ulm university universität
uulm

Universität Ulm | 89069 Ulm | Germany

Fakultät für Ingenieurwissenschaften und Informatik
Institut für Datenbanken und Informationssysteme

Konzeption und Entwicklung eines Cloud-basierten Persistenz-Systems für kollaboratives Checklisten-Management

Bachelorarbeit an der Universität Ulm

Vorgelegt von:
Daniel Reich
daniel.reich@uni-ulm.de

Gutachter:
Prof. Dr. Manfred Reichert

Betreuer:
Nicolas Mundbrod

2013

Fassung 14. November 2013

proCollab

Kurzfassung

Aufgrund der wachsenden Reife von prozessorientierten Informationssystemen können Prozesse in vielen Branchen automatisiert und teilweise unabhängig von Menschen bearbeitet werden. Jedoch gibt es Prozesse, die nicht automatisiert werden können, da sie sehr komplex und dynamisch sind und stark von den beteiligten Personen mit hohem Fachwissen abhängen. Jedoch sind die Menschen, die in diesen wissensintensiven Prozessen involviert sind, bis heute alleingelassen und vermissen eine adäquate prozessorientierte Unterstützung. Um in Zukunft wissensintensive Prozesse und die beteiligten Menschen besser unterstützen zu können, wird im Rahmen eines Forschungsprojektes (proCollab) an der Universität Ulm überprüft, ob sich digitale Checklisten verbunden mit einem innovativen Ansatz dafür eignen. Um Checklisten durch ein Informationssystem verfügbar zu machen, muss bei der Konzeption und Erstellung der darunterliegenden Datenverwaltung, auch Persistenzschicht genannt, auf verschiedenste Aspekte geachtet werden.

Die Arbeit beschreibt die Konzeption und prototypische Entwicklung einer Persistenzschicht für den proCollab-Prototypen, einem System zur Erstellung und kollaborativer Bearbeitung von Checklisten. Das Hauptaugenmerk der Konzeption befasst sich mit Besonderheiten von Verteilbarkeit und Skalierbarkeit im Bezug auf Datenbanken. Ebenfalls wird die Speicherung von hierarchisch strukturierten Daten in verschiedenen Datenbank-Architekturen betrachtet. Die Entwicklung beschreibt die Umsetzung einer Persistenz-Schicht mit Objektrelationaler Abbildung und den damit verbundenen Problemen.

Inhaltsverzeichnis

1

Einleitung

1.1. Motivation

Die heutige Zeit ist geprägt von automatisierten Prozessen, die oft selbstständig Tag und Nacht durch Informationstechnik abgearbeitet werden. Jedoch gibt es neben diesen automatischen Prozessen auch solche, die lediglich von Menschen bearbeitet werden können, da sie komplexes und kreatives Denken verlangen. Da diese wissensintensive Prozesse sehr dynamisch sind und auf Änderungen basierend auf vielen verschiedenen Rahmenbedingungen und Hintergrundwissen reagieren müssen, können diese Prozesse nicht oder sehr schlecht maschinell abgebildet werden.
Um die Menschen, die an diesen Prozessen arbeiten, besser zu unterstützen, können Checklisten verwendet werden. Checklisten bieten intuitiv einen Rahmen für den Prozess-Fortschritt, an dem man sich orientieren kann. Des Weiteren kann durch eine

Checkliste auch die Qualitätssicherung profitieren, da der Ablauf und noch anstehende Schritte stets geprüft werden können. Jedoch ist die digitale Nutzung von Checklisten für wissensintensive Prozesse bis heute schlecht erforscht. Am Institut für Datenbanken und Informationssysteme der Universität Ulm wird im Rahmen des Projekts proCollab ein Prototyp entwickelt. Hierbei handelt es sich um ein Checklisten-System, welches Nutzern unabhängig vom Standort gleichzeitiges Arbeiten an Checklisten ermöglicht. Diese Checklisten können auf die vorliegenden Prozesse angepasst und dadurch optimiert werden.
Gleichzeitiges Arbeiten mehrerer Menschen an einem Prozess gekoppelt mit stark verteilten Geschäftsstrukturen, wie sie in vielen Unternehmen heutzutage vorliegen, erschwert dieses Unterstützen durch Checklisten jedoch. Die Einbindung von Menschen, die sich an verschiedenen Orten aufhalten, bedeutet für den proCollab Prototypen, dass dieser die anfallenden Daten zwischen verschiedenen Instanzen des Prototypen koordinieren muss, um die Zugriffsgeschwindigkeit und Nutzbarkeit des Systems zu bewahren.
Die gleichzeitige Bearbeitung des Prozesses von mehreren Menschen birgt ebenfalls Probleme für das Datenverwaltungs-System, da durch parallele Zugriffe auf Daten bekannte Probleme der Nebenläufigkeit auftreten können. Sollte z.B. ein Nutzer eine Checkliste ändern, die ein entfernter Nutzer gleichzeitig angefordert hat, müssen diese Änderungen weitergeleitet und vor der Ausgabe bearbeitet werden. Wenn die Änderung für die Ausgabe unterbrochen oder der Zugriff auf die Checkliste während der Änderung für andere Nutzer nicht gesperrt wird, erhält der anfragende Nutzer eine veraltete Checkliste [BG81].

1.2. Ziel und Beitrag

Ziel der Arbeit ist es, eine Persistenzschicht für den *proCollab* Prototypen zu entwickeln und prototypisch zu implementieren. Diese Schicht übernimmt das Speichern, Auslesen, Verändern und Suchen der zu verarbeitenden Daten. Hierarchisch strukturierte Daten wie Checklisten können jedoch nicht einfach direkt in eine Datenbank gespeichert wer-

den. Je nach zugrunde liegender Datenbank-Architektur muss eine Checkliste zuerst in eine von der Datenbank verarbeitbare Form gebracht werden, was ebenfalls Aufgabe einer Persistenzschicht ist. Bei der Entwicklung wird darauf geachtet, dass sehr viele Änderungen an den Checklisten durchgeführt werden können, da bereits das Abhaken einer Checklisten-Frage eine Änderung ist und dementsprechend an die Datenbank weitergereicht wird. Des Weiteren werden für die Unterstützung paralleler Zugriffe auf die Datenbank Mechanismen zur Zugriffskontrolle umgesetzt. Um eine Kommunikation mit anderen Schichten des proCollab Prototypen zu ermöglichen, werden Schnittstellen spezifiziert und implementiert. Die Struktur dieser Schnittstellen wird anhand der konzipierten Architektur und der Anforderungsanalyse ermittelt. Da für die Umsetzung der Persistenzschicht verschiedene Technologien zur Verfügung stehen, wird überprüft, inwiefern sich die jeweiligen Technologien für die Umsetzung eignen.

1.3. Struktur der Arbeit

Im folgenden Kapitel 2 werden die der Arbeit zugrundeliegenden Problematiken und das Gesamtprojekt *proCollab* vorgestellt. Daraufhin wird in Kapitel 3 die Anforderungen an ein Persistenzsystem für Checklisten analysiert und definiert. In Kapitel 4 wird der Entwurf der Persistenzschicht für den proCollab Prototypen, der die aufgestellten Anforderungen umsetzt, vorgestellt. Die prototypische Entwicklung dieser Konzeption wird in Kapitel 5 auszughaft dargestellt. Kapitel 6 gibt abschließend eine Zusammenfassung der Ergebnisse und einen Ausblick auf zukünftige Herausforderungen.

2

Grundlagen

In Kapitel 2 werden die Grundlagen dieser Arbeit vermittelt und in diesem Zusammenhang wissensintensive Prozesse erläutert (siehe Kapitel 2.1), das Projekt proCollab vorgestellt (siehe Kapitel 2.2), sowie verschiedene aktuelle Einsatzgebiete von Checklisten dargestellt (siehe Kapitel 2.3) und grundlegende Begriffe des Checklisten-Managements (siehe Kapitel 2.4) und der Persistenz (siehe Kapitel 2.5) beschrieben.

2.1. Wissensintensive Prozesse

Die Welt unterliegt einer stets wachsenden Automatisierung. Beginnend mit den Fließbändern in Automobilfabriken durchzieht dieser Prozess mittlerweile beinahe jeden Wirtschaftssektor. Wo dies jedoch noch nicht so ausgeprägt ist, sind Abläufe, die aufgrund von benötigtem Hintergrundwissen oder einer zu großen Komplexität nicht automa-

tisiert werden können. Zu solchen wissensintensiven Abläufen zählen Prozesse wie Arztvisiten, Sicherheits-Checks in einem Flugzeug oder die Entwicklung eines neuen Software-Systems.

Charakteristisch für solche wissensintensive Prozesse ist eine Zielorientierung, eine schrittweise Entstehung der Arbeitsabläufe, eine hohe Anzahl an Einflussfaktoren sowie eine während des Prozesses stetig größer werdende Wissensbasis aller Beteiligten[MKR13]. Aufgrund der hohen Dynamik und Schnelllebigkeit solcher Prozesse muss oft der aktuelle Stand neu bewertet bzw. schnell auf Veränderungen reagiert werden.

Das Fehlen einer Automatisierbarkeit erweist sich letzten Endes als Problem, da der Mensch in wissensintensiven Prozessen auch die größte Fehlerquelle darstellt.
Dem Auftreten dieser durch den Mensch induzierten Fehler kann allerdings vorgebeugt werden, indem man ihn beim Ausführen des Prozesses unterstützt. Eine Art dieser Unterstützung bieten Checklisten, einfache Listen, auf denen erledigte Aufgaben abgehakt werden können und so einfach abgelesen werden kann, was noch erledigt werden muss und was bereits erledigt ist.

2.2. Projekt proCollab

Im Rahmen des Projekts proCollab am Institut für Datenbanken und Informationssysteme an der Universität Ulm soll evaluiert werden, in wie weit Checklisten genutzt werden können, um wissensintensive Prozesse unterstützen zu können. Bei dem *proCollab* Prototypen, für welches diese Persistenz-Schicht entwickelt wird, handelt es sich deshalb um ein Checklisten-System, das wissensintensiven Prozesse unterstützen soll. Um die Kollaboration der Nutzer bei den unterschiedlichen Prozessen zu ermöglichen, können im System angemeldete Nutzer Checklisten, für die sie autorisiert sind, parallel zu anderen Nutzern bearbeiten. Um eine größtmögliche Erreichbarkeit des Systems zu ermöglichen, gibt es Anwendungen für Mobiltelefone und Tablet-PCs. Des weiteren können über eine spezielle Webapplikation Administratoren, welche für die Nutzerverwaltung und Pflege der Checklisten-Vorlagen zuständig sind, ebenfalls auf das System zugreifen.

Die anfallenden Daten werden von einer Serverkomponente verwaltet. Diese setzt das für die Checklisten entwickelte Zustandsmodell um und kontrolliert es. Ebenso überprüft sie bei Nutzeraktionen, ob der Nutzer noch am System authentifiziert ist und, sollte dies der Fall sein, ob der Nutzer die für die Aktion benötigten Rechte besitzt. Letzten Endes fragt die Serverkomponente Daten bei der Persistenz-Schicht an oder gibt sie zum Speichern weiter. Die Persistenz-Schicht ist direkt mit der Datenbank verbunden und ist grundsätzlich für die sichere Datenspeicherung und -ausgabe verantwortlich. Hinzu kommt, dass sie die Kommunikation zwischen den verschiedenen Knotenpunkten des Systems realisiert. Das Zusammenspiel dieser Schichten ist in der folgenden Abbildung dargestellt

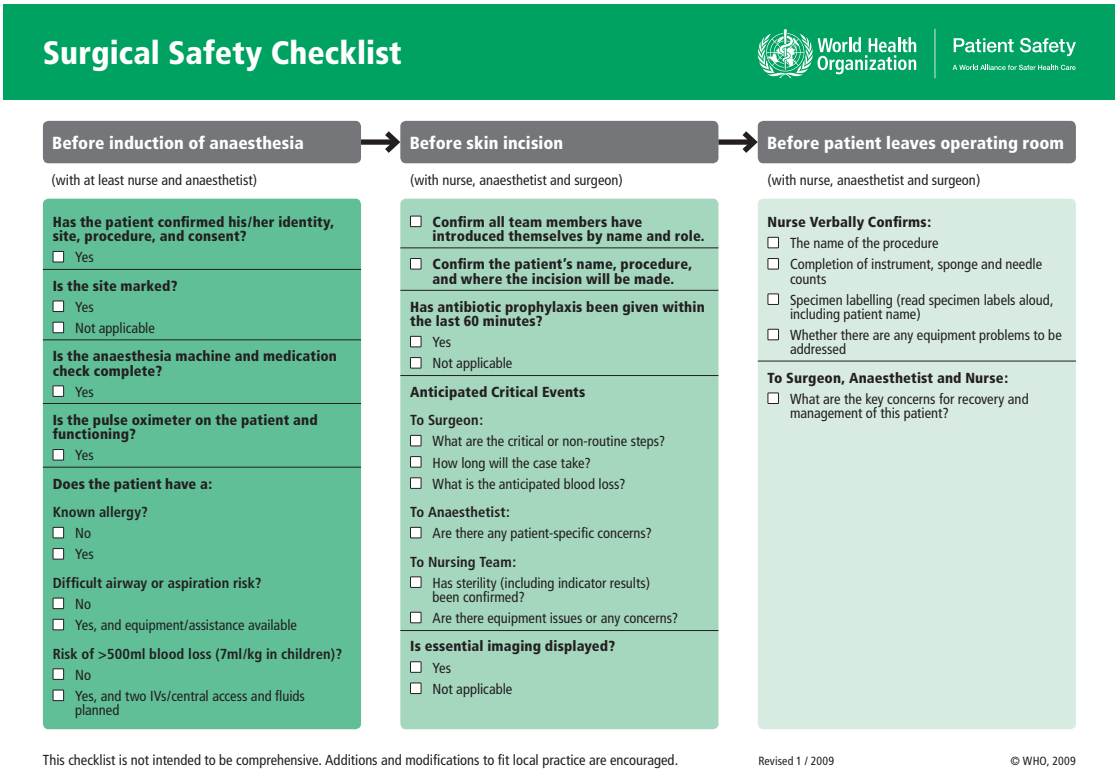
2.3. Anwendungsfälle

Checklisten bzw. computerbasierte Checklisten-Systeme finden bereits ihren Einsatz bei der Unterstützung wissensintensiver Prozesse. Diese unterscheiden sich jedoch stark in ihrer Granularität, dem verwendeten Medium und generell im Arbeitsbereich in dem sie eingesetzt werden.

Automobil-Branche Entwicklungsprojekte für mechatronische Bauteile sind in der Automobil-Branche aufgrund der verwendeten Entwicklungs-Stuktur in viele parallel ablaufende Prozesse unterteilt. Zur Unterstützung des Entwicklungsprozesses wird bei einem Automobilhersteller eine auf den Prozess angepasste Checkliste erstellt, wobei für jedes Entwicklungsprojekt spezifische Checklisten verwendet werden. Dies geschieht mithilfe von *IBM Rational Doors*, einem Werkzeug für das Requirements Engineering, das in diesem Fall die Verwaltung von Vorlagen für Checklisten erlaubt[IBM13]. Hierdurch werden bereits existierende Checklisten-Vorlagen auf den aktuellen Prozess angepasst und erweitert. Dies wird von einem Mitarbeiter der Qualitäts-Sicherung (QS) durchgeführt. Aus dieser spezifischen Checkliste werden Excel-Dateien erstellt, die für die entsprechenden Mitarbeiter zugänglich sind. Da Mitarbeiter nur lesenden Zugriff auf die erstellten Excel-Dateien haben,

können Änderungen an Checklisten lediglich von QS-Mitarbeitern durchgeführt werden [Mun13].

Medizin Die Genauigkeit und Korrektheit bei Prozessen in der Medizin, wie der Ablauf einer bestimmten Operation oder die Diagnose von Krankheiten, sind äußerst wichtig. Sollte ein Arzt z.B. bestimmte Arbeitsschritte vergessen oder verwechseln, kann dies schwere Folgen für den Patienten haben. Um Ärzte bei wissensintensiven Prozessen zu unterstützen und ihnen beispielsweise einen genauen Ablaufs-Plan einer Operation zu geben, werden Checklisten für diese Prozesse erstellt. Der grundsätzliche Rahmen einer solchen Checkliste ist ein medizinischer Fall, wobei es sich um den Krankheitsverlauf eines bestimmten Patienten handelt. In Abbildung 2.1 ist eine von der Weltgesundheitsorganisation erstellte Checkliste zu sehen, die als Basiskontrolllisten für Operationen dienen.



Surgical Safety Checklist

World Health Organization | Patient Safety
A World Alliance for Safer Health Care

Before induction of anaesthesia (with at least nurse and anaesthetist)

Has the patient confirmed his/her identity, site, procedure, and consent?
- ☐ Yes

Is the site marked?
- ☐ Yes
- ☐ Not applicable

Is the anaesthesia machine and medication check complete?
- ☐ Yes

Is the pulse oximeter on the patient and functioning?
- ☐ Yes

Does the patient have a:

Known allergy?
- ☐ No
- ☐ Yes

Difficult airway or aspiration risk?
- ☐ No
- ☐ Yes, and equipment/assistance available

Risk of >500ml blood loss (7ml/kg in children)?
- ☐ No
- ☐ Yes, and two IVs/central access and fluids planned

Before skin incision (with nurse, anaesthetist and surgeon)

- ☐ **Confirm all team members have introduced themselves by name and role.**
- ☐ **Confirm the patient's name, procedure, and where the incision will be made.**

Has antibiotic prophylaxis been given within the last 60 minutes?
- ☐ Yes
- ☐ Not applicable

Anticipated Critical Events

To Surgeon:
- ☐ What are the critical or non-routine steps?
- ☐ How long will the case take?
- ☐ What is the anticipated blood loss?

To Anaesthetist:
- ☐ Are there any patient-specific concerns?

To Nursing Team:
- ☐ Has sterility (including indicator results) been confirmed?
- ☐ Are there equipment issues or any concerns?

Is essential imaging displayed?
- ☐ Yes
- ☐ Not applicable

Before patient leaves operating room (with nurse, anaesthetist and surgeon)

Nurse Verbally Confirms:
- ☐ The name of the procedure
- ☐ Completion of instrument, sponge and needle counts
- ☐ Specimen labelling (read specimen labels aloud, including patient name)
- ☐ Whether there are any equipment problems to be addressed

To Surgeon, Anaesthetist and Nurse:
- ☐ What are the key concerns for recovery and management of this patient?

