Karlsruher Institut für Technologie

aifb

KONZEPTION UND REALISIERUNG EINES MICROSERVICES ZUR SYNCHRONISIERUNG VON NUTZEREINSTELLUNGEN

Bachelorarbeit von
LUKAS STRUPPEK

an der KIT-Fakultät für Wirtschaftswissenschaften

eingereicht am	:	19. Dezember 2017
Studiengang	:	Wirtschaftsingenieurwesen
Referent	:	Prof. Dr. Andreas Oberweis, Prof. Dr. York Sure-Vetter
Betreuer	:	Andreas Fritsch

Institut für Angewandte Informatik und Formale Beschreibungsverfahren
KIT - Die Forschungsuniversität in der Helmholtz-Gemeinschaft

ABSTRACT DEUTSCH

Mobile Anwendungen spielen im täglichen Leben eine große Rolle. Mit steigender Anzahl vernetzter Systeme wächst auch die Anzahl an Daten, die zwischen verschiedenen Geräten desselben Anwenders konsistent gehalten werden müssen. Hierzu sind zuverlässige Prozesse der Datensynchronisation erforderlich.

Den Kern der Arbeit bildet die Konzeption einer Serveranwendung zur geräteübergreifenden Datensynchronisation. Realisiert wird diese als Microservice, einem modernen Ansatz zur Modularisierung von Software.

Es erfolgt eine Einführung in die Software-Architektur der Microservices im Kontrast zu monolithischen Systemen. Hierzu wird ein Überblick gegeben über die verfügbaren, leichtgewichtigen Kommunikationstechnologien REST, SOAP und Messaging, die in verteilten Systeme zum Einsatz kommen. Für die serverseitige Datenhaltung wird zusätzlich ein entsprechendes Datenbanksystem benötigt. Es findet ein Vergleich traditioneller, relationaler Systeme mit den Konzepten von NoSQL-Datenbanken statt, aus welchen das dokumentenorientierte Datenbanksystem CouchDB verwendet wird.

Der praktische Aspekt befasst sich mit der Implementierung des Microservices unter Verwendung des Java-Frameworks Spring. Betrachtet werden diverse Aspekte der Implementierung sowie des strukturellen Aufbaus. Die Umsetzung wird anschließend mithilfe von Softwaretests auf die Erfüllung der definierten Spezifikationen hin untersucht. Das Deployment der Anwendung über Software-Container rundet den praktischen Teil der Arbeit ab.

ABSTRACT ENGLISCH

In daily life mobile applications getting more and more important. A rise in user data comes along with a growing number of cross-linked systems. Reliable processes are needed to grant a data consistency between different devices of the same user.

This thesis focuses on the conception of a server application for data synchronization. The application is implemented as a microservice, a modern approach for modularized software.

Besides an introduction into the topic of software architectures, attention is centered on microservices as a contrary architecture to monolithic systems. At this point an overview over the available, light-weighted technologies for communication, REST, SOAP and Messaging, is given as well. To hold the data on the server persistently, an appropriate data base system is required. For that purpose, a comparison between traditional, relational systems and the concept of NoSQL databases is performed. As a result, the document-oriented database CouchDB has been chosen.

The practical aspect addresses the realization of the microservice using the java framework Spring. A closer look will be taken on different aspects of implementation and structure of the software. Software tests are used to analyze compliance with the defined specifications. The deployment of the application by using software containers marks the end of the practical part of this thesis.

INHALTSVERZEICHNIS

ABBILDUNGSVERZEICHNIS

TABELLENVERZEICHNIS

ABKÜRZUNGEN

ACID atomicity, consistency, isolation und durability

AIFB Institut für Angewandte Informatik und Formale Beschreibungsverfahren

AOP Aspektorientierte Programmierung

API Application Programming Interface

AVARE Anwendung zur Verteilung und Auswahl rechtskonformer Datenschutzeinstellungen

