert, D., Brucker-Kley, E., & Keller, T. (o. J.-b). *oBPM - An Opportunistic Approach to Business Process Modeling and Execution*. Winterthur, Switzerland.

Jacobson, D., Woods, D., & Brail, G. (2011). *APIs: A strategy guide*. « O'Reilly Media, Inc.»

Jakl, M. (2005). Representational State Transfer. Citeseer.

keymetrics Inc. (2015). pm2. Abgerufen von http://pm2.keymetrics.io/

Maatouki, A., Meyer, J., Szuba, M., & Streit, A. (2015). A horizontally-scalable multiprocessing platform based on Node. js. In *Trustcom/BigDataSE/ISPA, 2015 IEEE* (Bd. 3, S. 100–107). IEEE.

Mapanga, I., & Kadebu, P. (2013). Database Management Systems: A NoSQL Analysis. *Interna-tional Journal of Modern Communication Technologies & Research (IJMCTR)*, *1*, 12–18.

Miller, J. J. (2013). Graph database applications and concepts with Neo4j. In *Proceedings of the Southern Association for Information Systems Conference, Atlanta, GA, USA* (Bd. 2324).

Ngamakeur, K., Yongchareon, S., & Liu, C. (2012). A framework for realizing artifact-centric business processes in service-oriented architecture. In *Database Systems for Advanced Applications* (S. 63–78).

Nikhil, M. (2014). An Introduction to libuv. Abgerufen 15. Mai 2016, von https://nikhilm.github.io/uvbook/introduction.html

Node.js Foundation. (2016). Node.js. Abgerufen 15. Mai 2016, von https://nodejs.org/en/

npm Inc. (2016). npm - express. Abgerufen 21. Mai 2016, von https://www.npmjs.com/package/express

OrientDB LTD. (o. J.). OrientDB - OrientDB Distributed Graph DatabaseOrientDB Distributed Graph Database. Abgerufen 6. Mai 2016, von http://orientdb.com/orientdb/

Redding, G., Dumas, M., ter Hofstede, A. H. M., & Iordachescu, A. (2010). A flexible, object-centric approach for business process modelling. *Service Oriented Computing and Applications*, *4*(3), 191–201.

Selfa, D. M., Carrillo, M., & Boone, M. del R. (2006). A database and web application based on MVC architecture. In *Electronics, Communications and Computers, 2006. CONIELECOMP 2006. 16th International Conference on* (S. 48).

Silberschatz, A., Korth, H. F., & Sudarshan, S. (1997). *Database system concepts* (Bd. 4). McGraw-Hill New York.

Stohr, E. A., & Zhao, J. L. (2001). Workflow automation: Overview and research issues. *Information Systems Frontiers*, *3*(3), 281–296.

StrongLoop, & IBM. (2016a). Express - Node.js web application framework. Abgerufen 21. Mai 2016, von http://expressjs.com/

StrongLoop, & IBM. (2016b). Express basic routing. Abgerufen 22. Mai 2016, von

http://expressjs.com/en/starter/basic-routing.html

StrongLoop, & IBM. (2016c). Serving static files in Express. Abgerufen 22. Mai 2016, von http://expressjs.com/en/starter/static-files.html

StrongLoop, & IBM. (2016d). Using Express middleware. Abgerufen 22. Mai 2016, von http://expressjs.com/en/guide/using-middleware.html

Wang, J., & Kumar, A. (2005). A framework for document-driven workflow systems. In *Business Process Management* (S. 285–301). Springer.

# 15 Anhang

## 15.1 Inhaltsangabe der Daten-CD

Die dieser Arbeit beigelegten Daten-CD beinhaltet folgende Datenstrukturen:

- Prototypische Implementation einer oBPM-Umgebung.pdf: Die von Remo Zumsteg verfasste Bachelor-Thesis im PDF-Format.
- Referenzen: Ein Ordner, welche alle verwendeten Referenzen der Thesis im PDF-Format enthält.
- Source-Code: Ein Ordner, welcher den gesamten Source-Code und alle Konfigurationsdateien des oBPM-Prototypen beinhaltet.
- Postman: Ein Ordner, welche Request-Templates für oBPM-API-Tests beinhaltet.
- oBPM Live Präsentation.mp4: Ein Video-Tutorial zur Benutzung des oBPM-Prototypen.