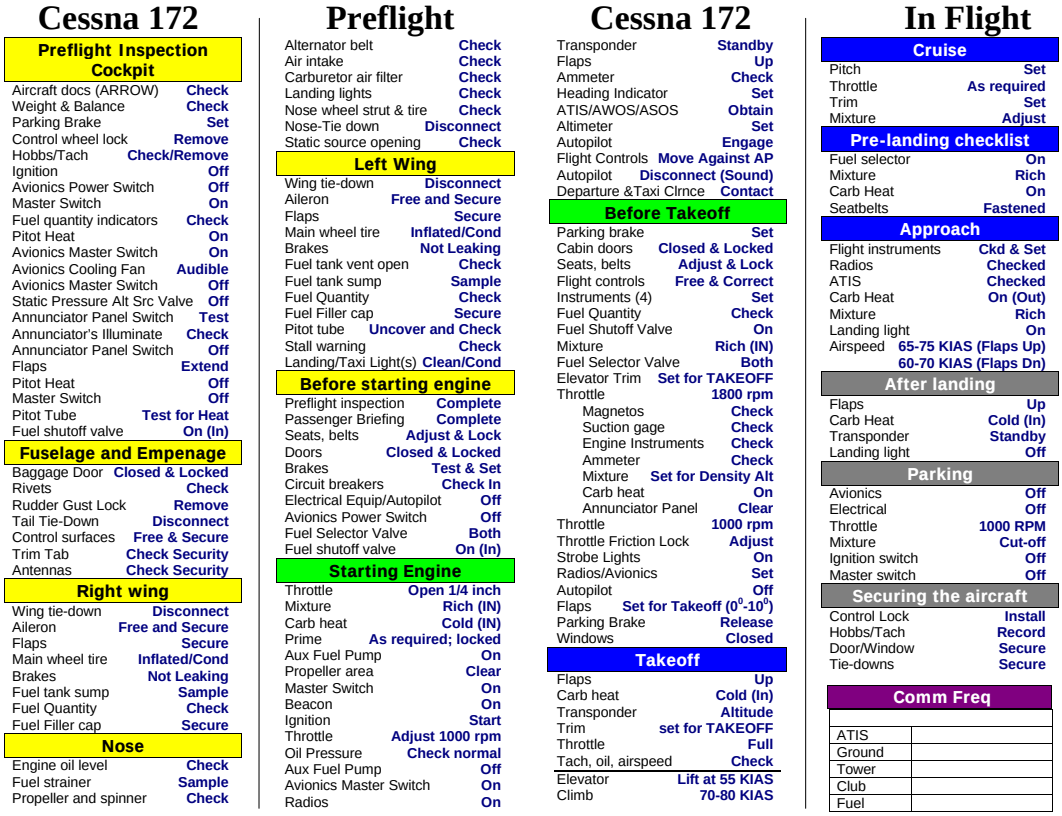
This checklist is not intended to be comprehensive. Additions and modifications to fit local practice are encouraged.

Revised 1 / 2009

Abbildung 2.1.: Sicherheits-Checkliste für Operationen [Wor09]

Die Checkliste besitzt ankreuzbare Fragen, wodurch festgestellt werden kann, ob bestimmte Schritte der Operation und Operationsvorbereitung bereits durchgeführt wurden und welche Schritte als nächstes durchgeführt werden müssen. Diese Checkliste dient als Vorlage und kann auf den Fall, für den diese Liste verwendet wird, angepasst werden.

Luftfahrt Die Luftfahrt ist ein sehr sicherheitskritischer Bereich, da bereits kleine maschinelle Fehler oder menschliches Versagen große Auswirkungen wie eine Absturz zur Folge haben kann. Um sowohl den maschinellen Fehlern und dem menschlichen Versagen entgegenzuwirken, finden in der Luftfahrt Checklisten schon seit langem Anwendung. Für jeden Flugzeug-Typ existieren Checklisten, die Piloten vor dem Start des Fluges abarbeiten. Sollten gewisse Punkte der Checklisten von den Norm-Werten abweichen, kann es beispielsweise bedeuten, dass das Flugzeug beschädigt ist und vor dem Start noch gewartet werden muss. Durch diese Checklisten werden neben der Flugtauglichkeit des Flugzeugs auch die für den Start nötige Geschwindigkeit oder die korrekte Treibstoff-Mischung überprüft. In Abbildung 2.2 ist eine Checkliste für den Flugzeugtyp Cessna 172 zu sehen. Diese Checkliste ist im Vergleich zur Checkliste in Abbildung 2.1 eine andere Art, da die Fragen der Checkliste in Abbildung 2.2 nicht ankreuzbar sind, sondern die regulären Soll-Werte dieses Flugzeugs in bestimmten Flugphasen enthält, die mit den aktuellen Ist-Werten des Flugzeugs verglichen werden.



Cessna 172

Preflight Inspection Cockpit

Aircraft docs (ARROW)	Check
Weight & Balance	Check
Parking Brake	Set
Control wheel lock	Remove
Hobbs/Tach	Check/Remove
Ignition	Off
Avionics Power Switch	Off
Master Switch	On
Fuel quantity indicators	Check
Pitot Heat	On
Avionics Master Switch	On
Avionics Cooling Fan	Audible
Avionics Master Switch	Off
Static Pressure Alt Src Valve	Off
Annunciator Panel Switch	Test
Annunciator's Illuminate	Check
Annunciator Panel Switch	Off
Flaps	Extend
Pitot Heat	Off
Master Switch	Off
Pitot Tube	Test for Heat
Fuel shutoff valve	On (In)

Fuselage and Empenage

Baggage Door	Closed & Locked
Rivets	Check
Rudder Gust Lock	Remove
Tail Tie-Down	Disconnect
Control surfaces	Free & Secure
Trim Tab	Check Security
Antennas	Check Security

Right wing

Wing tie-down	Disconnect
Aileron	Free and Secure
Flaps	Secure
Main wheel tire	Inflated/Cond
Brakes	Not Leaking
Fuel tank sump	Sample
Fuel Quantity	Check
Fuel Filler cap	Secure

Nose

Engine oil level	Check
Fuel strainer	Sample
Propeller and spinner	Check

Preflight

Alternator belt	Check
Air intake	Check
Carburetor air filter	Check
Landing lights	Check
Nose wheel strut & tire	Check
Nose-Tie down	Disconnect
Static source opening	Check

Left Wing

Wing tie-down	Disconnect
Aileron	Free and Secure
Flaps	Secure
Main wheel tire	Inflated/Cond
Brakes	Not Leaking
Fuel tank vent open	Check
Fuel tank sump	Sample
Fuel Quantity	Check
Fuel Filler cap	Secure
Pitot tube	Uncover and Check
Stall warning	Check
Landing/Taxi Light(s)	Clean/Cond

Before starting engine

Preflight inspection	Complete
Passenger Briefing	Complete
Seats, belts	Adjust & Lock
Doors	Closed & Locked
Brakes	Test & Set
Circuit breakers	Check In
Electrical Equip/Autopilot	Off
Avionics Power Switch	Off
Fuel Selector Valve	Both
Fuel shutoff valve	On (In)

Starting Engine

Throttle	Open 1/4 inch
Mixture	Rich (IN)
Carb heat	Cold (IN)
Prime	As required; locked
Aux Fuel Pump	On
Propeller area	Clear
Master Switch	On
Beacon	On
Ignition	Start
Throttle	Adjust 1000 rpm
Oil Pressure	Check normal
Aux Fuel Pump	Off
Avionics Master Switch	On
Radios	On

Cessna 172

Transponder	Standby
Flaps	Up
Ammeter	Check
Heading Indicator	Set
ATIS/AWOS/ASOS	Obtain
Altimeter	Set
Autopilot	Engage
Flight Controls	Move Against AP
Autopilot	Disconnect (Sound)
Departure &Taxi Clrnce	Contact

Before Takeoff

Parking brake	Set
Cabin doors	Closed & Locked
Seats, belts	Adjust & Lock
Flight controls	Free & Correct
Instruments (4)	Set
Fuel Quantity	Check
Fuel Shutoff Valve	On
Mixture	Rich (IN)
Fuel Selector Valve	Both
Elevator Trim	Set for TAKEOFF
Throttle	1800 rpm
Magnetos	Check
Suction gage	Check
Engine Instruments	Check
Ammeter	Check
Mixture	Set for Density Alt
Carb heat	On
Annunciator Panel	Clear
Throttle	1000 rpm
Throttle Friction Lock	Adjust
Strobe Lights	On
Radios/Avionics	Set
Autopilot	Off
Flaps	Set for Takeoff (0^0-10^0)
Parking Brake	Release
Windows	Closed

Takeoff

Flaps	Up
Carb heat	Cold (In)
Transponder	Altitude
Trim	set for TAKEOFF
Throttle	Full
Tach, oil, airspeed	Check
Elevator	Lift at 55 KIAS
Climb	70-80 KIAS

In Flight

Cruise

Pitch	Set
Throttle	As required
Trim	Set
Mixture	Adjust

Pre-landing checklist

Fuel selector	On
Mixture	Rich
Carb Heat	On
Seatbelts	Fastened

Approach

Flight instruments	Ckd & Set
Radios	Checked
ATIS	Checked
Carb Heat	On (Out)
Mixture	Rich
Landing light	On
Airspeed	65-75 KIAS (Flaps Up)
	60-70 KIAS (Flaps Dn)

After landing

Flaps	Up
Carb Heat	Cold (In)
Transponder	Standby
Landing light	Off

Parking

Avionics	Off
Electrical	Off
Throttle	1000 RPM
Mixture	Cut-off
Ignition switch	Off
Master switch	Off

Securing the aircraft

Control Lock	Install
Hobbs/Tach	Record
Door/Window	Secure
Tie-downs	Secure

Comm Freq

ATIS	
Ground	
Tower	
Club	
Fuel	

Abbildung 2.2.: Checkliste einer Cessna 172 [Wor08]

2.4. Begriffsdefinitionen für das Checklisten-Management

Auf Basis der Anwendungsfälle aus Kapitel 2.3 sollen im Folgenden die zentralen Begriffe des proCollab Prototypen eingeführt werden. Diese sind der **Organisatorische Rahmen** (siehe Kapitel 2.4.1), die **Checkliste** (siehe Kapitel 2.4.2) und der **Checklisteneintrag** (siehe Kapitel 2.4.3).

2.4.1. Organisatorischer Rahmen

Der *organisatorische Rahmen (OR)* dient der Festlegung und Definition bestimmter kontextueller Parameter und Bedingungen, die für die im OR enthaltenen Checklisten gelten oder für den unterstützten wissensintensiven Prozess relevant sind. In der Realität

stellt ein OR typischerweise ein Projekt, einen Fall oder eine schlichte Zusammenarbeit zwischen Menschen dar, die üblicherweise an wissensintensiven Prozessen arbeiten. Menschen, die an einem OR mitarbeiten, können je nach OR bestimmte Rollen zugewiesen werden. Ein OR kann eine oder mehrere Checklisten enthalten. Ein OR kann, vergleichbar zu Unterprojekten eines Projekts, untergeordnete ORs haben. Damit ist es möglich, dass ein kompletter Baum aus ORs entstehen kann. Ein OR wird stets unterteilt in OR-Typ und OR-Instanz.

Organisatorischer Rahmentyp

Bei einem *organisatorischen Rahmentyp (ORT)* handelt es sich um eine generische Vorlage eines OR, aus welcher eine organisatorische Rahmeninstanz (siehe Kapitel 2.4.1 erstellt werden kann. Die Parameter eines ORT (z.B. das Startdatum einer Zusammenarbeit), die angegeben werden, werden beim Erstellen einer Instanz an diese mitgegeben. Dies hat den Vorteil, dass Parameter, welche für viele OR-Instanzen gleich sind, nicht bei jedem Erstellen einer Instanz erneut eingegeben werden müssen.

Organisatorische Rahmeninstanz

Bei einer *organisatorischen Rahmeninstanz (ORI)* handelt es sich um einen konkreten OR, welcher für einen bestimmten Prozess angepasst und für diesen eingesetzt wird. Beispielsweise stellt eine ORI ein konkretes, laufendes Projekt der Software-Entwicklung dar. Eine ORI kann aus einem ORT (siehe Kapitel 2.4.1) erstellt werden, ist jedoch nicht zwingend erforderlich.

2.4.2. Checkliste

Die Checkliste (CL) ist das Kernelement im *proCollab* Projekt und seinem Prototypen. Sie dient der Darstellung und Überprüfung des Fortschritts im OR, dem sie zugeordnet ist. Ihr können bestimmte Personen zugeteilt werden, die sich ausschließlich mit der Bearbeitung dieser Checkliste befassen. Eine CL kann, vergleichbar mit einem OR,

untergeordnete CL haben. Sie enthält einen bis mehrere Checklisteneinträge (siehe Kapitel 2.4.3) und ist ebenfalls aufgeteilt in Typ (siehe Kapitel 2.4.2) und Instanz (siehe Kapitel 2.4.2).

Checklistentyp

Der Checklistentyp (CLT) ist, wie bereits der ORT eine Vorlage einer CL. Parameter einer CL lassen sich im CLT vordefinieren, sollten diese Parameter bei mehreren expliziten CL benötigt werden. Neben der Vorbereitung der Parameter können dem CLT Personen, welche alle aus diesem CLT erstellte Instanzen bearbeiten, zugewiesen werden. Ein CLT kann untergeordnete CLT und dem CLT hinzugefügte Checklisteneintragstypen (siehe Kapitel 2.4.3) besitzen. Vergleichbar ist ein CLT mit der in Abbildung 2.1 gezeigten Checkliste, die auf die Fälle je nach Bedarf angepasst werden kann.

Checklisteninstanz

Die Instanz einer Checkliste (CLI) kann entweder aus einem CLT oder ohne Vorlage erstellt werden. Konkret wäre eine CLI eine auf einen bestimmten Fall angepasste Checkliste der Vorlage in Abbildung 2.1. Diese CLI werden letztlich in dem durch eine ORI definierten Prozess bearbeitet.

2.4.3. Checklisteneintrag

Fragen, die Benutzer im Kontext einer CL beantworten bzw. abarbeiten, werden *Checklisteneinträge (CE)* genannt. Diese CEs sind in der CL-Hierarchie an unterster Stufe und können keine weiteren Elemente unter sich haben. Somit definieren sie einzelne Bearbeitungsschritte des zugeordneten Prozesses und sind in der Granularität am feinsten. Ein Beispiel eines CEs ist eine Frage der Checkliste in Abbildung 2.2. Bereits wie CL und OR ist ein CE in Typ und Instanz unterteilt.

Checklisteneintragstyp

Der Checklisteneintragstyp (CET) dient anderen CET oder konkreten Instanzen (siehe Kapitel 2.4.3) als Vorlage und kann, wie CLTs und ORTs, bei verschiedenen Parametern vordefiniert werden. Ein konkretes Beispiel eines CET ist der Eintrag über die Mindeststartgeschwindigkeit eines Flugzeugs, wobei für jeden Flugzeugtyp die erforderliche Geschwindigkeit angepasst werden kann. Anstatt lediglich einzelnen CEI als Vorlage zu dienen, werden typischerweise CETs CLTs zugeordnet, um mit ihnen Vorlagen für CLIs zu bilden.

Checklisteneintragsinstanz

Die *Checklisteneintragsinstanz (CEI)* ist ein auf den Prozess angepasster CE und wird entweder auf Basis eines CET oder ohne Vorlage erstellt. CEI werden typischerweise CLI zugeordnet anstatt ohne CLI erstellt zu werden und sind vergleichbar mit einer Frage der Checkliste in Abbildung 2.2, die auf den Flugzeugtyp Cessna 172 angepasst wurde.

2.5. Grundlegende Begriffe der Persistenz

2.5.1. Persistenz

Generell kann Persistenz verstanden werden als Dauerhaftigkeit oder Beständigkeit. In Bezug auf Daten bedeutet es, diese sicher und zuverlässig abzuspeichern und auszugeben.

Cache

Caching bedeutet, dass Daten in einem speziellen Speicher zwischengespeichert werden, damit diese, sollten sie später nochmals benötigt werden, nicht langsamer aus der Datenbank geladen werden müssen. Des weiteren können auch Änderungen an Dateien zwischengespeichert werden und erst zu einem späteren Zeitpunkt persistiert werden.

Das Problem hierbei ist jedoch, dass zum einen auf konkurrierende Zugriffe geachtet werden muss und zum anderen , dass es bei einem Systemausfall zu Datenverlusten kommen kann.

Datenbanksystem

Ein *Datenbanksystem* (DBS) ist ein Verbund aus einer Datenbank (DB) und einem Datenbankverwaltungssystem (DBMS). Die DB ist der eigentliche Datenspeicher des DBS und ist lediglich für die Datenhaltung zuständig. Welche Funktionalitäten von dem DBS angeboten werden, ist von dem verwendeten DBMS abhängig. Das DBMS dient als Schnittstelle für die in der DB gespeicherten Daten. Ein DBS-Knoten wird verstanden als einzelnes DBS in einem Verbund aus mehreren DBS.

Verschiedene Datenbankarchitekturen

Relationale Datenbanken Relationale DBSs speichern Daten in Tabellen und bilden Relationen zwischen Daten über Schlüssel ebenfalls auf Tabellen ab. Diese DBSs werden auch SQL-Datenbanken genannt, aufgrund der typischerweise verwendeten Anfragensprache, der *Stuctured Query Language* (SQL) [Int13]. Ein einzelner Datensatz entspricht einer Zeile in einer Tabelle. Welche Daten für die jeweiligen Datensätze gespeichert werden, ergibt sich aus dem Typ der Tabellenspalten. Relationen werden über Spalten in der Zieltabelle und Schlüssel, welche auf Ursprungs- und Zieltabelle verweisen, gespeichert. Diese Art des DBS ist die am weitest verbreitetste, bekannte Vertreter sind DB2 [db213], MySQL [mys13] und SQLite [sql13].

Nichtrelationale Datenbanken Anders als relationale Datenbanken besitzen nichtrelationale DBSs, auch NoSQL-Datenbanken genannt, keine Tabellenstruktur, sondern, je nach Typ, ein Schlüssel-Wert-Paar oder Dokumente [LM10]. Da diese DBS-Art nicht auf feste Tabellen basiert, ist sie hochskalierbar und sehr flexibel [FTD12]. Jedoch gerade wegen dieses schlichten Aufbaus bieten dieses DBS eher wenige Funktionalitäten an. Bekannte Vertreter dieser Architektur sind MongoDB [mon13], CouchDB [cou13] und BigTable [CDG08].

Graph-Datenbanken Graph-Datenbanken, eine spezielle Form von NoSQL-Datenbanken, speichern die Daten und Relationen untereinander als sowohl gerichteten als auch ungerichteten Graph. Die Graph-Knoten stellen die Datensätze und die Kanten die Relationen der Datensätze untereinander dar. Bekannte Graphdatenbanken sind Neo4j [neo13] und FlockDB.

Lazy Loading

Lazy Loading ist ein Design-Pattern, welches z.B. in der Java Persistence API umgesetzt wird. Es beschreibt, dass Objekte nur soweit wie nötig initialisiert und Daten, welche erst zu einem späteren Zeitpunkt gebraucht werden, dynamisch nachgeladen werden. Das Gegenteil hierzu und somit das Initialisieren der kompletten Daten eines Objekt wird *Eager Loading* genannt.

2.5.2. Verteilung

Verteilbarkeit im informationstechnischen Kontext ist die Auslagerung, Dezentralisierung und, daraus folgend, die Modularisierung von Diensten, Daten und Ressourcen sowohl lokal als auch logisch innerhalb des Systems, welche jedoch nach außen hin als einheitliches System gesehen werden. Jedoch entstehen eben durch diese Entkopplung verschiedener Komponenten Probleme, wie im Bezug auf die Datenverteilung, die Erhaltung der Persistenz und der sinnvollen Verteilung der Daten. Auf diese Problematik wird in Kapitel 4 näher eingegangen.