CRUD Create, Read, Update, Delete

DBMS Datenbankmanagementsystem

DI Dependency Injection

DTO Data Transfer Object

EJB Enterprise Java Bean

FZI Forschungszentrum Informatik in Karlsruhe

HATEOAS Hypermedia as the Engine of Application State

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

JEE Java Enterprise Edition

JSON JavaScript Object Notation

MVC Model View Controller

MVCC Multiversion Concurrency Control

NoSQL Not Only SQL

POJO Plain Old Java Object

REST Representational State Transfer

RPC Remote Procedure Call

SQL Structured Query Language

SSL Secure Sockets Layer

TLS Transport Layer Security

UC Use Case

UI User Interface bzw. Nutzeroberfläche

URI Uniform Resource Identifier

XML eXtensible Markup Language

ZAR Zentrum für Angewandte Rechtswissenschaft

1 EINLEITUNG

1.1 MOTIVATION

Im Zeitalter der Digitalisierung leben die Menschen in einer zunehmend vernetzten Welt. Laut einer Studie von Cisco IBSG werden im Jahr 2020 auf jeden Menschen der Welt 6,58 vernetzte Geräte kommen. Während für 2015 etwa 25 Milliarden vernetzte Geräte geschätzt worden sind, so werden es zehn Jahre später die doppelte Anzahl sein, Fortschritte in der Internet- oder Gerätetechnologie noch nicht berücksichtigt [Eva17, vgl. S.3f].

In einer vernetzten Welt, wie sie heute schon besteht, greifen verschiedene Geräte häufig auf dieselben Datensätze zu. Nutzer-individuelle[1] Fitnessdaten beispielsweise werden durch Smartwatches erhoben, in einer zentralen Datenbank abgelegt und können anschließend über Smartphone, Tablet oder den Webbrowser am Rechner abgerufen und analysiert werden. Notwendig für eine geräteübergreifende, konsistente Bereitstellung von Daten ist ein Synchronisationsprozess, welcher dafür sorgt, dass jedes Gerät stets auf aktuelle, konsistente Daten Zugriff besitzt.

Dienste für die Synchronisation von Daten werden zumeist in Form von Serveranwendungen bereitgestellt. Neben traditionellen, monolithischen Systemen, die ein „großes Ganzes“ bilden, haben sich in den vergangenen Jahren diverse Alternativen entwickelt. Hierzu zählt auch die Aufspaltung eines monolithischen Systems in viele kleine Programme, sogenannte Microservices, von denen jedes genau einen Dienst bereitstellt. Jeder Microservice arbeitet für sich gesehen unabhängig von anderen Programmen und kann somit einzeln in Produktion gegeben werden. Dies führt zu schrumpfenden Entwicklerteams und einer insgesamt agileren Softwareentwicklung. Sie eigenen sich daher optimal für die Umsetzung einer Serveranwendungen zur geräteunabhängigen Datensynchronisation.

1 Aus Gründen der besseren Lesbarkeit wird in der vorliegenden Arbeit auf eine geschlechtsspezifische Differenzierung verzichtet.

1.2 INHALTLICHER AUFBAU DER ARBEIT

Das erste Kapitel der vorliegenden Arbeit liefert einen kurzen Einstieg in das Thema der Datensynchronisation sowie Microservices und erklärt die Motivation hinter diesen Konzepten.

Im zweiten Kapitel werden die Zielsetzung der Arbeit, das Konzept der Synchronisation sowie der praktische Einsatz im Kontext des Projekts AVARE besprochen.

Die Kapitel drei bis fünf befassen sich mit den theoretischen Grundlagen der vorliegenden Arbeit.
Das dritte Kapitel erläutert das Konzept der Software-Architekturen und als modernen Vertreter davon die Microservices als Gegenentwurf zu monolithischen Systemen. Beide Konzepte werden ausführlich vorgestellt und die Vorteile der Realisierung der Serveranwendungen in Form eines Microservices aufgezeigt. Als Ergänzung zum Thema Microservices werden Methoden zur Kommunikation der Programme untereinander verglichen und insbesondere auch auf das Thema REST als Programmierparadigma eingegangen.

Kern des vierten Kapitels ist eine Darstellung des Spring Frameworks, welches den technischen Unterbau des entstandenen Microservice bildet. Mit Spring Boot und Spring Web MVC werden zusätzlich zwei Teilprojekt von Spring detaillierter betrachtet, die eine zentrale Rolle in der Umsetzung spielen.