## 15.2 Installation, Inbetriebnahme und Ausführen von Tests

Dieser Anhang beschreibt das Vorgehen zur Installation und Inbetriebnahme des oBPM-Prototypen. Dazu müssen die folgenden Schritte in der korrekten Reihenfolge ausgeführt werden:

1. Installation und Starten der neusten Version der ArangoDB-Datenbank auf dem Standard-Port 8529.
2. Installation der neusten Version von Node.js
3. In einem Terminal oder Shell den im Source-Code-Verzeichnis liegenden Ordner `api` öffnen. Das Current-Working-Directory muss das Verzeichnis `api` sein.
4. Ausführen des Befehls `build.bat`, bzw. `build.sh` zum Kompilieren des Prototyps.
5. Im Verzeichnis `api` den Befehl `npm start` ausführen um den Prototypen auf Port 8090 zu starten.

Alle vom Prototypen benötigten Datenbanken und Standard-Dokumente werden automatisch erstellt.

Um alle API-Tests automatisiert ausführen zu lassen, kann im Verzeichnis `api` der Befehl `npm test` ausgeführt werden.

## 15.3 API-Testszenarios

Tabelle 46: API-Testresultate

| ID | Beschreibung | Status |
|---|---|---|
| **T0** | **Test-Vorbereitung** | |
| T0.01 | Authentifizierung des Benutzers *admin* | OK |
| T0.02 | Erstellen der Benutzer *modeler*, *teacher1*, *teacher2*, *student1*, *student2* | OK |
| T0.03 | Authentifizierung aller erstellten Benutzer | OK |
| **T1** | **Erstellung eines Datenmodells unter dem Benutzer *modeler*** | |
| T1.01 | Erstellen des Typs Case | OK |
| T1.02 | Erstellen des Typs Thesis | OK |
| T1.03 | Erstellen des Typs Person unter dem Attribut *student* | OK |
| T1.04 | Erstellen des Typs Person unter dem Attribut *professor* | OK |
| T1.05 | Erstellen des Typs Upload | OK |
| T1.06 | Überprüfen der Objektstruktur des Datenmodells. | OK |
| T1.07 | Überprüfen ob der Typ Case bearbeitbar ist. | OK |
| T1.08 | Erstellen des Typs Test1 und Test2 | OK |
| T1.09 | Ändern des Parent-Typs von Typ Test1 | OK |
| T1.10 | Überprüfen der Objektstruktur des Datenmodells. | OK |
| T1.11 | Typ Test1 und alle angehängten Typen löschen | OK |
| **T2** | **Erstellen von Action-Definitionen unter dem Benutzer *modeler*** | |
| T2.01 | Action *Create Thesis* für *teacher* erstellen | OK |
| T2.02 | Action *Assign Professor* für *teacher* erstellen | OK |
| T2.03 | Action *Assign Student* für *teacher* erstellen | OK |
| T2.04 | Action *Upload Document* für *student* erstellen | OK |
| T2.05 | Action *Reject Upload* für *teacher* erstellen | OK |

| | | |
|---|---|---|
| T2.06 | Action *Accept Upload* für *teacher* erstellen | OK |
| T2.07 | Action *Edit Upload* für *student* erstellen | OK |
| **T3** | **Ausführen von Actions** | |
| T3.01 | Überprüfen der ausführbaren Actions für *teacher1* | OK |
| T3.02 | Überprüfen der ausführbaren Actions für *student1* und *student2* | OK |
| T3.03 | Ausführen der Action *Create Thesis* als *teacher1* | OK |
| T3.04 | Überprüfen der Prozessdokumente | OK |
| T3.05 | Überprüfen der ausführbaren Actions für *teacher1* | OK |
| T3.06 | Ausführen der Action *Assign Professor* als *teacher2* | OK |
| T3.07 | Ausführen der Action *Assign Student* als *teacher1* | OK |
| T3.08 | Überprüfen der ausführbaren Actions für *student1* | OK |
| T3.09 | Überprüfen der ausführbaren Actions für *student2* | OK |
| T3.10 | Ausführen der Action *Upload Document* als *student1* | OK |
| T3.11 | Überprüfen der ausführbaren Actions für *teacher1* | OK |
| T3.12 | Überprüfen der ausführbaren Actions für *teacher2* | OK |
| T3.13 | Ausführen der Action *Reject Upload* als *teacher1* | OK |
| T3.14 | Ausführen der Action *Edit Upload* als *student1* | OK |
| T3.15 | Ausführen der Action *Accept Upload* als *teacher1* | OK |