Load Balancing

Unter Load-Balancing versteht man im Datenbank-spezifischen Kontext die Verteilung der Anfragen an die verschiedenen einzelnen Knoten bzw. Datenbanken.

ACID vs. BASE

ACID steht für *Atomicity*, *Consistency*, *Isolation*, *Durability*. *Atomicity* beschreibt, dass eine Transaktion, z.B. eine DB-Anfrage, bei einem Fehler einer Teil-Transaktion komplett rückgängig gemacht wird. Somit lassen sich die Transaktionen als einzelne logische Einheit betrachten. Mit *Consistency* ist gemeint, dass keine nicht-validen Transaktionen durchgeführt werden dürfen. Dadurch wird sichergestellt, dass alle Integritätsbedingungen eingehalten werden. *Isolation* bedeutet, dass parallel am System eintreffende Transaktionen isoliert voneinander bearbeitet werden müssen. Unter *Durability* versteht man, dass Datenänderungen auch dauerhaft sind. Die durchgeführten Änderungen müssen somit auch nach einem Systemausfall noch vorhanden sein.

BASE, das für *Basically Available*,*Soft-state* und *Eventual consistency* steht, legt den Schwerpunkt im Gegensatz zu ACID auf Verfügbarkeit des DBS anstatt auf Datenkonsistenz [Pri08]. Unter *Basically Available* versteht man, dass das DBS selbst bei internen Fehlern verfügbar ist und Anfragen in gewisser Zeit bearbeitet. Mit *Soft-state* ist gemeint, dass die Daten des DBS auch inkonsistent sein können und *Eventual consistency* beschreibt, dass das DBS bei Inkonsistenzen nach unbestimmter Zeit in einen konsistenten Zustand überführt wird.

CAP-Theorem

Das CAP-Theorem teilt die Eigenschaften verteilter DBS in die drei Kategorien *Consistency* (Konsistenz), *Availability* (Verfügbarkeit) und *Partition tolerance* (Partitionstoleranz) ein, die zueinander in Beziehung stehen [Bre00]. Die Relation dieser drei Kategorien wird in Abbildung 2.5.2 dargestellt.

Konsistenz Die *Konsistenz* beschreibt, dass Daten auf allen DB-Knoten den selben Zustand haben. Somit müssen die Transaktionen atomar und bei allen DB-Knoten zur gleichen Zeit ausgeführt werden.

Verfügbarkeit Wenn jede Anfrage an das DBS in einem gewissen Zeitrahmen beantwortet wird, spricht man von *Verfügbarkeit*.

Partitionstoleranz Unter *Partitionstoleranz* versteht man, dass das DBS bei Verlust von Anfragen oder einzelner DB-Knoten weiterhin stabil funktioniert.

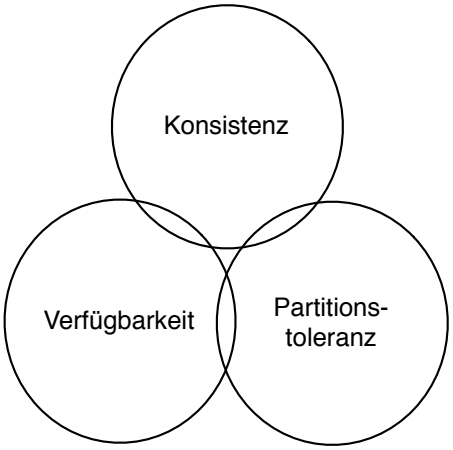


Abbildung 2.3.: Kategorien des CAP-Theorem

Das CAP-Theorem besagt, dass nie alle drei Eigenschaften umgesetzt werden können, sondern maximal zwei davon. Somit muss abgewägt werden, welche Eigenschaft man für die anderen beiden vernachlässigt. Dadurch entstehen die drei Kombinationen **Konsistenz und Verfügbarkeit** (CA), **Konsistenz und Partitionstoleranz** (CP) und **Verfügbarkeit und Partitionstoleranz** (AP).

CA DBSs, die diese Eigenschaften umsetzen, haben hochverfügbare und konsistente DBs. Jedoch wird der Umgang mit Partitionsfehlern schlecht unterstützt, was bei dem Einsatz vieler Partitionen zu Inkonsistenzen führen kann.

CP Da bei dieser Kombination die Datenkonsistenz und Partitionstoleranz im Vordergrund steht. Somit kann es vorkommen, dass das DBS bei einem Anfragefehler eines DB-Knotens nicht im vorgegebenen Zeitrahmen antworten kann, da eine Fehlerbehandlung über alle DB-Knoten hinweg durchgeführt werden muss.

AP Sollte ein DBS hochverfügbar und partitionstolerant sein, kann es vorkommen, dass sich die Daten eines DB-Knotens kurzzeitig in einem anderen Zustand als die Daten des restlichen DBS befinden. Sollte ein Anfragefehler bei einem einzelnen

DB-Knoten auftreten, wird die Fehlerbehandlung lediglich auf diesem Knoten durchgeführt, wodurch die Daten dieses Knotens zeitweise inkonsistent werden.

Partitionierung

Die *Partitionierung* bezogen auf DBS beschreibt die logische Unterteilung der DB. Es wird unterschieden zwischen der *horizontalen* und *vertikalen Partitionierung*.

Horizontale Partitionierung Sollten sehr viele Datensätze in einer DB gespeichert sein, kann sich dies auf die Performanz auswirken, da z. B. zum Auslesen eines bestimmten Datensatzes viele Datensätze durchlaufen werden müssen. Um die Menge der Datensätze in einer DB zu verringern, werden die Datensätze auf mehrere DBs mit gleicher Struktur aufgeteilt. Die Unterteilung der Datensätze kann über zwei verschiedene Partitionierungs-Methoden erreicht werden. Bei der *Hash*-Methode werden Datensätze mittels einer vordefinierten Hash-Funktion in die gewünschte Anzahl an Partitionen unterteilt. Bei der sogenannten *Range*-Methode gibt man gewünschte Bereiche oder Werte einer Spalte an, nach welchen die Datensätze aufgeteilt werden sollen. Diese Art der Verteilung wird horizontale Partitionierung oder Sharding genannt [NCWD84].

Vertikale Partitionierung Oft werden nur einzelne Attribute eines Datensatzes benötigt [NCWD84]. Wenn jedoch mehrere Anfragen verschiedene Attribute eines Datensatzes benötigen, wird dieser jeweils für die Dauer einer Anfrage gesperrt. Dadurch können Anfragen, die auf dem Datensatz ein anderes Attribut benötigen, dieses erst auslesen nachdem die Sperre aufgehoben wurde. Um dieses Problem zu beheben, kann die DB-Struktur in kleinere DBs aufgeteilt werden. Die unterschiedlichen Anfragen werden somit unabhängig voneinander abgearbeitet, was die Performanz des DBS erhöht. Dieses Verfahren wird *vertikale Partitionierung* genannt. Ein weiterer Vorteil der vertikalen Partitionierung ist, dass durch das Aufteilen die kleineren DBs übersichtlicher und für den Administrator besser zu verwalten sind. Sollten jedoch alle Attribute der partitionierten Tabelle benötigt werden, muss die ursprüngliche Tabelle über bestimmte Datenbankoperationen zusammengefügt werden.

Skalierbarkeit

Eine generelle Definition von Skalierbarkeit kann nicht bestimmt werden, da wirtschaftliche Skalierbarkeit grundsätzlich andere Ziele und Voraussetzungen hat als die informationstechnische Skalierbarkeit. Betrachtet man Skalierbarkeit im Kontext von DBS bedeutet es zum einen mehr Speicher zur Verfügung zu stellen und zum anderen Datenbankabfragen schneller zu bearbeiten. Für eine schnellere Bearbeitung kann der Datenbank mehr Hardware, wie schnellere Prozessoren oder mehr Systemspeicher hinzugefügt oder der für die Datenbankabfrage verwendete Programmcode bzw. die Datenbankstruktur optimiert werden. Generell lassen sich zwei Arten von Skalierbarkeit unterscheiden. Es gibt zum einen die *horizontale Skalierbarkeit*.

Horizontale Skalierbarkeit Unter *horizontaler Skalierbarkeit* versteht man das Erhöhen der Ressourcen durch Hinzufügen von weiteren Knoten zu einem Netz. Als Resultat hat man ein weiter gefächertes Netz von kleinen Knotenpunkten. Bezogen auf DBSs wäre *horizontale Skalierbarkeit* das hinzufügen weiterer DB-Knoten in das DBS.

Vertikale Skalierbarkeit Mit *vertikaler Skalierbarkeit* ist gemeint, dass anstatt kleinere Knoten zum Netz hinzugefügt werden, die bestehenden Knoten mit mehr Ressourcen versorgt werden, oder im datenbankspezifischen Kontext die Speichervergrößerung der bereits vorhandenen Knoten.

3

Anforderungsanalyse

In Kapitel 3 wird zuerst ein Anwendungsszenario geschildert (siehe Kapitel 3.1), danach Bedienvorgänge von Benutzern definiert (Kapitel 3.2), anschließend anhand dieser Vorgänge und weiterer benötigter Eigenschaften die Anforderungen definiert (Kapitel 3.3). Diese sind unterteilt in funktionale Anforderungen (Kapitel 3.3.1) und nicht-funktionale Anforderungen (Kapitel 3.3.2).
Die Anforderungsanalyse eines Systems ist maßgeblich an der Qualität eines Projekts beteiligt. Hierbei werden die Aufgaben und die grundsätzlichen Eigenschaften, die das System annehmen und umsetzen muss, definiert. Dies geschieht mittels Analyse von Arbeitsabläufen und grundsätzlichen Gegebenheiten des Systems. Das Ziel dieser Bachelor-Arbeit ist es, für den *proCollab* Prototypen eine Persistenzschicht zu konzipieren und prototypisch zu implementieren.

3.1. Anwendungsszenario

Im folgenden wird ein Anwendungsszenario des proCollab Prototypen aufgezeigt.

Ein global agierendes Softwareunternehmen verwendet proCollab für die Checklistenverwaltung seiner Softwareprojekte. Die Mitarbeiter bearbeiten ausschließlich die Daten des lokalen Standortes, an dem sie tätig sind. Da CLs oft gelesen und geändert, jedoch nur selten erstellt werden, sind die Anfragen an das DBS größtenteils lesend und ändernd. Aufgrund der projektorientierten Arbeit variiert die Anzahl der Anfragen stark, da außerhalb von Projekten keine CLs bearbeitet werden.

3.2. Anwendererzählungen

Für die Anforderungsanalyse des Gesamtprojekts wurden Anwendererzählungen erstellt, die Vorgänge der Interaktion des Nutzers mit dem System beschreiben. Diese haben den Zweck, die Definition der funktionalen und nicht-funktionalen Anforderungen zu unterstützen. Indem konkrete Bedien- und Bearbeitungsvorgänge des Nutzers mit dem Gesamtsystem beschrieben werden, kann auf die benötigte Funktionalität geschlossen werden.
Für den *proCollab* Prototypen lassen sich die Userstories in fünf Bereiche gliedern; Benutzerverwaltung, organisatorische Rahmenverwaltung, Checklistenverwaltung, Checklisteneintragsverwaltung und der übergreifenden Verwaltung. Im Folgenden werden für jeden Bereich zwei repräsentative Userstories vorgestellt- die restlichen Anwendererzählungen befinden sich im Anhang A.

Benutzerverwaltung

Dieser Bereich beschreibt Anwendererzählungen, die Vorgänge schildern, bei denen mit Nutzerdaten interagiert wird.

Benutzer anlegen
Ein Administrator kann einen Nutzer in der administrativen Umgebung, am System

registrieren. Dazu kann er den Vorname, Nachnamen, Passwort, E-Mail Adresse und die Firmenzugehörigkeit des Nutzers angeben. Die E-Mail-Adresse wird validiert und die Passworteingabe muss zweimalig erfolgen. Nach dem Bestätigen wird dem Nutzer eine Aktivierungs-E-Mail an sein E-Mail Konto gesendet. Nach der Aktivierung erhält der Nutzer eine weitere E-Mail, mit all seinen Nutzerangaben. Der Ablauf wird in Abbildung 3.1 dargestellt.

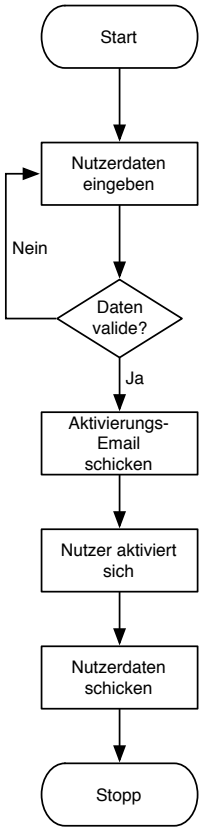


Abbildung 3.1.: Benutzer anlegen

Einfache Anmeldung

Ein Nutzer gibt seine E-Mail Adresse und Passwort ein. Anschließend werden seine Eingaben vom Server überprüft. Bei einer Bestätigung wird ihm das Backend angezeigt, bis er sich vom System abmeldet. Bei Ablehnung, kann er erneut seine Eingaben tätigen. Der Ablauf wird in Abbildung 3.2 dargestellt.

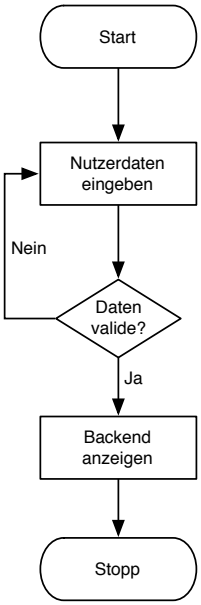


Abbildung 3.2.: Benutzer anlegen

Organisatorische Rahmenverwaltung

Vorgänge, die die Verwaltung der ORT und ORI beschreiben, werden in diesem Bereich angegeben.

Anpassen von Details eines ORT
Nach der Auswahl eines ORT, kann ein Administrator die Details dessen anpassen. Er kann den Namen, Ziel, Startdatum, Enddatum und die Bearbeitungszeit verändern.

Entfernen einer ORI
Nach dem Suchen und Auswahl einer ORI werden dem Administrator die Details angezeigt. Hier kann er mit Klick den Rahmen aus dem System entfernen. Anschließend werden alle Verbindungen zu existierenden ORIs gelöscht.

Checklistenerwaltung

Im folgenden werden Vorgänge zur Verwaltung der CLTs und CLIs dargestellt.

CLT suchen
Ein Administrator kann eine Listenvorlage mit der Suchfunktion, anhand der Attribute Namen und Beschreibung suchen. Wenn eine oder mehr Listenvorlagen den Vorgaben entsprechen wird eine Liste der Treffer angezeigt. Wird einer der Treffer in der Liste angeklickt, werden die Details des CLT angezeigt. Wird keine Listenvorlage zu den Vorgaben gefunden, wird eine Nachricht angezeigt.

Erstellen einer CLI
Ein Administrator kann eine neue Listeninstanz erstellen. Hierzu kann ein Name, eine Beschreibung und ein Status angegeben werden. Anschließend kann der Administrator CEIs hinzufügen und Nutzer mit ihren zugehörigen Rollen angeben.

Checklisteneintragsverwaltung

Für die Verwaltung von CEIs und CETs relevante Vorgänge werden in diesem Abschnitt abgebildet.

Entfernen einer CET
Nach dem Suchen und Auswahl einer CET kann ein Administrator die mit einem Klick auf einen Knopf, entfernen.

Anpassen einer CEI
Nach dem Suchen und Auswahl einer CEI kann ein Administrator diese anpassen. Er kann den Namen, den Text und den Status ändern.

Übergreifende Verwaltung

Vorgänge, die mehrere Bereiche überschneiden und sich somit nicht explizit zu einem Bereich zuordnen lassen, werden hier geschildert.

Hinzufügen eines Benutzers zu einer ORI
Nachdem ein Benutzer gesucht wurde, kann ein Verwalter den Benutzer zu einer ORI des Verwalters hinzufügen und die Rolle des Benutzers anpassen.

Einen Benutzer von einer ORI entfernen
Während eine ORI geöffnet ist, kann ein Verwalter einen Benutzer von dieser entfernen. Dies muss durch einen weiteren Dialog vom Verwalter bestätigt werden.

3.3. Funktionale und Nicht-Funktionale Anforderungen

Das Hauptaugenmerk der Konzipierung der Persistenzschicht des proCollab Prototypen liegt darin, hierarchische Daten zu speichern, da sowohl ORs als auch CLs sich als hierarchische Datenstruktur, hier als Baum, darstellen lassen. Hieraus ergeben sich in Kombination aus den Grundoperationen für Bäume mit der Anforderung, Daten persistent zu speichern, die funktionalen Anforderungen. Funktionale Anforderungen beschreiben die Aufgaben des Systems. Nicht-funktionale Anforderungen beschreiben im Gegensatz dazu Funktionalitäten, die das Bedienen des Systems erleichtern bzw. welche Eigenschaften das System hat.

3.3.1. Funktionale Anforderungen

Die Hauptaufgaben einer Persistenzschicht lassen sich am besten mittels der sogenannten SCRUD-Grundoperationen beschreiben. CRUD bedeutet *Create (Erstellen), Read (Auslesen), Update (Ändern), Delete (Löschen)* und beschreibt im Allgemeinen die grundsätzlichen Funktionen eines Datenverwaltungs-System. Dies lässt sich anhand der in Kapitel 3.2 beschriebenen Userstories bestätigen. Ein Teil der funktionalen Anforderungen im Zusammenhang mit einem Datenverwaltungs-System wären somit das Umsetzten der Operationen für **Erstellen**,**Lesen**,**Ändern** und **Löschen** von Daten, wobei Lesen unterschieden muss nach Lesen eines einzelnen Elementes und Lesen spezifischer Elemente. Neben diesen generellen Anforderungen an die Datenverwaltung sollte darauf geachtet werden, dass **Schnittstellen-Spezifikationen** eingehalten werden.
Eine Schnittstellen-Spezifikation ist eine Definition, wie eine Schnittstelle eines Systems heißt, welche Werte sie entgegen nimmt und zurückgibt bzw. was das System nach

Ansprechen jener ausführt. Das Erstellen und Einhalten dieser Spezifikation erhöht die Modularität des Systems, da andere Komponenten über die definierten Schnittstellen mit dem Restsystem kommunizieren können und auch die Funktionen der Schnittstellen den Entwicklern der neuen Komponente klar sind.

3.3.2. Nicht-Funktionale Anforderungen

Neben den funktionalen Anforderungen, welche auf jeden Fall eingehalten und umgesetzt werden müssen, gibt es noch nicht-funktionale Anforderungen. Diese können umgesetzt werden, sind jedoch nicht für das Funktionieren der Anwendung zwingend nötig.