Im fünften Kapitel werden die Konzepte relationaler sowie NoSQL-Datenbanksysteme verglichen. Die verschiedenen Varianten von NoSQL-Datenbanken werden anschließend separat betrachtet. Es folgt schließlich eine detailliertere Auseinandersetzung mit dem verwendeten NoSQL-Datenbanksystem Apache CouchDB.

Kapitel sechs präsentiert den praktischen Teil der Arbeit und beschäftigt sich mit der programmiertechnischen Umsetzung der Aufgabenstellung. Es wird auf die Struktur des Microservice und seiner Bestandteile detailliert eingegangen und verschiedene Aspekte erläutert. An verschiedenen Stellen werden für das bessere Verständnis weitere theoretische Grundlagen behandelt, die zuvor unerwähnt geblieben sind.

Im siebten Kapitel wird sich dem Prinzip der Softwaretests angenommen und verschiedene Varianten dieser beschrieben. Besonders Augenmerk wird auf die Realisierung von Integrationstests gelegt, welche die korrekte Funktionalität der einzelnen Komponenten im Zusammenspiel prüft.

Das achte Kapitel umfasst das Software Deployment, d.h. die Inbetriebnahme des Microservice sowie des gewählten Datenbanksystems mithilfe des Container-Tools Docker.

Das letzte, neunte Kapitel beinhaltet eine kritische Reflexion der Arbeit sowie der eingesetzten Techniken und Konzepte. Die Bewertung findet im Kontext alternativer Lösungsansätze statt. Zusätzlich werden Grenzen und Probleme bei der Umsetzung der Arbeit aufgezeigt und bewertet. Ein Ausblick hinsichtlich des Einsatzgebietes und möglicher Weiterentwicklungen des Microservice bilden den Abschluss der Arbeit.

Im Anhang finden sich diverse Unterlagen, welche für das grundlegende Verständnis der Arbeit nicht notwendig sind und daher ausgegliedert worden sind. Für den praktischen Einsatz bzw. einer möglichen Erweiterung des Projekts liefern diese hilfreiche Ergänzungen.

2 ZIEL DER ARBEIT

2.1 AUFGABENSTELLUNG UND ZIELSETZUNG

Die vorliegende Arbeit beschäftigt sich im Kern mit der Konzeption und Realisierung einer Serveranwendungen für eine geräteübergreifende Synchronisation von Daten. An die Umsetzung wurden diverse Anforderungen gestellt, die im vorliegenden Abschnitt erläutert werden.

Grundsätzlich sollen die zu synchronisierenden Datensätze auf den verschiedenen, den Service nutzende Geräten auch dann verfügbar sein, wenn diese nicht mit dem Internet verbunden sind. Möglich ist dies über den Einsatz redundanter Datensätze, d.h. die Geräte erhalten eine Replikation der jeweiligen Datensätze aus der Serverkomponente. Replikation beschreibt die erstmalige Einführung einer Redundanz im Speicher eines Endgerätes, welches zuvor noch nicht über die Daten verfügte. Somit ist eine Bearbeitung der Daten in Offline-Szenarien weiterhin möglich.

Wechselt ein Gerät aus dem Offline- in einen Online-Zustand, so muss eine Konsistenz der Daten zwischen den Geräten, welche diese Daten gemeinsam nutzen, garantiert werden. An diesem Punkt setzt die Synchronisation an, welche dafür zuständig ist, einen Abgleich der Daten zwischen Clientgeräten und Server vorzunehmen und sicherzustellen, dass nur eine gültige, aktuelle Version jedes Datensatzes existiert [MS04, vgl. S.79f]. Die Serveranwendungen soll sowohl für die Replikation als auch für die Synchronisation entsprechende Dienste anbieten.

Eine Eingrenzung der Endgeräte hinsichtlich verwendetem Betriebssystem oder Programmiersprache geschieht nicht. Daher wird die Serveranwendungen unabhängig von den Endgeräten und deren Spezifikationen entworfen. Hierzu gehört neben einem unabhängigen Austauschformat für die Daten eine Kommunikationsart, welche mit verschiedenen Programmiersprachen umgesetzt werden kann. Zur Absicherung des Transports der Daten zwischen Client und Server wird eine Verschlüsselung der Verbindung gefordert, sodass ein unberechtigtes Mitlesen unterbunden wird.