Geringe Antwortzeiten sowohl für Lese- als auch Schreiboperationen sind von Vorteil, da dies den Arbeitsfluss des Anwenders deutlich verbessert. Dies könnte bei höheren Antwortzeiten zum Problem werden, da der Anwender sehr oft diese Operationen in kurzer Zeit ausführt. Außerdem wäre es nützlich, wenn das System **Caching** oder **Lazy Loading** anbietet. Eine weitere Eigenschaft, die lediglich den Bedienkomfort für den Nutzer steigert, ist die Konfiguration und **Anpassung des Systems** an die Laufzeitumgebung des Nutzers so einfach wie möglich zu gestalten, um eine schnelle Inbetriebnahme des Systems für den Nutzer zu ermöglichen. Dies kann z. B. durch Bereitstellen von Schnittstellen zur Konfiguration erreicht werden. Um den proCollab Prototyp für Szenarien wie in Kapitel 3.1 geschildert zu verwenden, sollte auf eine **verteilte Datenbank** zurückgegriffen werden. So bleiben die Antwortzeiten des DBS trotz unterschiedlicher Standorte gering.

4

Konzept

In Kapitel 4 wird die Konzeption der Persistenzschicht vorgestellt, wobei zuerst die zugrundeliegenden Paradigmen erklärt werden (siehe Kapitel 4.1) und das Datenmodell beschrieben wird (siehe Kapitel 4.2). Danach wird basierend auf den diskutierten Verteilungsarten (siehe Kapitel 4.3) und den Möglichkeiten, die Daten des Datenmodells zu speichern (siehe Kapitel 4.3.2 und Kapitel 4.3.3), die konzipierten Architekturen vorgestellt (siehe Kapitel 4.3.4). Im Anschluss daran werden die Schnittstellen der Persistenzschicht definiert (siehe Kapitel 4.4)

4.1. Programmierparadigmen

4.1.1. Model-View-Controller Muster

Grundsätzlich ist der *proCollab* Prototyp nach dem sogenannten *Model-View-Controller* (MVC) Muster aufgebaut. Darunter versteht man die Aufteilung eines Informationssystems in die drei Komponenten *Model* (dt. Modell), *View* (dt. Ansicht) und *Controller* (dt. Steuerung). Diese drei Komponenten stehen in einem zirkulären Zusammenschluss (Abbildung 4.1). Eingaben werden von der **View** entgegengenommen und an den **Controller** weitergeleitet. Dieser aktualisiert die im **Model** gespeicherten Daten [KP88].

Model Diese Komponente hält die Daten des Systems. Neben diesen Systemdaten kann das *Model* auch Logik enthalten, die zur Datenhaltung benötigt wird.

View Diese Komponente ist zuständig für die Anzeige der Daten des *Models*. Des weiteren nimmt die *View* die Nutzereingaben entgegen und leitet diese an die *Controller*-Komponente weiter.

Controller Die *Controller*-Komponente bekommt die Nutzereingaben der *View* übergeben. Diese Nutzereingaben werden verarbeitet und abhängig von den verarbeiteten Eingaben werden die im *Model* gespeicherten Daten geändert.

Die Komponenten des proCollab Prototypen (siehe Kapitel 2.2) lassen sich auf die Komponenten des MVC-Musters übertragen. Hierbei entspricht die Persistenzschicht dem Model, die Serverkomponente dem Controller und die verschiedenen Benutzeroberflächen der View.

4.1.2. Platform as a Service

Platform as a Service (PaaS) ist eine spezielle Form eines Cloud-Dienstes. Er bietet Serverstrukturen und die damit verbundenen Systemressourcen, wie Festplattenspeicher und Prozessoren an, auf denen von einem Nutzer entwickelte Programme ausgeführt werden können. Da die verwendeten Ressourcen komplett von dem PaaS-Dienst verwaltet werden, übernimmt dieser auch die Skalierbarkeit des Programms. Die vom

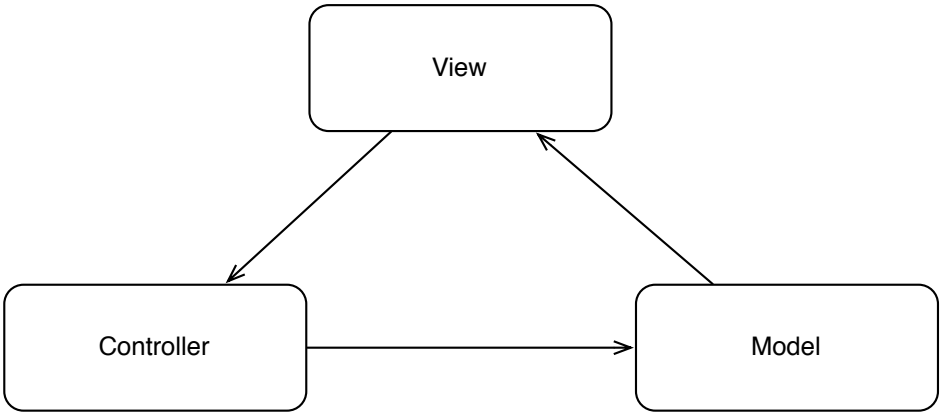


Abbildung 4.1.: Model-View-Controller Muster

Programm verwendeten Programmiersprachen und Zusatzprogramme muss der Dienst jedoch unterstützen.

Die Basis für die von PaaS angebotenen Serverstrukturen bilden *Anwendungsserver*. Ein Anwendungsserver dient als Rahmen für darauf ausgeführte Programme und besitzt Komponenten, die von den Programmen benötigte Funktionalitäten unterstützen. Zu diesen Funktionalitäten gehört beispielsweise das Verwalten der Transaktionen für DB-Anfragen. Die Funktionalitäten des Anwendungsservers können jedoch nur mit den von dem Anwendungsserver abhängigen Programmiersprachen angesprochen werden.

Der proCollab Prototyp kann, für das Umsetzen von Skalierbarkeit, auf einem Server eines PaaS-Dienstes eingepflegt werden. Dadurch werden dynamisch weitere Knoten, auf denen automatisch der proCollab Prototyp eingepflegt wird, zu dem Gesamtsystem hinzugefügt. Es muss jedoch sichergestellt werden, dass der PaaS-Dienst sowohl die benötigte DB-Art als auch die verwendete Programmiersprache unterstützt.

4.2. Datenmodell

Das konzeptionelle Datenmodell spezifiziert, in welcher Struktur die Daten des proCollab Prototypen aufgebaut sind. Das Datenmodell von *proCollab* lässt sich in hierarchisch strukturierte (siehe Abbildung 4.2) und in nicht-hierarchisch strukturierte Entitäten (siehe

Abbildung 4.3) unterteilen. In Abbildung 4.2 und Abbildung 4.3 sind die verschiedenen Entitäten vereinfacht dargestellt, um die Hierarchie der Daten untereinander deutlich zu machen. Die Relationen zwischen den Entitäten, die genauen Attribute und weitere Spezifikationen wie *«enumeration»* bei Aufzählungen sind im Anhang A.2 beschrieben.

Die hierarchischen Entitäten (siehe Abbildung 4.2) sind, wie in Kapitel 2.4 beschrieben, unterteilt in Instanzen und Typen. Ein OR kann sowohl mehrere CLs als auch weitere ORs enthalten. CLs können wiederum mehrere untergeordnete CLs und CEs besitzen, wobei CEs keine untergeordneten Elemente mehr enthalten.

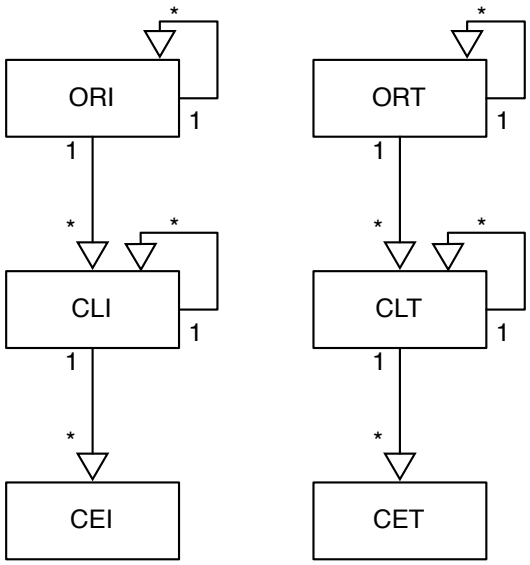


Abbildung 4.2.: Hierarchie Checkliste und Organisatorischer Rahmen

Die nicht-hierarchisch strukturierten Entitäten (siehe Abblidung 4.3) besitzen Relationen zu anderen Daten, lassen sich jedoch nicht in eine hierarchische Struktur ordnen. Sie dienen der Speicherung der Daten, die für die Verwaltung der CLs benötigt werden, beispielsweise welcher Nutzer einen bestimmten OR erstellt hat oder welche Rolle er im OR einnimmt.

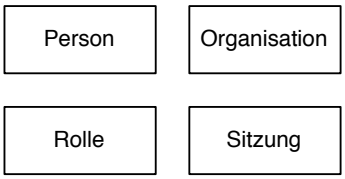


Abbildung 4.3.: Nicht-hierarchische Daten

4.3. Konzeption der Persistenzschicht

4.3.1. Vergleich von Konzepten

Daten verteilt in einem Datenbanksystem zu speichern, kann auf verschiedenste Arten bewerkstelligt werden. Jedoch eignet sich nicht jeder Ansatz für jedes Projekt bzw. für jedes System gleich gut. Somit muss im Hinblick auf die Anforderungsanalyse (siehe Kapitel 3.3) der am besten passende Ansatz für die Persistenzschicht gewählt werden. Ein generell eher simpler Ansatz, Daten verteilt zu speichern, wäre alle Daten auf allen DB-Knoten zu replizieren. Das Verteilen der Daten kann von der aufliegenden Verwaltungsschicht, oft Teil eines DBS, übernommen werden. Das Anbinden neuer DBS-Knoten an diese Architektur ist einfach, da nur ein bereits bestehender DBS-Knoten gespiegelt werden muss. Die Ausfallsicherheit ist ebenfalls sehr gut, da beim Ausfall eines oder mehrerer DBS-Knoten andere die Last übernehmen können und bei Reaktivierung der ausgefallenen DBS-Knoten diese einfach von den anderen gespiegelt werden. Dieser Ansatz eignet sich jedoch für die Anforderungen dieses Systems nicht. Bei komplett replizierten Daten wächst die Antwortzeit bei zunehmender örtlicher Verteilung, da Datenänderungen an alle DB-Knoten propagiert werden [NCWD84].

Ein weiterer Ansatz ist, Daten mittels Sharding (siehe Kapitel 2.5.2) standortabhängig zu verteilen. Dadurch wird die Anfragezeit für standortspezifische Daten verringert, da die Daten lokal vorliegen. Betrachtet man das in Kapitel 3.1 vorgestellte Anwendungsszenario, ist dieser Ansatz geeignet für den proCollab Prototypen.

4.3.2. Persistierung hierarchischer Daten

Den größten Anteil der Daten für den *proCollab* Prototypen nehmen CLs und CEs ein. Bei diesen CLs, welche auch u.a. untergeordnete CLs enthalten können, handelt es sich um hierarchische Daten (Bäume). Da das Problem der Speicherung hierarchischer Daten in Datenbanken schon seit langem besteht, wurden viele verschiedene Möglichkeiten entwickelt, diese Daten zu speichern. Jedoch besitzt jede dieser Möglichkeiten ihre Vor- und Nachteile und ist dadurch mehr oder weniger für die Umsetzung des Systems geeignet. Die Auswahl eines zu verwendenden DBS muss immer in Betrachtung der Anforderungen des Systems geschehen. Eine Anforderung an das DBS ist, *ACID* zu unterstützen, da durch das Einhalten von *ACID* Inkonsistenz der Daten vermieden wird. Bei NoSQL-Datenbanken ist dies nicht der Fall. Diese setzen das *BASE*-Modell (siehe Kapitel 2.5.2) um [Pri08]. Bei diesem Modell liegt das Hauptaugenmerk auf Skalierbarkeit und Performanz anstatt wie bei *ACID* auf Konsistenz. Da Datenkonsistenz für die Daten des proCollab Prototypen wichtiger ist als Performanz, werden NoSQL-Datenbanken nicht näher für die Systemkonzeption in Betracht gezogen.
Graph-Datenbanken sind durch ihre Natur sehr geeignet für die Speicherung hierarchischer Daten. Des weiteren sind sie hochskalierbar und sehr performant, vorausgesetzt der ist Baum tief. Da die horizontale Partitionierung bei graphenbasierten Datenbanken nicht angeboten wird, wird diese Art der DBSs nicht verwendet. Aufgrund dessen wird bei diesem System eine relationale Datenbank gewählt.

Hierarchische Daten in einer relationalen Datenbank zu speichern, lässt sich auf verschiedene Art und Weise umsetzen. Hierbei wird typischerweise der Baum in eine relationale Struktur transformiert. Im folgenden sollen bekannte Verfahren verglichen werden. Als Modell dient ein Stammbaum, wobei B und C Kinder von A und D und E Kinder von C sind (siehe Abbildung 4.4). Für diesen Stammbaum werden die Tabellenstrukturen der verschieden Verfahren dargestellt.

Adjazenzlisten Bei *Adjazenzlisten* handelt es sich um die einfachste Möglichkeit, hierarchische Daten in eine relationale Datenbank zu speichern [Cel04]. Es wird in die Tabelle, welche zur Speicherung dieser Daten genutzt wird, eine Spalte mit einem Fremdschlüssel auf das Elternelement eingefügt. Somit wird für jedes Element

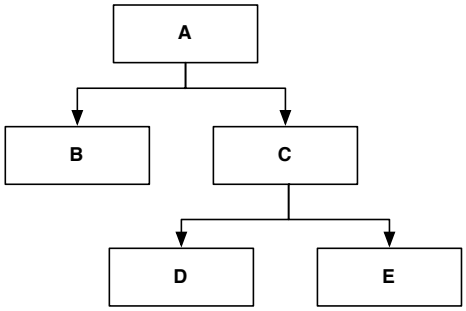


Abbildung 4.4.: Modell eines Stammbaum

des Baums das entsprechende Elternelement gespeichert. Die Performanz beim Finden eines oder mehrerer Vorfahren oder bei Speicher- und Änder-Operationen ist sehr gut. Jedoch ist es sehr aufwendig, Elemente bis zu einer bestimmten Tiefe oder einen bestimmten Pfad zwischen Elementen zu finden, sofern der Pfad oder die Tiefe nicht in einer extra Spalte gespeichert ist. [Adj09]. In Tabelle 4.1 wird eine Adjazenzlistenstruktur für den in Abbildung 4.4 gezeigten Stammbaum dargestellt. NULL bedeutet, dass der Knoten keinen übergeordneten Knoten besitzt.

Knoten	Vater
A	NULL
B	A
C	A
D	C
E	C

Tabelle 4.1.: Adjazenzliste des Stammbaums

Nested Sets Nested Sets, auch Modified Preorder Tree Traversal genannt, bildet hierarchische Daten als verschachtelte Mengen ab. Es werden hierbei für die Position eines Elements zwei Werte gespeichert, *links* und *rechts* [Cel04]. Die links- und rechts-Werte aller Kind-Knoten eines Elements befinden sich zwischen dem links und rechts des Elements. So hat das erste Kind den Wert links+1 und das letzte Kind den Wert rechts-1 des Elements. Durch diese Struktur ist das Traversieren des Baums, das Bestimmen eines Pfades oder die Ermittlung der Anzahl aller Knoten sehr einfach, jedoch sind Änderungen wie das Tauschen zweier Knoten

sehr aufwendig, da links und rechts des Elements und aller Kind-Elemente geändert werden müssen [Cel04]. Die Tabelle 4.2 beschreibt, wie die Elemente des Stammbaums (siehe Abbildung 4.4) bei Nested Sets in eine Tabelle gespeichert werden.

Knoten	links	rechts
A	1	10
B	2	3
C	4	9
D	5	6
E	7	8

Tabelle 4.2.: Nested Sets Tabelle des Stammbaums

Materialized Path Bei der *Materialized Path*-Methode wird der Pfad zu dem entsprechenden Knoten als Text in einer Spalte der Tabelle gespeichert [Tro05]. Wenn z.B. ein Knoten den Pfad 1 besitzt, haben alle Kinder dieses Knotens den Pfad 1.1, 1.2, 1.3 usw, wie es in Tabelle 4.3 für den in Abbildung 4.4 dargestellten Stammbaum zu sehen ist. Jedoch kann bei dieser Methode verschiedene Pfadnummerierungen oder Trennzeichen zwischen den Pfadelementen gewählt werden. Aufgrund dieser Struktur ist die Geschwindigkeit der Traversion oder der Ermittlung von Nachfolgern und Vorfahren nicht von der Datenbank an sich, sondern von der Funktion, die den Pfad-Text aufspaltet und verarbeitet, abhängig.

Knoten	Pfad
A	1
B	1.1
C	1.2
D	1.2.1
E	1.2.2

Tabelle 4.3.: Materialized Path Tabelle des Stammbaums

Die CLs von *proCollab* werden häufig geändert und es werden oft Elemente entfernt oder hinzugefügt. Da Tabellen mit Adjazenzlistenstruktur schnelles Ändern und Einfügen von Daten erlaubt, wurde diese Struktur ausgewählt.

4.3.3. Persistierung relationaler Daten

Neben hierarchischen Daten gibt es im proCollab Prototypen auch nicht-hierarchische Daten. Hierunter fallen Daten wie Nutzerdaten oder organisatorische Daten, wie sie z.B. für die Rechte- oder Sitzungsverwaltung im System benötigt werden. Diese Daten können, wie hierarchische Daten, in Relation zueinander stehen. Da dies ebenfalls Herausforderungen wie das Abbilden der Relationen, mit sich bringt, eignen sich verschiedene Datenbank-Arten unterschiedlich gut zur Speicherung. NoSQL-Datenbanken sind aufgrund der Schlüssel-Wert-Struktur eher ungeeignet für die Speicherung dieser Daten, da hier ebenfalls Konsistenz und Verfügbarkeit immens wichtig sind. Beim Vergleich zwischen Graph-Datenbanken und relationalen Datenbanken im Bezug auf nicht-hierarchische Daten bestehen die gleichen Probleme wie bei hierarchischen Daten. Somit wird bei nicht-hierarchischen Daten ebenfalls eine relationale Datenbank verwendet.

4.3.4. Systemarchitektur

Bei einem verteilten System muss im Bezug auf das genutzte DBS und die Persistenz-Schicht sowohl auf die Kooperation der einzelnen System-Knoten (siehe Kapitel 4.3) als auch auf die interne Architektur jedes Knotens (siehe Kapitel 3.3) geachtet werden.

Gesamtsystem

Bei der Konzeption der Gesamt-Architektur (siehe Abbildung 4.3.4) liegt der Fokus auf Verteilbarkeit des DBS, damit die in Kapitel 3.3.2 gestellte Anforderung an eine verteilte Datenbank erfüllt wird. Für die Bereitstellung und Verwendung eines Caches ist die Verteilung des DBS ebenfalls zuträglich, da das Caching der Daten nicht nur von einem einzigen Cache durchgeführt werden muss, sondern von mehreren kleinen Caches, welche für die jeweils darunterliegende DB das Caching übernehmen. Um Anfragen auf standortspezifische Daten, wie in Kapitel 3.1 beschrieben, zu verbessern, wird Sharding (siehe Kapitel 2.5.2) eingesetzt, wodurch diese Datensätze nach Standorten aufgeteilt werden.

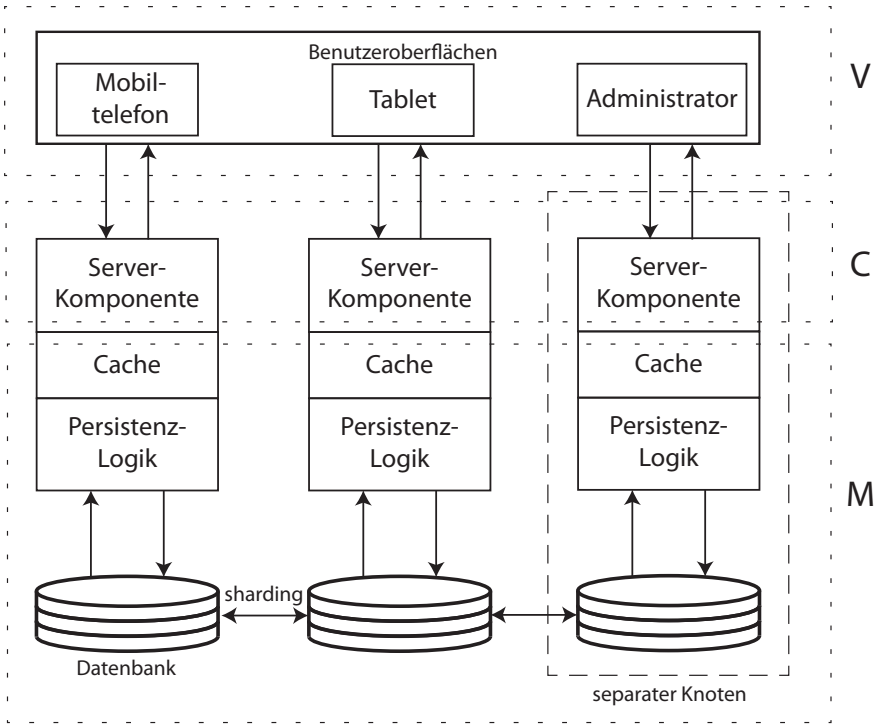


Abbildung 4.5.: Gesamt-Architektur

Persistenzschicht

Die Persistenzschicht besteht aus einem Cache, der Systemlogik und einer Datenbank-Anbindung (siehe Abbildung 4.6). Anfragen der Server-Komponente an die Persistenz-Schicht werden zuerst an den Cache geleitet, der überprüft, ob der angeforderte Datensatz bereits vorliegt. Ist dies nicht der Fall, wird die Anfrage an die Systemlogik weitergeleitet, welche daraufhin die passende Datenbank-Anfrage identifiziert und mittels der Datenbank-Anbindung eine Verbindung zur Datenbank herstellt.

Im folgenden wird der entsprechende Datensatz angefordert und nach Einlagern im Cache an die Server-Komponente übermittelt. Als Verdrängungsstrategie des Caches wird die *Least-Recently-Used*-Strategie verwendet. Dadurch wird, wenn zu wenig Spei-

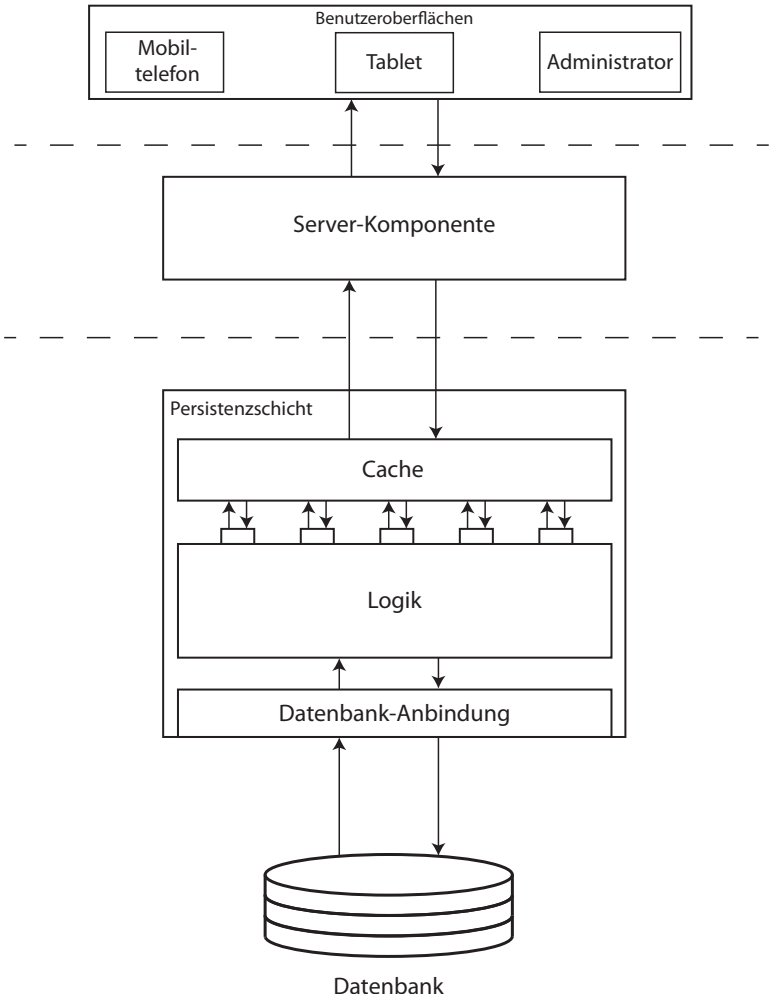


Abbildung 4.6.: Architektur der Persistenzschicht

cherplatz im Cache ist, der am längsten nicht verwendete Datensatz durch den neu einzulagernden Datensatz ersetzt.

4.4. Schnittstellen

Für die Kommunikation der Server-Komponente mit der Persistenz-Schicht sind bestimmte Schnittstellen erforderlich. Welche Schnittstellen und somit welche Funktionen angeboten werden, lässt sich aus den in Kapitel 3.3 definierten Anforderungen und dem vorhandenen Datenmodell (Kapitel 4.2) ermitteln. Für jede Entität müssen die CRUD-Operationen (*Erstellen, Auslesen, Ändern, Löschen*) umgesetzt werden, wobei es Varianten des Auslesens gibt. Es wird unterschieden zwischen dem Auslesen eines einzelnen Elementes, aller Elemente bzw. auf bestimmte Suchkriterien zutreffende Elemente einer Entität. Für die Sitzungsentität wird nur eine Schnittstelle für das Auslesen einzelner Elemente benötigt, da für die Sitzungsentität nur einzelne Elemente angefordert werden. Der Rückgabewert der Schnittstellen, die die Ausleseoperationen umsetzen, ist immer die Menge der angeforderten Elemente. Für die Schnittstellen der Änderungs-, Lösch- und Erstellungsoperationen werden bei erfolgreichem Durchführen der Operationen Bestätigungen zurückgegeben. Die Schnittstelle für das Auslesen aller OR würde beispielsweise eine Liste aller in der DB gespeicherten OR zurückgeben. In folgender Tabelle wird dargestellt, welche Grundfunktionen für die jeweiligen Entitäten umgesetzt werden. Ein "X"bedeutet, dass die Funktion umgesetzt, ein "-", dass sie nicht umgesetzt werden muss.

	OR	CL	CE	Person	Organisation	Sitzung
Schreiben	X	X	X	X	X	X
Lesen (einzeln)	X	X	X	X	X	X
Lesen (alle)	X	X	X	X	X	-
Lesen (spezifisch)	X	X	X	X	X	-
Ändern	X	X	X	X	X	X
Löschen	X	X	X	X	X	X

Tabelle 4.4.: Umzusetzende Funktionen der einzelnen Daten

5

Implementierung

In Kapitel 5 wird die prototypische Entwicklung der Persistenzschicht des proCollab Prototypen vorgestellt. Hierfür werden zuerst die verwendeten Technologien erklärt (siehe Kapitel 5.1), wobei zwei Datenbankverbindungen verglichen werden (siehe Kapitel 5.1.7), danach die Prototypen-Architektur mit Ablauf beschrieben (siehe Kapitel 5.2), und zuletzt Auszüge der Implementierung dargestellt (siehe Kapitel 5.3.1).

5.1. Technologien

Im folgenden werden die Technologien erklärt, welche bei der Implementierung der Persistenzschicht betrachtet wurden.

5.1.1. Java Persistence API

Die *Java Persistence API* (JPA) ist eine Spezifikation für die *objektrelationale Abbildung* (ORM) für Java. Unter ORM versteht man das automatisierte Abbilden von Objekten aus einer objektorientierten Programmiersprache in ein relationales DBS. Hierbei werden aus den Attributen der Klassen die Tabellenspalten und aus den Beziehungen zwischen Klassen Relationstabellen bzw. Fremdschlüssel erstellt. Bei JPA im Speziellen werden die für die Spezifizierung von Tabellen und Relationen benötigten Meta-Daten über spezielle XML-Dateien oder über Java-Annotationen direkt im Programm-Code definiert. Des weiteren können zur Laufzeit Objekte aus Datensätzen generiert oder vom System erstellte Objekte gespeichert werden. JPA unterstützt Anfragen in sowohl einer eigenen Datenbanksprache *Java Persistence Query Language*(JPQL) als auch in SQL.

JPA EntityManager

Objekte werden unter JPA durch den EntityManager (EM) verwaltet. Der EM ist eine JPA-Klasse, welche zur Kontext-Verwaltung der Objekte dient. Unter Kontext im Bezug auf JPA versteht man die Verwaltung des Lebenszyklus eines Objekts, das mit dem jeweiligen Datensatz verknüpft ist. Bei Auslesen eines Datensatzes wird ein neues Objekt mit den von dem DBS erhaltenen Daten erstellt. Dieses Objekt wird daraufhin von dem vorhandenen EM mit dem Datensatz, aus dem es erstellt wurde, verknüpft und in den Kontext des EM eingefügt. Solange sich dieses Objekt im Kontext des aktuellen EM befindet, werden bei Änderungen am Objekt diese ebenfalls an dem entsprechende Datensatz durchgeführt. Bei Schließen des EM werden alle Verbindungen von Objekten zu dem DBS, welche sich im Kontext des jeweiligen EM befanden, geschlossen. Wenn bestehende Objekte wieder verknüpft werden sollen, müssen diese mit einem neuen EM verbunden werden. Dies kann über die Methode `refresh()` geschehen, wodurch die Daten des Objekts von den in dem DBS enthaltenen Daten überschrieben werden. Sollten jedoch Änderungen des Objekts in das DBS übernommen werden, muss das Objekt über den `merge(text)`-Befehl mit dem EM verbunden werden.
Bei der *EntityManagerFactory* (EMF) handelt es sich um eine von JPA bereitgestellte *Factory* für die Erstellung und Verwaltung der EM. Unter einer Factory versteht man ein

Pattern, das eine Schnittstelle für die Erstellung bestimmter Objekte beschreibt. Da die Erzeugung einer EMF unter Geschwindigkeitsaspekten und Ressourcen-technisch eine sehr teure Operation ist, sollte diese ein einziges Mal beim Start der Anwendung erzeugt werden. Jedoch müssen für die Erzeugung einer EMF Parameter wie die URL des DBS und die Zugangsdaten für diese definiert sein. Dies wird in einer Konfigurationsdatei vorgenommen. Hier werden die Parameter in *Persistence-Einheiten* (PU) eingeteilt. Jede PU kann für sich eigene Parameter wie das Cache-Verhalten, die Login-Daten oder welche Klassen zu persistieren sind enthalten. Bei Erstellen der EMF wird angegeben, welche PU gewählt werden soll. Die in der PU festgelegten Parameter werden an die EMF weitergegeben, die anhand der Parameter eine Datenbankverbindung herstellt und, wenn nötig, die Datenbanktabellen erstellt bzw. ändert.

Vererbung bei JPA

Das Datenmodell, das dem proCollab Prototypen zugrunde liegt, setzt die verschiedenen ORs, CLs und CEs als hierarchische Klassen mit zweifacher Vererbung um (siehe Anhang A.2). Dies bedeutet, dass eine untergeordnete Klasse von einer Klasse erbt, die ebenfalls die untergeordnete Klasse einer anderen Klasse ist. Vererbung bei JPA muss jedoch explizit annotiert werden. Um eine Klasse generell als persistierbare Klasse zu markieren, muss über der Klassendeklaration die Annotation `@Entity`, zu sehen in Listing 5.4, Zeile 1, geschrieben werden. Des weiteren muss innerhalb der Persistenz-Einheiten der Klassenpfad in `<class>`-Tags geschrieben werden. Dadurch wird beim Erstellen der Datenbank eine entsprechende Tabelle für die Klasse angelegt. Vererbung bei Objekten kann über mehrere Arten durch Annotationen dargestellt werden.

@MappedSuperclass Sollte eine Klasse keine zu persistierende Klasse sein, dennoch in der untergeordneten Klasse Attribute aus der übergeordneten Klasse benötigt werden, kann die übergeordnete Klasse mit `@MappedSuperclass` annotiert werden. Dies kann für mehrfache Vererbung eingesetzt werden, jedoch muss die letzte untergeordnete Klasse mit `@Inheritance` annotiert werden. Damit wird diese Klasse zur Wurzel der Vererbungshierarchie gemacht, da eine `@MappedSuperclass` nicht das Wurzelelement sein darf.

@Inheritance(strategy=InheritanceType.TABLE_PER_CLASS) Hierbei wird für jede nicht abstrakte Klasse der Hierarchie eine Tabelle erstellt. Die Umsetzung ist sehr simpel, jedoch müssen alle Relationen bidirektional sein. Des weiteren muss die ID über alle Tabellen der Hierarchie eindeutig sein.

@Inheritance(strategy=InheritanceType.SINGLE_TABLE) Bei `SINGLE_TABLE` werden alle in der Hierarchie enthaltenen Klassen in eine einzelne Tabelle übertragen. Für die Unterscheidung der verschiedenen Klassen wird eine sogenannte Discriminator-Spalte eingefügt. Zugleich erhält jede Klasse der Hierarchie einen Unterscheidungs-Wert der standardmäßig der Name der Klasse ist. Will man diesen ändern, muss die entsprechende Klasse mit `@DiscriminatorValue` annotiert werden. Darin wird der neue Unterscheidungs-Wert angegeben, der letzten Endes für diese bestimmte Klasse in die Discriminator-Spalte eingetragen wird. Um diese Strategie umzusetzen, müssen die Spalten der untergeordneten Klassen es erlauben, keine Werte einzutragen. Man muss beachten, dass das Ändern des Datenmodells sich auf alle in der Tabelle enthaltenen Datensätze auswirkt, da Spalten geändert, gelöscht oder hinzugefügt werden müssen.

@Inheritance(strategy=InheritanceType.JOINED) Diese Strategie, auch Table-per-Subclass genannt, ruft Joins auf den untergeordneten Klassen mit der übergeordneten Klasse auf. Für das Ändern der Join-Konditionen muss die untergeordnete Klasse mit `@PrimaryKeyJoinColumn()` annotiert werden. Wird dies nicht angegeben, werden die IDs der beiden per JOIN zu vereinenden Tabellen verglichen. Der Nachteil dieser Strategie ist, dass beim Auslesen der Objekte bei tiefen Hierarchien sehr viele Joins gemacht werden, was sich negativ auf die Performance auswirkt.

5.1.2. Java Database Connectivity

Unter *Java Database Connectivity* (JDBC) versteht man die Datenbankschnittstelle von Java [Ora13]. Diese stellt über einen Datenbank-spezifischen Treiber, Connector genannt, eine Verbindung mit dem darunterliegenden DBS her. Nach der Verbindungs-

herstellung werden die im Programmcode eingebetteten Datenbankabfragen an das DBS weitergeleitet, dort verarbeitet und die Resultate anschließend entgegen genommen.

5.1.3. Hibernate

Hibernate ist ein Open-Source Framework für JPA oder .Net, welches zur Umsetzung von Datenpersistenz über ORM dient. Es unterstützt neben Lazy Loading (siehe Kapitel 2.5.1) auch Sharding (siehe Kapitel 2.5.2) und eine eigene Datenbanksprache, Hibernate Query Language, welche eine Spezifikation der JPQL ist.

Relationen in Hibernate

In Hibernate können über Annotationen Relationen vom Typ *1-zu-1*, *1-zu-n* oder *n-zu-m* beschrieben werden. 1-zu-1 mit der Annotation `@OneToOne` dient der Relation zwischen einzelnen Objekten, 1-zu-n mit den Annotationen `@OneToMany/@ManyToOne` ist für Relationen zwischen einem einzelnen Objekt und einer Liste von Objekten und n-zu-m mit der Annotation `@ManyToMany` für Relationen zwischen Listen. Hierbei wird die entsprechende Annotation entweder über das jeweilige Attribut oder die `get()`-Methode des Attributs geschrieben. Bei einer 1-zu-n Relation muss jedoch darauf geachtet werden, dass die Liste mit `@OneToMany` und das einzelne Objekt mit `@ManyToOne` annotiert wird. Bei Hibernate können 1-zu-n Relationen über zwei Möglichkeiten dargestellt werden. Der Relationspartner kann entweder bei der Liste (n) mit dem zusätzlichen Wert `mappedby` und dem Namen des Relationspartners oder bei dem einzelnen Objekt (1) über die Annotation `@JoinColumn` und der ID des Relationspartners angegeben werden.

5.1.4. Hazelcast

Hazelcast ist ein Java-basierter Cluster-Speicher, der Daten selbstständig auf alle Knoten im Netzwerk verteilt. Darüber hinaus können dynamisch neue Knoten hinzugefügt werden, die automatisch die bereits im System gespeicherten Daten erhalten. Des weiteren kann Hazelcast als Cache für Datenbanksysteme genutzt werden, wobei die

Daten nicht im Systemspeicher sondern in der angebundenen Datenbank gespeichert werden.

5.1.5. JBoss AS

JBoss AS ist ein auf Java basierender Open-Source Anwendungsserver (siehe Kapitel 4.1.2) für Java Enterprise Anwendungen [jbo13]. Aufgrund der hohen Flexibilität des JBoss AS können externe Dienste leicht eingebunden und den eingepflegten Anwendungen zur Verfügung gestellt werden. Da der JBoss AS speziell für Java Enterprise Anwendungen entwickelt ist, wird die Verwaltung der von JPA erstellten Transaktionen durch den JBoss AS unterstützt.

5.1.6. MySQL

Bei *MySQL* handelt es sich um ein relationales Datenbanksystem, das sowohl als kommerzielle als auch als Open Source Version angeboten wird [mys13]. Es ist eines der weltweit am meisten genutzten Datenbanksysteme und bietet neben der Unterstützung der SQL-Grundfunktionen den Einsatz von Replikation, Partitionierung und in der Datenbank gespeicherten Funktionen an. Aufgrund der hohen Verbreitung und Beliebtheit stehen für viele Betriebssysteme Anwendungen zur Verwaltung von MySQL DBSs zur Verfügung, wie z. B. *phpMyAdmin* oder *MySQL Workbench*.