Da die Software im Kontext des Projekt AVARE entsteht, ist eine Veröffentlichung der Serveranwendungen als Open-Source-Software geplant. Daraus ergibt sich die Anforderung, bei der Realisierung des Projekts ausschließlich Komponenten zu verwenden, welche unter der Apache-Lizenz 2.0 bzw. damit kompatiblen Lizenzen vertrieben werden. Die genannte Lizenz garantiert dem Lizenznehmer das unbefristete, weltweite, nicht-exklusive, gebührenfreie Recht, das vorliegende Werk zu vervielfältigen, weiterzuentwickeln, veröffentlichen und zu verteilen. Außerdem besteht das Recht zur Lizenzweitergabe [o.V17f, vgl.]. Durch die ausschließliche Verwendung von Software unter dieser und damit kompatiblen Lizenzen entstehen keine zusätzlichen Kosten für das Projekt.

Weitere Einschränkungen in Bezug auf die Art der Realisierung bzw. der verwendeten Software oder Programmiersprache sind nicht gegeben. Es wurde sich für die Realisierung in Form eines Microservice entschieden. Für die Ablage der Daten wird für jeden Anwender ein Profil angelegt, das in einer Datenbank gespeichert und dessen Verwaltung von der Serveranwendungen übernommen wird. Aus Serversicht ergeben sich daraus folgende fünf Anwendungsfälle (Use Cases (UCs)):

1. UC Bei erstmaliger Anmeldung eines Anwenders über ein Endgerät beim Server wird für diesen ein Profil mit individueller ID generiert oder ein Profil basierend auf einer bestehenden ID wiederhergestellt. Diese ProfileID soll dabei eine solche Komplexität aufweisen, dass eine doppelte Vergabe derselben ID an zwei unterschiedliche User nicht möglich ist.

2. UC Dem Nutzer soll die Möglichkeit gegeben werden, sein Profil auf dem Server zu löschen. Hierzu wird das Profil durch eine verschlüsselte Nachricht ersetzt, aus welcher die Client-Geräte die Aufforderung zum Beenden der Synchronisierung der Daten entnehmen können. Das ersetzte Profil bleibt zunächst weiterhin in der Datenbank erhalten, allerdings ohne Nutzerdaten.

3. UC Wird auf einem Client-Gerät ein Profil aktualisiert, so wird dieses bei der nächsten Verbindung mit dem Server auf diesen übertrage (Push-Vorgang). Um jeweils nur die aktuelle Version des Profils zu besitzen, nimmt der Server einen Vergleich der letzten Änderung des Profils des Clients und des Servers vor und lehnt veraltete Profile ab. Um Ungenauigkeiten beim Vergleich der Zeitstempel sowie eine Überlastung des Servers zu vermeiden, wird eine Zeitspanne festgelegt, um welche ein Client-Profil mindestens aktueller sein muss als das Server-Profil, um ein Überschreiben in der Datenbank zu rechtfertigen.

4. UC Verbindet sich ein Client-Gerät erstmals mit dem Server oder wurde das Profil auf dem Server aktualisiert, so kann das Client-Gerät die aktuelle Version anfordern (Pull-Vorgang). Auch hier findet ein Zeitstempelvergleich der Profile statt, wobei ebenfalls eine festgelegte Zeitspanne überschritten sein muss, damit ein Profil auf dem Server als aktueller gilt verglichen mit dem des Clients.

5. UC Nach einem vorher festgelegten Zeitraum werden nicht genutzte Profile aus der Datenbank gelöscht. Hierunter fallen sowohl ungenutzte Profile als auch auf den Status unsynchronisiert gesetzte Profile, welche ebenso erst nach Ablauf der Zeitspanne gelöscht werden (siehe 2. UC).

Um diesen Anforderungen gerecht zu werden, wird weiterhin eine Datenstruktur für die Verwaltung der Profile innerhalb des Servers gefordert. Jedes Profil besitzt auf dem Server eine eindeutige ID, anhand welcher Client und Server die Profile eindeutig identifizieren können. Hinzu kommen zwei Zeitstempel: der erste legt fest, zu welchem Zeitpunkt ein Client zuletzt auf das entsprechende Profil auf dem Server zugegriffen hat und wird zum Aufspüren ungenutzter Profile verwendet (5. UC). Der zweite Zeitstempel beinhaltet die Information, wann die vom Nutzer gespeicherten Daten zuletzt aktualisiert wurden.

Durch den Vergleich des zweiten Zeitstempels ist es möglich, Profile hinsichtlich der Aktualität ihrer Daten zu vergleichen, was insbesondere beim Push- oder Pull-Vorgang (3. und 4. UC) zum Einsatz kommt. Die vierte Komponente eines Profils sind die abzulegenden Daten selbst. Diese liegen in Form eines Strings, d.h. einer einfachen Zeichenkette, vor. Eine Verschlüsselung dieser durch den Client ist ebenso möglich wie die Ablage lesbarer Texte. Für den Server soll es nicht möglich sein, den Inhalt der Zeichenkette zu interpretieren, sodass eine automatische, maschinelle Auswertung der Daten unterbunden wird und die Privatsphäre des Nutzers gewährleistet ist.

Um die Profile über einen längeren Zeitraum zu speichern wird zusätzlich eine Datenbankkomponente benötigt, auf welche die Softwarekomponente Zugriff besitzt und diese verwaltet. Um einen unautorisierten Zugriff Dritter zu vermeiden, ist die Kommunikation nach außen auf die Softwarekomponente zu beschränken, ein direkter Datenbankzugriff von außen soll nicht ohne weiteres möglich sein. Eine Auswahl des Datenbanksystems soll dem Zweck dienend stattfinden, ist grundsätzlich jedoch freigestellt.

Bei der Inbetriebnahme der Serveranwendungen ist ein einfaches Deployment gewünscht, sodass größere vorzunehmende Einstellungen oder zusätzliche Installationen vermieden werden. Zugleich soll die Möglichkeit bestehen, an zentraler Stelle einzelne Parameter wie den Zeitraum vor dem Löschen ungenutzter Profile festzulegen. Ziel ist es, die fertige Software ohne große Vorkenntnisse und zusätzliche Einarbeitung in Betrieb nehmen zu können.

Die vorliegende Arbeit und deren Ergebnis sind für sich genommen unabhängig von der Art der zu verwaltenden Daten, solange sich diese in Form eines Strings repräsentieren lassen. Nichtsdestotrotz entsteht die Software im Kontext des Projekt AVARE, in welchem sie Einsatz findet. Im folgenden Abschnitt wird dieses Projekt kurz vorgestellt.

2.2 PROJEKT AVARE

Mit Aufkommen der digitalen Revolution, welche neben der Industrie und Forschung insbesondere auch alle Bevölkerungsschichten durchdringt, steigt die digitale Vernetzung und Transparenz der Menschen in sämtlichen Alltagssituationen. Bei der Nutzung alltäglicher Smartphone-Apps wie beispielsweise die des Videoportals YouTube gibt der Anwender mehr persönliche Daten von sich preis, als er sich zunächst bewusst sein dürfte: so fordert die App des Videodienstes Berechtigungen für den Abruf des Standorts, den Zugriff auf persönliche Kontakte oder den Empfang von SMS [o.V17j, vgl.]. Ohne genaues Hinsehen bei der Installation dürften die gewährten Berechtigungen den wenigsten Menschen auffallen.

Um den Menschen ein Stück digitale Souveränität zurückzugeben, wurde von der Baden-Württemberg Stiftung im Rahmen des Programms IKT-Sicherheit[1] das Projekt „Anwendung zur Verteilung und Auswahl rechtskonformer Datenschutzeinstellungen (AVARE)“ ins Leben gerufen. AVARE ist ein Kooperationsprojekt zwischen dem Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB), dem Zentrum für Angewandte Rechtswissenschaft (ZAR) und dem Forschungszentrum Informatik in Karlsruhe (FZI). Das Projekt wurde mit einem Budget von 597.000 Euro am 01. November 2015 gestartet und ist auf drei Jahre Laufzeit ausgelegt [FZI16, vgl.].