5.1.7. Vergleich von Datenbankanbindungen

Eine Datenbankanbindung der Persistenzschicht kann über verschiedene Arten umgesetzt werden. Auf Basis der Programmiersprache Java kann die Datenbank entweder über JDBC angesprochen werden oder über ein ORM basierend auf JPA und Hibernate. Der Vorteil von JDBC ist, dass außer dem Treiber keine weitere Software benötigt wird, um sich mit einer Datenbank zu verbinden. Des Weiteren ist die Entwicklung von Datenbankanbindungen mit JDBC einfach und schnell. Jedoch muss die Struktur der Datenbank dem Entwickler bekannt sein. Da die Datenbankabfragen in SQL im Java-Code

geschrieben werden müssen, kann der Code dadurch schnell unübersichtlich werden. JPA, umgesetzt durch Hibernate, übernimmt die Erstellung der Datenbankstruktur, wodurch die Anbindung einer neuen Datenbank vereinfacht wird. Dies wird ebenfalls noch weiter vereinfacht, indem Hibernate das darunterliegende DBS erkennt und entsprechend integriert. Da Hibernate aus simplen Methodenaufrufen selbstständig SQL-Anfragen generiert, ist der eigentliche Programmcode nicht mit SQL-Anfragen vermischt. Die Transaktionsverwaltung und der Einsatz eines Caches wird sehr gut von Hibernate unterstützt, da dies lediglich in einer Konfigurationsdatei eingetragen werden muss. Jedoch müssen für die Nutzung von Hibernate profunde Kenntnisse über Java Enterprise vorhanden sein. Zusätzlich kann bei zu kleinen Datenmengen der durch Hibernate entstehende Daten-Overhead die Anfragen-Geschwindigkeit immens verringern. Betrachtet man die beiden Datenbankanbindungen qualitativ, ist Hibernate besser geeignet für die Datenbankanbindung des proCollab Prototypen, da die einfachere Konfiguration und der leichtere Wechsel von eingesetzten Komponenten wichtiger ist, als die einfache Anbindung an das DBS.

Um den qualitativen Vergleich der Anbindungen zu erweitern, wurden beide Technologien zur Anbindung eines DBS über ein Experiment verglichen. Es wurde die Geschwindigkeit bei Schreibe- und Update-Operationen verglichen. Lese-Operation können leider nicht ohne weiteres verglichen werden, da Hibernate hier Lazy Loading (siehe Kapitel 2.5.1) umsetzt, was JDBC nicht bietet. Als auszulesende Daten wurde ein künstlicher, aber vergleichbarer Baum gewählt, in dem jeder Knoten lediglich einen String enthielt. Die Zeit wurde vom Zeitpunkt des Absendens der Datenbankabfrage bis zum Zeitpunkt der vollen Speicherung des Baums gemessen. Rahmenparameter der Versuchs waren bzgl. der verwendeten Hardware ein Apple MacBook Pro 15";2,56 GHz Core2Duo; 8GB Ram, sowie als verwendete Datenbank eine MySQL-Datenbank auf einem lokalem Server. Bei der JDBC-Umsetzung wurde für hierarchische Daten eine Adjazenzlisten-Struktur (siehe Kapitel 4.3.2) verwendet. Da in den Knoten keine Attribute für Position und Tiefe vorgesehen waren, wurde mittels einer Queue und eines Hilfsobjekts die aktuelle Position und Tiefe des zu speichernden Elements errechnet. Für die Operationen wurden SQL-Anfragen mit Prepared Statements verwendet [Ora11]. Hibernate verwendet für hierarchische Daten ebenfalls eine Adjazenzlisten-Struktur, jedoch ermittelt es eigen-

ständig die Struktur des Baums. Für die Objekt-Verwaltung wurde der EM (siehe Kapitel 5.1) und als Operations-Aufruf die JPA-Methode *persist(Objekt)* verwendet.
Die Ergebnisse des Experiments können in Tabelle 5.1 eingesehen werden.

Knotenanzahl	Tiefe	Breite	Abfragezeit SQL	Abfragezeit Hibernate
3616	4	15	3,4 s	5,2 s
5461	7	4	5,8 s	7,5 s
9331	6	6	8,6 s	10,0 s

Tabelle 5.1.: Performance-Vergleich zwischen JDBC und Hibernate

Die ähnlichen und bei weniger Knoten höheren Zeiten von Hibernate im Vergleich zu JDBC lassen sich durch den großen Datenüberschuss von Hibernate erklären, der für die Abbildung von Objekten auf Datenbank-Tabellen erstellt wird. Jedoch wird dieser bei größeren Datenmengen stetig vernachlässigbar.
Auf Basis des qualitativen wie auch quantitativen Vergleichs wurde für die Umsetzung des Systems Hibernate gewählt, da es eine höhere Modularität im Vergleich zu JDBC bietet. Darüber hinaus ist der Programm-Code kompakter und übersichtlicher. Außerdem ist die Einbindung und Konfiguration eines Caches bei Hibernate simpler und besser zu ändern, was die Nachteile bzgl. der konkreten Geschwindigkeit bei der Persistierung aufwiegt.

5.2. Technische Architektur

Für die Implementierung der Schnittstellen der Persistenzschicht wurde das sogenannte *Fassaden-Pattern* umgesetzt [GHJV95]. Dieses Pattern beschreibt die Verknüpfung mehrerer Schnittstellen in einer einzigen Schnittstelle. Dies hat zur Folge, dass der Zugriff vereinfacht wird und dem Benutzer die darunterliegenden Schnittstellen verborgen bleiben. Die Persistenzschicht bietet deshalb eine Schnittstelle (`ProCollabPersistence`), die alle Methoden der einzelnen Schnittstellen (siehe Kapitel 4.4) beinhaltet. Die einzelnen Schnittstellen (`Sub-Interfaces`) wurden für jedes systemspezifische Objekt erstellt und setzen die für die jeweiligen Objekte benötigten Methoden um. Diese Methoden umfassen die vom *CRUD*-Pattern (siehe Kapitel 3.3.1) geforderten Funktionen bzw.

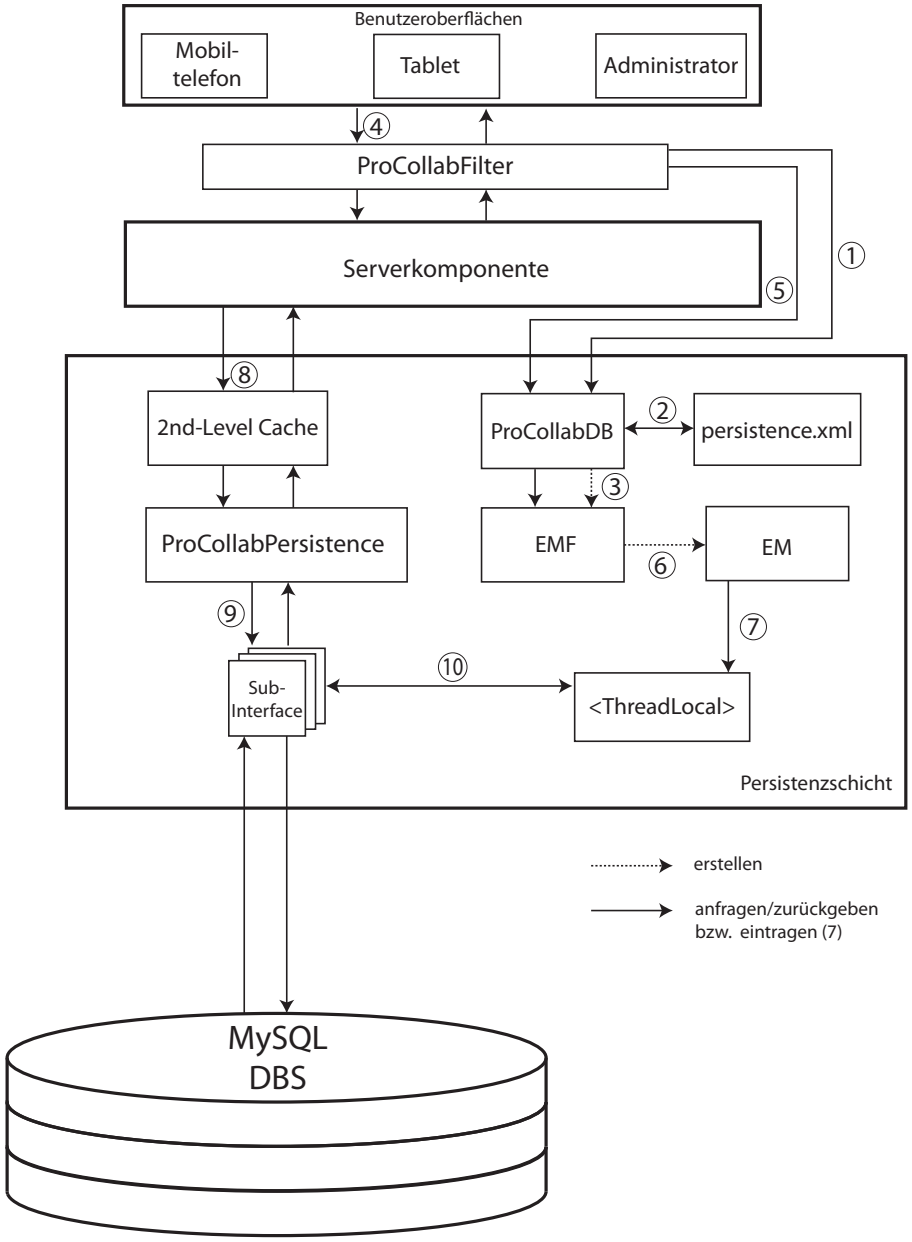


Abbildung 5.1.: Technische Systemarchitektur der Persistenzschicht

Variationen davon, z. B. ”Alle Mitarbeiter einer CLI ausgeben”. Als angebundenes DBS wird eine MySQL-Datenbank verwendet.

Der sogenannte `2nd-Level Cache` ist, bezogen auf Datenverwaltung, ein von Hibernate eingesetzter Cache. Er wird 2nd-Level (dt. zweite Stufe) genannt, da Hibernate bereits selbstständig einen Cache einsetzt, der Objekte über die Existenz-Dauer des mit dem Objekt verbundenen EntityManagers (EM) zwischenspeichert. Der `2nd-Level Cache` speichert Objekte über die Existenzdauer des EM hinweg. Die konkrete Speicherzeit hängt von der persönlichen Konfiguration des eingesetzten Caches ab.

Systemablauf

Beim Starten des proCollab Prototypen wird die `init()`-Methode von `ProCollabFilter`, welcher die Filter-Schnittstelle von Java [Sun07] implementiert, aufgerufen. In dieser `init()`-Methode wird die Klasse `ProCollabDB` aufgerufen (Abbildung 5.2, Schritt 1), welche die für die Konfiguration der Persistenzeinheit (PU) benötigten Daten aus der Konfigurationsdatei lädt (Abbildung 5.2, Schritt 2). Mit dieser Konfiguration wird eine EMF erstellt (Abbildung 5.2, Schritt 3). Anfragen an die Serverkomponente werden über den `ProCollabFilter` geleitet (Abbildung 5.2, Schritt 4), der daraufhin die in `ProCollabDB` gespeicherte EMF aufruft (Abbildung 5.2, Schritt 5). Diese erstellt einen `EntityManager` (Abbildung 5.2, Schritt 6) und trägt ihn in der `ThreadLocal`-Variablen ein (Abbildung 5.2, Schritt 7).
Parallel zu dem in Schritt 5 gestarteten Ablauf wird die in Schritt 4 eintreffende Anfrage an die Serverkomponente weitergeleitet. Es wird überprüft, ob das angeforderte Objekt bereits im Cache gespeichert ist (Abbildung 5.2, Schritt 8). Sollte das Objekt nicht gespeichert sein, wird in `ProCollabPersistence` jene Schnittstelle, die die Anfrage der Serverkomponente bearbeitet, ermittelt (Abbildung 5.2, Schritt 9). Diese Schnittstelle lädt den im `ThreadLocal` gespeicherten EM (Abbildung 5.2, Schritt 10) und ruft über ihn die Datenbank-Anfrage auf. Der dadurch erhaltene Datensatz wird an die Serverkomponente zurückgegeben und im Cache zwischengespeichert.

5.3. Ausgewählte Auszüge der Implementierung

5.3.1. Annotationen

Für die Implementierung der Persistenzschicht des proCollab Prototypen wird Hibernate als ORM gekoppelt mit dem EM von JPA verwendet. Hierzu wurde das Datenmodell in Form von Java-Klassen mit JPA- und Hibernate-Annotationen erweitert. Es wurden Annotationen anstatt der XML-Konfigurationsdateien verwendet, da sowohl Annotationen von JPA als auch Hibernate in den Code eingebunden werden können (siehe Listing 5.3), wohingegen die Verwendung von jeweiligen XML-Konfigurationsdateien aufwändiger gewesen wäre.

5.3.2. Filter

Ein Problem ist, dass eine PU nicht gleichzeitig für Anwendungsserver und Webserver, beispielsweise Apache Tomcat [Apa13], verwendet werden kann, da für eine Webserver-Verwendung Parameter angegeben werden müssen, die bei Anwendungsservern zu Fehlern führen.

Ein Anspruch an das System war jedoch, es so modular wie möglich zu halten. Um trotzdem sowohl Anwendungs- als auch Webserver verwenden zu können, wurde, wie in Listing 5.1 zu sehen, das `Filter`-Interface implementiert. Die `init()`-Methode der Filterklasse wird lediglich beim Start der Anwendung ausgeführt. Somit überprüft das System, ob es eine EMF mit der PU *ProCollabPersistence*, welche für Anwendungsserver konfiguriert ist, erstellen kann. Sollte dies aufgrund einer falschen Konfiguration nicht funktionieren, wird eine `PersistenceException` geworfen. Daraufhin wird mit einer PU-Konfiguration passend für Webserver ebenfalls versucht, eine EMF zu erstellen.

```
1 public class ProCollabDB{
2 ...
3 private static EntityManagerFactory emf;
4 public static EntityManagerFactory setUp() {
```

```
5     if (emf == null) {
6       try {
7           emf = Persistence.
8           createEntityManagerFactory("proCollabPersistence");
9       } catch (PersistenceException e) {
10          emf = null;
11          emf = Persistence.
12          createEntityManagerFactory("proCollabPersistenceLocal");
13      }
14    }
15    return emf;
16  }
17  ...
18  }
19
20  public class ProCollabFilter implements Filter {
21  ...
22  public void init(FilterConfig arg0) throws ServletException {
23      factory = ProCollabDB.setUp();
24    }
25  ...
26  }
```

Listing 5.1: Implementierung mehrerer Persistenz-Units

5.3.3. Laufzeit des EntityManagers

Der EM ist ein sehr leichtgewichtiges Objekt, jedoch nicht *Thread-sicher*. Darunter versteht man, dass ein Objekt oder eine Methode mehrfach gleichzeitig aufgerufen werden kann, ohne dass Fehler hervorgerufen werden [CSD94]. Die Laufzeit des EM ist deshalb je nach Anforderung zu setzen. Es gibt verschiedene Konzepte, dies umzusetzen:

EntityManager-per-Operation Hierbei wird für jede Datenbank-Anfrage ein neuer EM erstellt. Diese Art hält einen EM zeitlich am kürzesten im Speicher. Das Problem hierbei ist jedoch, dass ein Objekt, welches bei einer Transaktion erstellt wurde, keinen Kontext mehr besitzt. Sollte direkt nach dem Erstellen das Objekt geändert werden, muss ein neuer EM erstellt und das Objekt mit dessen Kontext verknüpft werden. Dies ist bei häufigen Datenbank-Anfragen sehr ressourcenintensiv.

EntityManager-per-Anfrage Bei dieser Methode wird für jede Server-Anfrage ein EM erstellt und nach Versenden der Server-Antwort geschlossen. Diese Methode wird häufig in Web-Anwendungen eingesetzt, da hier, sollte der Client Änderungen am Objekt vornehmen wollen, eine neue Anfrage an den Server gestellt wird. Jedoch sind bis zur neuen Anfrage alle Objekte mit keinem Kontext mehr verknüpft, weshalb Objekte, die geändert werden sollen, in einen neuen Kontext eingefügt werden müssen.

Da es sich bei dem proCollab Prototypen um ein webbasiertes System handelt, wurde das EntityManager-per-Anfrage Konzept gewählt. Um auf entsprechende Anfragen reagieren zu können, wurde, wie in Listing 5.2 zu sehen, das Filter-Interface implementiert. Der dadurch implementierte Filter führt bestimmte Methoden beim Eintreffen einer Anfrage aus, in diesem Fall wird ein EM erstellt und in eine sogenannte `ThreadLocal`-Variable gespeichert. Hierbei handelt es sich um eine Speichervariable, welche für jeden Thread, im Falle eines Webservers für jede Abfrage, erstellt wird.

5.3.4. Threadsicherheit des EntityManagers

Objekte, die in einer `ThreadLocal`-Variable gespeichert sind, können nur von dem Thread, der die Variable bestitzt, aufgerufen werden. `doFilter(request,response)` leitet die aktuelle Anfrage zur Verarbeitung an den Server weiter, wodurch in Zeile 17 `ENTITY_MANAGER.remove()` erst aufgerufen wird, wenn der Server die Antwort an den Client zurückgibt. Der Aufruf der Methode löscht den in Zeile 14 erstellten EM aus der `ThreadLocal`-Variablen.
Des Weiteren wird durch die Verwendung einer `ThreadLocal`-Variablen bei der Erstellung neuer Objekte automatisch ein bestimmter Abstand zu der zuletzt von der Daten-

bank erstellten ID eingehalten. Hierdurch wird verhindert, dass bei mehrfachen parallelen Create-Aufrufen es zu Komplikationen wegen gleichen IDs kommt. Der Standard-Versatz beträgt 32768, kann jedoch individuell festgelegt werden. In Listing 5.4, (Zeile 2-6), wird ein spezifischer ID-Generator definiert. Dadurch wird eine Tabelle erzeugt, die die letzte erstellte ID und, bei der Verwendung des Generators für mehrere Klassen, den Klassennamen zwischenspeichert. Durch `allocationSize` wird der Versatz vorgegeben, der bei der Verwendung von `ThreadLocal`-Variablen eingehalten wird.

```
1  public class ProCollabDB{
2  ...
3  public static final ThreadLocal<EntityManager>
4  ENTITY_MANAGER = new ThreadLocal<EntityManager>();
5  ...
6  }
7  public class ProCollabFilter implements Filter {
8  ...
9  public void doFilter(ServletRequest request,
10 ServletResponse response, FilterChain chain)
11       throws IOException, ServletException {
12     EntityManager em = null;
13     try {
14       em = factory.createEntityManager();
15       ProCollabDB.ENTITY_MANAGER.set(em);
16       chain.doFilter(request, response);
17       ProCollabDB.ENTITY_MANAGER.remove();
18     } finally {
19       em.close();
20     }
21   }
22 ...
23 }
```

Listing 5.2: Implementierung eines Filters

Ein Vorteil von ORM ist das bereits beschriebene Lazy Loading (siehe Kapitel 2.5.1), was bei JPA mit Annotationen umgesetzt werden kann. Dadurch müssen, je nach Bedarf, bestimmte oder alle Elemente, die benötigt werden, nachgeladen werden. Soll nur die Referenz auf ein bestimmtes Objekt geladen werden, um dieses z.B. einem anderen Objekt zuzuweisen, kann dies über `EntityManager.getReference(ID)` erreicht

werden. Wenn allerdings auch die zu dem Objekt in Relation stehende Daten, wie bei einer Eltern-Kind-Relation, beim Laden des Objekts mitinstanziiert werden sollen, muss Eager Loading (siehe Kapitel 2.5.1), wie in Listing 5.3 zu sehen, explizit für die Liste mittels der Annotation `(fetch = FetchType.EAGER)` markiert werden. Hier wurde bei der `children`-List (siehe Anhang A.2), Eager Loading der Objekte eingesetzt, da die Kind-Elemente für weitere Methoden benötigt werden.

```
1  @OneToMany(fetch =FetchType.EAGER)
2  @OrderColumn(name = "position")
3  @Cascade(org.hibernate.annotations.CascadeType.ALL)
4  private List<TreeElement> children;
```

Listing 5.3: Eager Loading

5.3.5. Vererbung

Da davon ausgegangen wird, dass die beim proCollab Prototypen verwendeten Bäume von ORTs, ORIs, CLTs und CLIs tief werden und das Datenmodell einfach zu ändern sein soll, wurde die Table-per-Class-Strategie umgesetzt. Somit werden, da lediglich die untersten Klassen der Hierarchie konkret sind, nur Tabellen für diese erstellt. Jedoch enthalten die Tabellen Spalten für Attribute der jeweiligen übergeordneten Klassen. Um zu garantieren, dass alle IDs der Objekte eindeutig sind, wurde in Zeile 6 die mit der `@GeneratedValue` Erzeugungsstrategie `TABLE` verwendet. Durch die Annotation `@GeneratedValue` der IDs übernimmt JPA die Erzeugung und Verwaltung der IDs der Klasse bzw. deren untergeordneten Klassen.

Durch die Verwendung von Vererbung tritt *Polymorphie* der Klassen in der Persistenzschicht auf. Bei Polymorphie handelt es sich um die Eigenschaft eines Objekts, die Instanz von verschiedenen Klassen gleichzeitig zu sein.

Zum Beispiel besitzen beide Klassen `CListType` und `ClistInstance` die Liste `children` die sie beide von `TreeNode` erben (siehe Listing 5.5, Zeile 14). Alle Elemente enthalten ebenfalls das Attribut `parent`, das beide ebenfalls von `TreeElement` erben.

```
1 @Entity
2 @Inheritance(strategy = InheritanceType.TABLE_PER_CLASS)
3 public abstract class TreeElement {
4   @Id
5   @Column(nullable = false, updatable = false)
6   @GeneratedValue(strategy = GenerationType.TABLE)
7 % protected int ID;
8   ...
9 }
```

Listing 5.4: Vererbungshierarchie der genutzten Klassen

Diese Liste steht in einer 1-zu-n Relation mit dem Attribut `Treenode parent` in Zeile 9. Somit kann `parent` sowohl eine Instanz der Klassen `CListItem` und `CListInstance` sein. Die genaue Vererbungshierarchie für ORs ist in Abbildung A.2 und für CLs bzw. CEs in Abbildung A.2 dargestellt. Die Polymorphie stellt für die Programmierung keine Probleme dar, jedoch lassen sich polymorphe Objekte nicht direkt durch ORM in eine Datenbank speichern. Da JPA den Klassen Tabellen zuordnet, können diese Objekte nicht eindeutig einer Tabelle zugewiesen werden.

Da die `children`-Liste entweder `CListType` und `CItemType` oder `CListInstance` und `CItemInstance` enthalten kann, liegt hier ebenfalls eine Polymorphie vor. Dadurch entfällt die Möglichkeit, den Relationspartner der `children`-Liste über `mappedby` anzugeben, da nur eine einzelne Klasse angegeben werden darf (siehe Kapitel 5.1.3). Da das `parent`-Attribut in `TreeElement` deklariert wird (siehe Listing 5.5, Zeile 9) , kann auch nicht mit der `@JoinCollumn`-Annotation (siehe Kapitel 5.1.3) der exakte Relationspartner am `parent`-Attribut angegeben werden. Um dennoch eindeutig der Relation die Partner zuzuweisen, wird in den konkreten Klassen die `getParent()`-Methode überschrieben (siehe Listing 5.5, Zeile 17) und mit `@JoinColumn` annotiert. Dadurch kann das entsprechende Objekt auf den korrekten Typ gecastet werden.

```
1  public abstract class TreeElement{
2    @TableGenerator(name = "TreeGenerator", allocationSize =
        10000, table = "treeGen", pkColumnName = "name",
        valueColumnName = "sequence")
3    @Id
4    @Column(nullable = false, updatable = false)
5    @GeneratedValue(strategy = GenerationType.TABLE, generator =
        "TreeGenerator")
6    protected int ID;
7    ...
8    @ManyToOne
9    protected TreeNode parent;
10 }
11 public abstact class TreeNode extends TreeElement {
12   @OneToMany(fetch =FetchType.EAGER)
13   @OrderColumn(name = "position")
14   @Cascade(org.hibernate.annotations.CascadeType.ALL)
15   private List<TreeElement> children;
16 ...
17 }
18 public class CListType extends TreeNode {
19 ...
20   @ManyToOne
21   @JoinColumn(name="parentID")
22   public CListType getParent(){
23     return (CListType) this.parent;
24   }
25 ...
26 }
```

Listing 5.5: Polymorphismen und ID-Generator

5.3.6. Suche

Eine klare Anforderung an die Persistenzschicht ist es, eine Suche bereitzustellen (siehe Kapitel 3.3). Um weiterhin Modularität im Bezug auf verwendete DBSs zu gewähren, wurde eine dynamische Suche mithilfe des von JPA bereitgestellten `CriteriaBuilder` implementiert (siehe Listing 5.6). Dieser wird vom EntityManager bereitgestellt. In Zeile 5 wird eine Datenbankabfrage mit bestimmten Kriterien für die Suche einer `Person` erstellt und ein Wurzelelement gesetzt. Mit der Methode `crit.from()` wird festgelegt, dass in der Tabelle der Personenklasse gesucht werden soll. Falls nur bestimmte Spalten ausgelesen werden sollen, kann dies über eine `select`-Methode mit Attributen des Wurzelelements als Parameter erreicht werden.
Für eine dynamische Suche wird eine Prädikatenliste erstellt. Prädikate werden benötigt, um gewisse Bedingungen der Datenbank-Anfrage darzustellen. Sollte für `person_search` der Parameter `prename` gesetzt sein, was in Listing 5.6, Zeile 9 überprüft wird, erstellt in Zeile 9 der `CriteriaBuilder` eine Bedingung, die besagt, dass der Parameter dem Attribut des Wurzelelements (`p.get("prename")`) gleichen muss (`cb.equal()`).
Das dadurch erstellte Prädikat wird daraufhin zur Liste hinzugefügt (siehe Listing 5.6, Zeile 10) und es wird mit dem Überprüfen der weiteren Parameter fortgefahren. In Listing 5.6, Zeile 25 werden alle in der Liste stehenden Prädikate in einen Array übertragen. Die Größe des Arrays beträgt 0 (`new Predicate[0]`) da bei der Umwandlung einer Liste in ein Array die Größe an die Liste angepasst wird, sollte die Größe des erstellten Arrays kleiner sein als die Größe der Liste. Mit `cb.where` wird der WHERE-Bereich der Anfrage gesetzt, woraufhin der `CriteriaBuilder` durch das Prädikate-Array iteriert und alle darin enthaltenen Prädikate mit *und* verknüpft(`cb.and()`) und in den WHERE-Bereich einfügt. In Zeile 26 wird mit der konstruierten Suchanfrage eine Datenbankabfrage losgeschickt und die Resultate werden zurückgegeben.

```
1  public ArrayList<Person> person_search(String prename, String
       surname, String email,
2        String organisation) {
3     EntityManager em = ProCollabDB.ENTITY_MANAGER.get();
4     CriteriaBuilder cb = em.getCriteriaBuilder();
5     CriteriaQuery<Person> crit = cb.createQuery(Person.class);
6     final Root<Person> p = crit.from(Person.class);
7     List<Predicate> clist = new ArrayList<Predicate>();
8     if (prename != null) {
9       Predicate p1 = cb.equal(p.get("prename"), prename);
10      clist.add(p1);
11    }
12    if (surname != null) {
13      Predicate p2 = cb.equal(p.get("surname"), surname);
14      clist.add(p2);
15    }
16    if (email != null) {
17      Predicate p3 = cb.equal(p.get("email"), email);
18      clist.add(p3);
19    }
20    if (organisation != null) {
21      Join<Person, Organisation> join = p.join("worksFor");
22      Predicate p4 = cb.equal(join.get("name"), organisation);
23      clist.add(p4);
24    }
25    crit.where(cb.and(clist.toArray(new Predicate[0])));
26    return (ArrayList<Person>) em.createQuery(crit).
         getResultList();
27  }
```

Listing 5.6: Dynamische Suche in der Persistenzschicht

6

Zusammenfassung

6.1. Fazit

Anhand der Konzeptionierung und der prototypischen Implementierung lässt sich zusammenfassend sagen, dass sich ORM mit Hibernate als Framework für die Implementierung einer Persistenzschicht für den proCollab Prototypen eignet. Es muss allerdings bei der Speicherung hierarchisch strukturierter Daten auf die in Kapitel 5 geschilderte Besonderheiten geachtet werden. Hierunter fallen Probleme wie die Umsetzung von Polymorphismen in Hibernate bzw. JPA oder die Wahl des am besten geeigneten Konzepts für die Erstellung des EM.

6.2. Ausblick

Um für die zukünftige Entwicklung Verteilung des DBS zu erreichen, sollte das DBS des proCollab Prototypen durch ein DBS ersetzt werden, das Verteilung und Skalierbarkeit von verteilten Datenbanken unterstützt. Hierfür würden sich Datenbanken wie MySQL Cluster oder eine Erweiterung von Hibernate, Hibernate Shards, eignen. Hibernate Shards wird jedoch seit 2007 nicht weiterentwickelt. Jedoch muss der Grad der Partitionierung von den Anfragen abhängen, da bei verteilteren Anfragen das Sharding angepasst werden muss. Deshalb sind ständige Analysen der Anfragen wichtig.
Neben der Unterstützung von Verteilbarkeit kann die Geschwindigkeit des Systems optimiert werden. Um die Anfragegeschwindigkeit zu verbessern, kann ein sogenannter *Query-Cache* eingesetzt werden. Dieser speichert häufig verwendete Datenbank-Anfragen eines bestimmten Datensatzes zwischen. Die Struktur der Tabellen kann ebenfalls optimiert werden, indem die in Listing 5.3.5 beschriebenen abstrakten Klassen in konkrete Klassen überführt werden, damit die entsprechenden Tabellen der Sub-Klassen aufgeteilt werden.
Um das Hinzufügen von Dokumenten an CE oder CL zu unterstützen, könnte eine dokumentenbasierte Datenbank, eine spezielle NoSQL-Datenbank, angebunden werden. Jedoch muss darauf geachtet werden, dass JPA die verwendete Datenbank unterstützt.

A

Quelltexte

A.1. User Stories

A.1.1. Benutzerverwaltung

Benutzer erstellen

Ein Administrator kann einen Nutzer in der administrativen Umgebung am System registrieren. Dazu kann er den Vorname, Nachnamen, Passwort, E-Mail Adresse und die Firmenzugehörigkeit des Nutzers angeben. Die E-Mail Adresse wird validiert und die Passworteingabe muss zweimalig erfolgen. Nach dem Bestätigen wird dem Nutzer eine Aktivierungs-E-Mail an sein E-Mail Konto gesendet. Nach der Aktivierung erhält der Nutzer eine weitere E-Mail, mit all seinen Nutzerangaben.

Einfache Anmeldung

Ein Nutzer gibt seine E-Mail Adresse und Passwort ein. Anschließend werden seine

Eingaben vom Server überprüft. Bei einer Bestätigung wird ihm das Backend angezeigt, bis er sich vom System abmeldet. Bei Ablehnung, kann er erneut seine Eingaben tätigen.

Benutzerdaten anpassen
Ein Administrator kann die Daten eines Nutzers in der administrativen Umgebung, ändern. Dazu kann er den Vorname, Nachnamen, Passwort, E-Mail Adresse und die Firmenzugehörigkeit des Nutzers ändern. Wird die E-Mail Adresse verändert, muss diese erneut vom Nutzer bestätigt werden. Weiterhin kann ein Administrator den Nutzer einer Rahmeninstanz hinzufügen oder ihn von einer entfernen. Beim Hinzufügen muss die Rahmenrolle angegeben werden. Ein Administrator kann einen anderen Nutzer zum Administrator ernennen oder ihm dieses recht nehmen.

Benutzer entfernen
Ein Administrator kann einen Nutzer, in der administrativen Umgebung, vom System entfernen. Der Benutzer wird vom System abgemeldet und anschließend entfernt. Anschließend wird dem Nutzer eine E-Mail zugesendet, die seine Entfernung bestätigt.

Benutzer suchen
Ein Administrator kann einen Nutzer mit der Suchfunktion, anhand der Attribute Vorname, Nachname, E-Mail Adresse und Firmenzugehörigkeit suchen. Wenn einer oder mehr Nutzer den Vorgaben entsprechen wird eine Liste der Treffer angezeigt. Wird einer der Treffer in der Liste angeklickt, werden die Nutzerdaten angezeigt. Wird kein Nutzer zu den Vorgaben gefunden, wird eine Nachricht angezeigt.

A.1.2. Organisatorische Rahmenverwaltung

Erstellen eines neuen ORT
Ein Administrator kann, in der administrativen Umgebung, eine neue Rahmenvorlage erstellen. Dazu muss er einen Rahmennamen angeben und kann zusätzlich eine Beschreibung, Ziel, Startdatum, Enddatum und die Bearbeitungszeit in Tagen angeben. Die Bearbeitungszeit repräsentiert die Zeit, die für die Erfüllung des Rahmens nach dem Start zur Verfügung stehen. Zusätzlich kann eine Rahmenvorlage einer anderen Rahmenvorlage untergeordnet werden.

Ableitung eines ORT von einem ORT

Ein Administrator kann ein ORT anhand eines bereits existierenden ORT ableiten. Dazu muss er einen existierenden ORT suchen und kann anschließend Details anpassen.

Ableitung eines ORT von einer ORI

Ein Administrator kann einen ORT anhand einer bereits existierenden ORI ableiten. Dazu muss er eine existierende ORI suchen und kann anschließend Details anpassen.

Anpassen von ORT-Details

Nach der Auswahl eines ORT, kann ein Administrator die Rahmendetails anpassen. Er kann den Rahmennamen, Ziel, Startdatum, Enddatum und die Bearbeitungszeit verändern.

Entfernen eines ORT

Nach dem Suchen und Auswahl eines ORT werden dem Administrator die ORTdetails angezeigt. Hier kann er mit Klick auf einen Knopf, den ORT vom System entfernen. Anschließend werden alle Verbindungen zu verbundenen ORI gelöscht.

ORT suchen

Ein Administrator kann einen ORT mit der Suchfunktion, anhand der Attribute ORname, Beschreibung und Ziel suchen. Wenn eine oder mehr ORTs den Vorgaben entsprechen wird eine Liste der Treffer angezeigt. Wird einer der Treffer in der Liste angeklickt, werden die ORdetails angezeigt. Wird kein ORT zu den Vorgaben gefunden, wird eine Nachricht angezeigt.

Instanziieren eines ORT

Ein Administrator kann eine neue ORI anhand eines existierenden ORT instanziieren. Dazu muss er einen ORI suchen und instanziieren klicken. Anschließend kann ein Administrator die ORIdetails spezifizieren und muss einen verantwortlichen ORmanager bestimmen. Weiterhin kann er die ORI einer anderen unterordnen.

Anpassen von ORdetails

Nach der Auswahl einer ORI, kann ein Administrator die ORdetails anpassen. Er kann den ORnamen, Ziel, Startdatum, Enddatum und die Bearbeitungszeit verändern.

Entfernen einer ORI

Nach dem Suchen und Auswahl einer ORI werden dem Administrator die ORdetails angezeigt. Hier kann er mit Klick auf einen Knopf, den ORI vom System entfernen. Anschließend werden alle Verbindungen zu weiteren ORI gelöscht.

Abschließen einer ORI

Nach dem Suchen und Auswahl einer ORI kann ein Administrator die ORI abschließen. Nach dem Abschluss wird die ORI archiviert.

ORI suchen

Ein Administrator kann eine ORI mit der Suchfunktion, anhand der Attribute ORname, Beschreibung und Ziel suchen. Wenn eine oder mehr ORIs den Vorgaben entsprechen wird eine Liste der Treffer angezeigt. Wird einer der Treffer in der Liste angeklickt, werden die ORdetails angezeigt. Wird keine ORI zu den Vorgaben gefunden, wird eine Nachricht angezeigt.

A.1.3. Checklisteneintragsverwaltung

Erstellen eines CET

Ein Administrator kann neue CETs erstellen. Hierzu muss er dem CET einen Namen und kann zusätzlich einen Text und einen CEstatus angeben.

Anpassen eines CET

Nach dem Suchen und Auswahl eines CET kann ein Administrator den CET anpassen. Er kann den Namen, den Text und den Status ändern.

Entfernen eines CET

Nach dem Suchen und Auswahl eines CET kann ein Administrator den CET mit einem Klick auf einen Knopf entfernen.

CET suchen

Ein Administrator kann einen CET mit der Suchfunktion, anhand der Attribute Namen und Text suchen. Wenn eine oder mehr CETs den Vorgaben entsprechen wird eine Liste der Treffer angezeigt. Wird einer der Treffer in der Liste angeklickt, werden die CEdetails angezeigt. Wird kein CET zu den Vorgaben gefunden, wird eine Nachricht angezeigt.

Instanziieren eines CET

Ein Administrator kann, beim Erweitern einer CLI, neue CEIs anhand eines existierenden CET instanziieren. Hierzu muss ein CET gesucht werden. Die CEdetails werden angezeigt und können angepasst werden. Mit einem Knopf kann der CET zu einer CEI instanziiert werden. Diese wird dann der Liste hinzugefügt.

Erstellen einer CEI

Ein Administrator kann, beim Erweitern einer CLI, neue CEIs ohne Instanziierung einer Eintragsvorlage erstellen. Hierzu muss er dem Eintrag einen Namen, einen Text und einen CEstatus angeben.

Anpassen einer CEI

Nach dem Suchen und Auswahl einer CEI kann ein Administrator die CEI anpassen. Er kann den Namen, den Text und den Status ändern.

Entfernen einer CEI

Nach dem Suchen und Auswahl einer CEI kann ein Administrator die CEI mit einem Klick auf einen Knopf entfernen.

CEI suchen

Ein Administrator kann eine CEI mit der Suchfunktion, anhand der Attribute Namen, Text, Status und Abschluss suchen. Wenn eine oder mehr CEIs den Vorgaben entsprechen wird eine Liste der Treffer angezeigt. Wird einer der Treffer in der Liste angeklickt, werden die CEdetails angezeigt. Wird keine CEI zu den Vorgaben gefunden, wird eine Nachricht angezeigt.