AVARE stellt grundsätzlich vier Funktionalität bereit:

- Zentrale Steuerung: Jeder Nutzer kann zentral auf seinem Endgerät ein Präferenzprofil anlegen, mithilfe dessen er auf intuitive Weise an zentraler Stelle Einstellungen bezüglich seiner Datenschutzpräferenzen festlegen kann.
- Geräteübergreifende Synchronisation: Über einen zentralen Dienst kann der Nutzer sein einmal konfiguriertes Präferenzprofil auf allen von ihm genutzten Geräte verteilen und aktuell halten. Dies soll unabhängig vom verwendeten Betriebssystem möglich sein. An diesem Punkt findet die vorliegende Arbeit Einsatz.
- Feststellung von Präferenzverletzungen: AVARE prüft bei jeder Installation bzw. jedem Update von Software, ob und inwiefern diese die festgelegten Präferenzen verletzen. Ist dies der Fall, wird der Anwender über diesen Vorgang automatisch informiert.

1 Weitere Informationen unter: `https://www.bwstiftung.de/forschung/programme/neue-technologien/ikt-sicherheit/`

- Reaktion auf Präferenzverletzungen: Bei der Verletzung der Präferenzen sind dem Nutzer grundsätzlich drei Möglichkeiten gegeben, auf diese zu reagieren. Zunächst einmal kann der Nutzer die Anwendung für den gesamten Datenzugriff sperren. Daneben ist eine feingranulare Festlegung von Zugriff auf einzelne Daten möglich. Die dritte Möglichkeit umfasst die Bereitstellung von sogenannten Ersatzdaten, welche der Anwendung falsche Daten vortäuschen.

Ziel des aktuell laufenden Projekts ist eine lauffähige Android-Anwendung, welche als Prototyp für weitere Betriebssysteme und Anwendungsbereiche gilt. Die Ergebnisse werden schließlich als Open-Source-Software veröffentlicht [FZI16, vgl.].

Abbildung 1 zeigt grafisch das Konzept von AVARE und das Einsatzgebiet des im Rahmen dieser Arbeit entstehenden Servers als Schnittstelle zwischen verschiedenen Endgeräten zur Synchronisierung der Nutzereinstellungen im Kontext von AVARE.

Abbildung 1: Konzipierte Funktionsweise des AVARE-Systems [FZI16]

3 SOFTWARE-ARCHITEKTUR

3.1 DEFINITION VON SOFTWARE-ARCHITEKTUREN

An dieser Stelle wird zunächst ein theoretischer Blick auf die grundlegende Struktur der im Rahmen dieser Arbeit entstehenden Serverkomponente geworfen werden. Hierbei spielt das Konzept der Software-Architekturen eine wichtige Rolle. Der Begriff ist in der Literatur nicht eindeutig festgelegt. Eine mögliche Definition liefern Bass, Clements und Kazman:

„The software architecture of a program or computing system is the structure or structures of the system, which comprise software elements, the externally visible properties of those elements, and the relationships among them.“[1] [BCK10, S.3]

Zunächst einmal legt eine Architektur die Software-Elemente fest, aus welchen das Projekt besteht. Die Architektur bildet eine Abstraktion der Implementierung der Elemente und betrachtet die Interaktion dieser untereinander. Aus der Definition geht hervor, dass ein System nicht auf eine einzige Struktur festgelegt ist, sondern aus verschiedenen Strukturen bestehen kann. Ein System besitzt häufig eine übergeordnete Struktur, welche das Zusammenspiel verschiedener Elemente beschreibt. Jedes dieser Elemente selbst kann intern auf andere Art und Weise strukturiert sein.

Die eigentliche Implementierung einzelner Elemente bildet keinen Bestandteil einer Software-Architektur, welche sich nur mit dem von außen sichtbaren Verhalten beschäftigt. Festzuhalten ist auch, dass nach der Definition jedes Software-System eine Architektur besitzt, unabhängig von der Qualität dieser. Eine Bewertung im Zuge einer Evaluation der Architektur stellt nichtsdestotrotz einen wichtigen Aspekt der Software-Entwicklung dar [BCK10, vgl. S.20ff].