Abschluss einer CEI

Nach der Auswahl einer CEI, kann ein Administrator diese mit einem Klick abschließen.

A.1.4. Checklistenverwaltung

Erstellen eines CLT

Ein Administrator kann einen neuen CLT erstellen. Hierzu kann ein Name, eine Beschreibung und ein Status angegeben werden. Anschließend kann der Administrator CET hinzufügen und Nutzer mit ihren zugehörigen Listenrollen angeben.

Anpassen eines CLT

Nach dem Suchen und Auswahl eines CLT kann ein Administrator den CLT anpassen. Er kann den Namen, die Beschreibung und den Status anpassen.

Entfernen eines CLT

Nach dem Suchen und Auswahl eines CLT kann ein Administrator den CLT mit einem Klick auf einen Knopf entfernen.

CLT suchen

Ein Administrator kann einen CLT mit der Suchfunktion, anhand der Attribute Namen und Beschreibung suchen. Wenn eine oder mehr CLTs den Vorgaben entsprechen wird eine Liste der Treffer angezeigt. Wird einer der Treffer in der Liste angeklickt, werden die CLdetails angezeigt. Wird kein CLT zu den Vorgaben gefunden, wird eine Nachricht angezeigt.

Erweiterte Operationen

Bewegen von Einträgen

Ein Administrator kann die Position von CETs verändern.

Einfügen von Listenvorlagen

Ein Administrator kann CLTs anderen CLTs hinzufügen. Hierfür muss der Administrator einen weiteren CLT auswählen, eine Position selektieren und diese, mit einem Knopf, hinzufügen.

Erstellen einer CLI

Ein Administrator kann eine neue CLI erstellen. Hierzu kann ein Name, eine Beschreibung und ein Status angegeben werden. Anschließend kann der Administrator Einträge hinzufügen und Nutzer mit ihren zugehörigen CLrollen angeben.

Instanziieren einer CLI

Ein Administrator kann, beim Erweitern einer ORI, neue CLIs anhand eines existierenden CLT instanziieren. Hierzu muss ein CLT gesucht werden. Die CLdetails werden angezeigt und können angepasst werden. Mit einem Knopf kann der CLT zu einer CLI instanziiert werden. Diese wird dann der ORI hinzugefügt.

Abschluss einer CLI

Nach der Auswahl einer CLI, kann ein Administrator diese mit einem Klick abschließen. Diese wird anschließend archiviert.

Anpassen einer CLI

Nach dem Suchen und Auswahl einer CLI kann ein Administrator die CLI anpassen. Er kann den Namen, die Beschreibung, und den Status anpassen.

Entfernen einer CLI

Nach dem Suchen und Auswahl einer CLI kann ein Administrator die CLI mit einem Klick auf einen Knopf entfernen.

CLI suchen

Ein Administrator kann eine CLI mit der Suchfunktion, anhand der Attribute Namen, Beschreibung, Status und Abschluss suchen. Wenn eine oder mehr CLIs den Vorgaben entsprechen wird eine Liste der Treffer angezeigt. Wird einer der Treffer in der Liste angeklickt, werden die CLdetails angezeigt. Wird keine CLI zu den Vorgaben gefunden, wird eine Nachricht angezeigt.

Erweiterte Operationen

Bewegen von CEIs

Ein Administrator kann die Position von CEIs verändern.

Einfügen von CLIs

Ein Administrator kann CLIs anderen CLIs hinzufügen. Hierfür muss der Administrator eine weitere Listeninstanz auswählen, eine Position selektieren und diese, mit einem Knopf, hinzufügen.

A.1.5. Übergreifende Verwaltung

Hinzufügen eines Benutzers zu einem ORI

Nachdem ein Benutzer gesucht wurde, kann ein Verwalter den Benutzer zu einem ORI des Verwalters hinzufügen und die Rolle des Benutzers anpassen.

Hinzufügen eines Benutzers zu einem ORI

Nachdem ein Benutzer gesucht wurde, kann ein Administrator den Benutzer zu einem beliebigen ORI hinzufügen und die Rolle des Benutzers anpassen.

Einen Benutzer von einer ORI entfernen

Während eine OI geöffnet ist, kann ein Verwalter einen Benutzer von der ORI entfernen. Dies muss durch einen weiteren Dialog vom Verwalter bestätigt werden.

A.2. Datenmodell mit Attributen

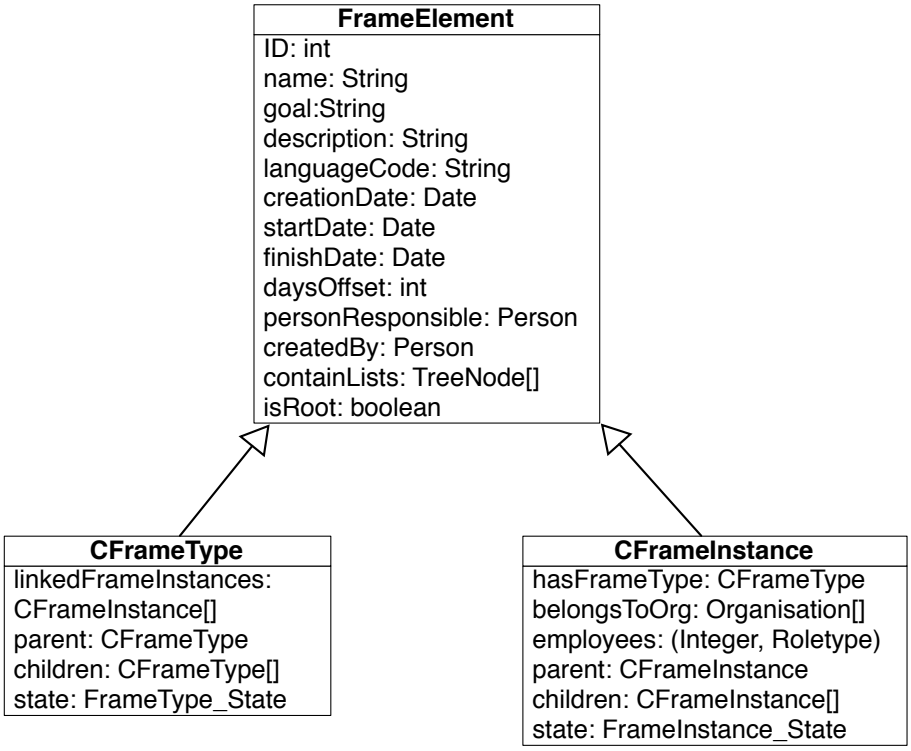


Abbildung A.1.: Klassenhierarchie des Organisatorischen Rahmens

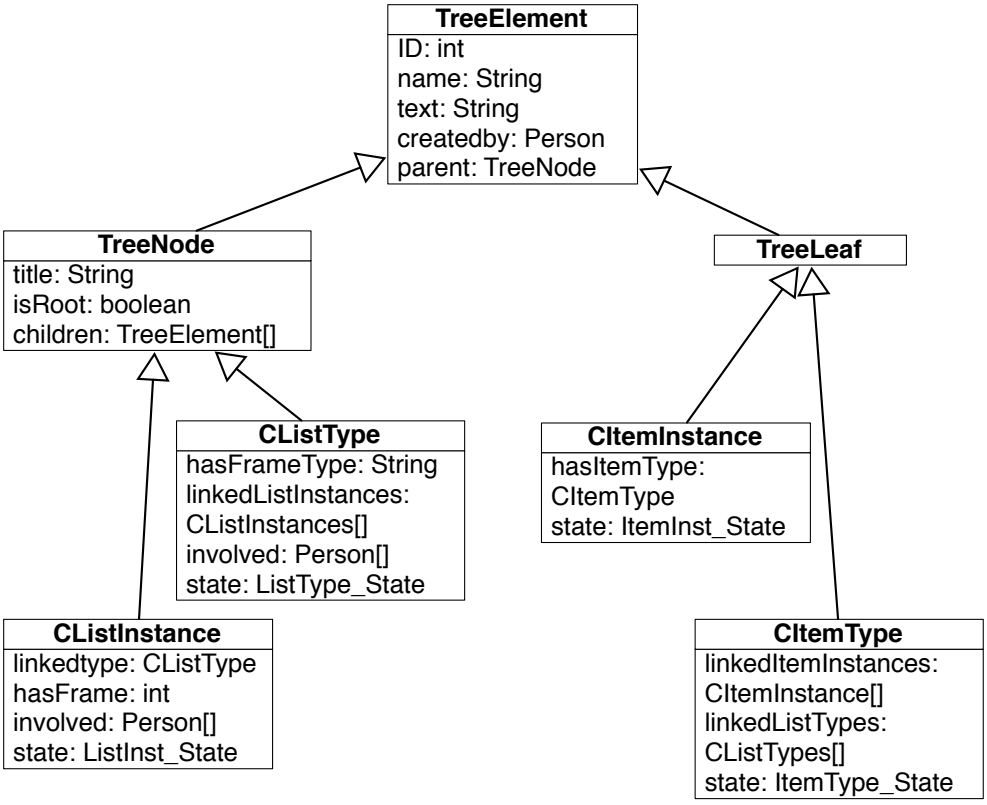


Abbildung A.2.: Klassenhierarchie der Checklisten

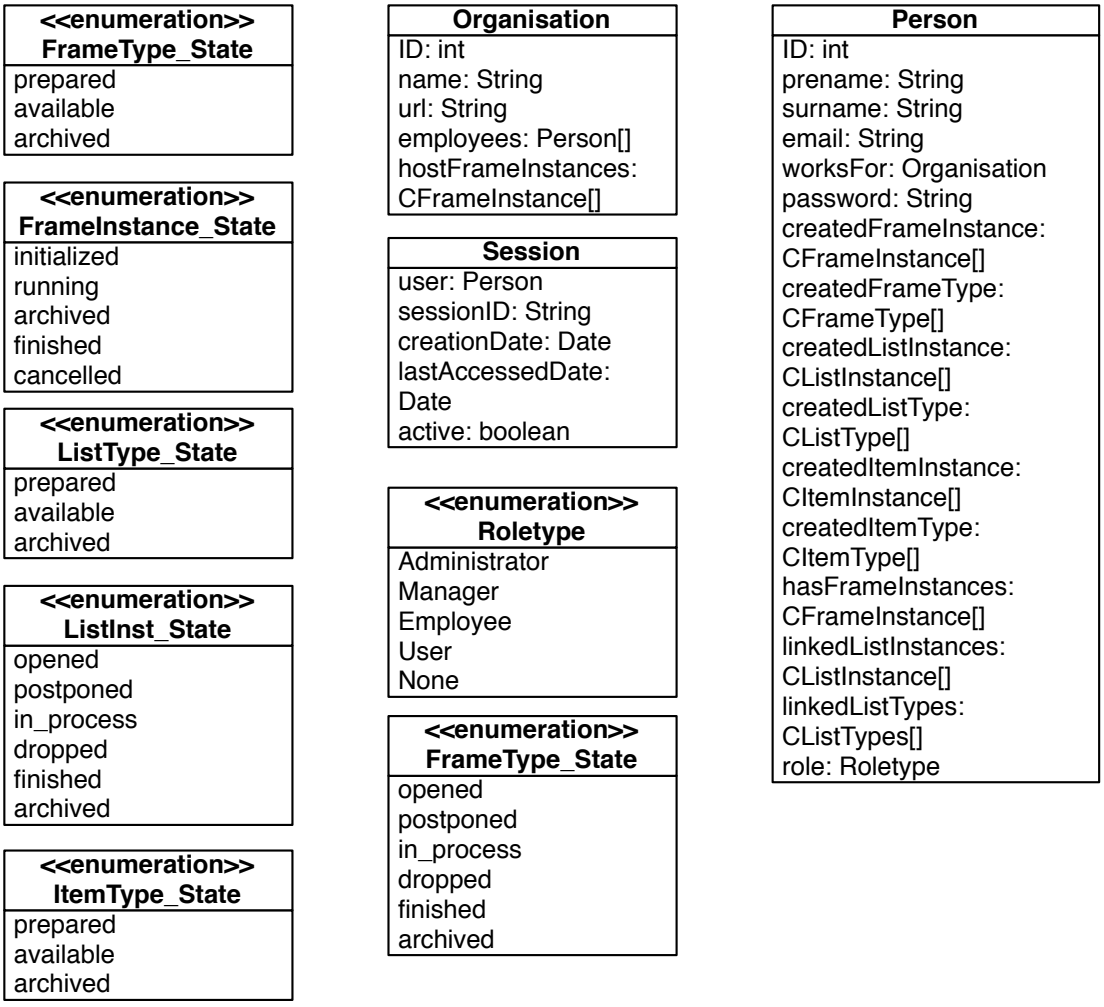


Abbildung A.3.: Relationale Klassen

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[Adj09] *Adjacency list vs. nested sets: SQL Server | EXPLAIN EXTENDED*. http://explainextended.com/2009/09/25/adjacency-list-vs-nested-sets-sql-server/. Version: 2009. – Abruf: 23.10.2013

[Apa13] APACHE SOFTWARE FOUNDATION: *Apache Tomcat - Welcome!* http://tomcat.apache.org/. Version: 2013. – Abruf: 13.11.2013

[BG81] BERNSTEIN, PA ; GOODMAN, N: Concurrency control in distributed database systems. In: *ACM Computing Surveys (CSUR)* (1981). http://dl.acm.org/citation.cfm?id=356846

[Bre00] BREWER, EA: Towards robust distributed systems. In: *PODC* (2000). http://openstorage.gunadarma.ac.id/~mwiryana/Kuliah/Database/PODC-keynote.pdf

[CDG08] CHANG, F ; DEAN, J ; GHEMAWAT, S: Bigtable: A distributed storage system for structured data. In: *ACM Transactions on ...* (2008). http://dl.acm.org/citation.cfm?id=1365816

[Cel04] CELKO, Joe: *Trees and Hierarchies in SQL for Smarties*. Elsevier, 2004. – ISBN 978–0123877338

[cou13] *Apache CouchDB*. http://couchdb.apache.org/. Version: 2013. – Abruf: 12.11.2013

[CSD94] CHOWDAPPA, Aswini K. ; SKJELLUM, Anthony ; DOSS, Nathan E.: *Thread-Safe Message Passing with P4 and MPI*. http:

//www.cis.uab.edu/sites/default/files/hpcl-documents/messaging-papers/p4-mpi-pthreads.pdf. Version: 1994

[db213] *IBM - DB2 database software*. http://www-01.ibm.com/software/data/db2/. Version: Oktober 2013. – Abruf: 12.11.2013

[FTD12] FLORATOU, A ; TELETIA, N ; DEWITT, DJ: Can the elephants handle the NoSQL onslaught? In: *Proceedings of the ...* (2012). http://dl.acm.org/citation.cfm?id=2367511

[GHJV95] GAMMA, Erich ; HELM, Richard ; JOHNSON, Ralph ; VLISSIDES, John: *Design Patterns: Elements of Reusable Object-Oriented Software*. Addison-Wesley, 1995

[IBM13] *IBM - Rational DOORS - Germany*. http://www-03.ibm.com/software/products/de/ratidoor/. Version: November 2013. – Abruf: 11.11.2013

[Int13] INTERNATIONAL ORGANIZATION OF STANDARDIZATION: *ISO/IEC 9075-1:2011 - Information technology – Database languages – SQL – Part 1: Framework (SQL/Framework)*. http://www.iso.org/iso/home/store/catalogue_tc/catalogue_detail.htm?csnumber=53681. Version: 2013. – Abruf: 12.11.2013

[jbo13] *JBoss Application Server 7 - JBoss Community*. http://www.jboss.org/jbossas. Version: 2013. – Abruf: 23.10.2013

[KP88] KRASNER, Glenn E. ; POPE, Stephen T.: A cookbook for using the model-view controller user interface paradigm in Smalltalk-80. In: *Journal of Object-Oriented Programming* 1 (1988), August, Nr. 3, 26–49. http://dl.acm.org/citation.cfm?id=50757.50759. – ISSN 0896–8438

[LM10] LITH, A ; MATTSSON, J: Investigating storage solutions for large data. In: *Master's thesis. Chalmers ...* (2010). http://jakobadam.googlecode.com/svn-history/r161/trunk/src/msc-report.pdf

[MKR13] MUNDBROD, N ; KOLB, J ; REICHERT, M: Towards a System Support of Collaborative Knowledge Work. In: *Business Process Manage-*

ment ... (2013). http://link.springer.com/chapter/10.1007/978-3-642-36285-9_5

[mon13] *MongoDB*. http://www.mongodb.org/. Version: 2013. – Abruf: 12.11.2013

[Mun13] MUNDBROD, Nicolas: *Interner Bericht: Nutzung von Checklisten zur Unterstützung der Entwicklung mechatronischer Systeme im Automobilbereich*. Ulm, 2013

[mys13] *MySQL :: The world's most popular open source database*. http://www.mysql.com/. Version: 2013. – Abruf: 12.11.2013

[NCWD84] NAVATHE, S ; CERI, S ; WIEDERHOLD, G ; DOU, J: Vertical partitioning algorithms for database design. In: *ACM Transactions on ...* (1984). http://dl.acm.org/citation.cfm?id=2209

[neo13] *Neo4j - The World's Leading Graph Database*. http://www.neo4j.org/. Version: 2013. – Abruf: 12.11.2013

[Ora11] ORACLE CORPORATION: *PreparedStatement (Java Platform SE 6)*. http://docs.oracle.com/javase/6/docs/api/java/sql/PreparedStatement.html. Version: 2011. – Abruf: 13.11.2013

[Ora13] ORACLE CORPORATION: *Java SE Technologies - Database*. http://www.oracle.com/technetwork/java/overview-141217.html. Version: 2013. – Abruf: 13.11.2013

[Pri08] PRITCHETT, Dan: BASE: AN ACID ALTERNATIVE. In: *Queue* 6 (2008), 48–55. http://dx.doi.org/10.1145/1394127.1394128. – DOI 10.1145/1394127.1394128. – ISSN 15427730

[sql13] *SQLite Home Page*. http://www.sqlite.org/. Version: 2013. – Abruf: 12.11.2013

[Sun07] SUN MICROSYSTEMS: *Filter (Java EE 5 SDK)*. http://docs.oracle.com/javaee/5/api/javax/servlet/Filter.html. Version: 2007. – Abruf: 07.11.2013

[Tro05] TROPASHKO, Vadim: Nested intervals tree encoding in SQL. In: *ACM SIGMOD Record* 34 (2005), Juni, Nr. 2, 47. http://dx.doi.org/10.1145/1083784.1083793. – DOI 10.1145/1083784.1083793. – ISSN 01635808

[Wor08] WORCESTER POLYTECHNIC INSTITUTE: *Orville Flight Assistant by Matt Tavares*. http://web.cs.wpi.edu/~rich/courses/cs525u-s08/projects/mtavares/. Version: 2008. – Abruf: 7.11.2013

[Wor09] WORLD HEALT ORGANISATION: *Surgical Safety Checklist*. http://whqlibdoc.who.int/publications/2009/9789241598590_eng_Checklist.pdf. Version: 2009. – Abruf: 26.10.2013

Name: Daniel Reich Matrikelnummer: 711164

Erklärung

Ich erkläre, dass ich die Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel verwendet habe.

Ulm, den .....

Daniel Reich