1 Deutsch etwa: Die Software-Architektur eines Programms oder Computersystems beschreibt die Struktur bzw. die Strukturen des Systems, welche aus Software-Elementen bestehen, den nach außen hin sichtbare Eigenschaften dieser Elemente sowie derer Beziehung untereinander.

3.2 MONOLITHISCHE ARCHITEKTUR

Als traditioneller Ansatz zur Softwareentwicklung existiert die monolithische Architektur. Diese lässt sich folgendermaßen beschreiben:

„Monolithische Software-Architekturen verbinden ihre funktionalen Elemente in einem einzigen, untrennbaren sowie homogenen Gebilde. [...] Eine monolithische Struktur folgt nicht dem Ansatz einer expliziten Gliederung in Teilsysteme oder Komponenten. So sind diese Systeme häufig stark gebunden an Ressourcen wie Hardware, bestimmte Datenformate und proprietäre Schnittstellen.“[Le13]

Ein Monolith durchläuft als komplette Einheit alle Phasen der Softwareentwicklung, hierzu gehören unter anderem das Testen, die Abnahme und der Release der Software. Die monolithische Struktur widerspricht jedoch nicht der Möglichkeit, die Software intern in einzelne Module zu unterteilen. Die einzelnen Produktionsschritte muss das System hingegen stets als ganze Einheit durchlaufen [Wol16, vgl. S. 3].

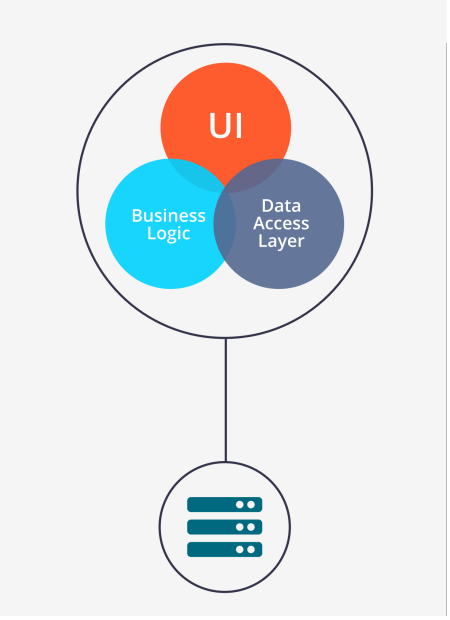

Abbildung 2: Aufbau einer monolithischen Architektur [Bue16]

Abbildung 2 zeigt eine grundsätzliche Struktur für einen Monolithen. Wie zu erkennen ist, bilden die drei Bereiche User Interface bzw. Nutzeroberfläche (UI), Business Logic (Anwendungslogik) und Data Access Layer (Datenzugriff) zusammen eine gemeinsame Einheit. Der Zugriff auf die Datenbank erfolgt aus dieser Einheit heraus, d.h. der Monolith als Ganzes ist mit dieser verbunden, ein Austausch der Datenbank ist nicht ohne Neustart des kompletten Systems möglich.

Die Entwicklung von monolithischen Systemen gilt als relativ einfach, solange das Projekt nicht über eine leicht handhabbare Größe hinaus wächst. Da am Ende der Entwicklung ein einziges Programm bzw. eine einzige Datei vorliegt, ergeben sich beim Deployment der Software in der Regel keine größeren Probleme. Durch die Ausführung mehrerer Instanzen eines monolithischen Programms und einem vorgeschalteten Load Balancer ist eine horizontale Skalierbarkeit[2] umsetzbar [NSS14, vgl. S.24]. Der Load Balancer wird den Instanzen vorgeschalten und verhält sich nach außen hin wie eine einzige Instanz des Software, während er im Hintergrund die Anfragen auf mehrere Instanzen verteilt [Wol16, vgl. S. 146].

Probleme bei monolithischen Systemen ergeben sich insbesondere bei wachsendem Umfang der Software. Für neue Mitarbeiter wird es zunehmend schwerer, sich in bestehende Programmstrukturen einzuarbeiten, der Austausch einzelner Mitarbeiter oder ganze Teams erweist sich als großes Problem für eine funktionierende Weiterentwicklung und Wartung der Software. Auch in bestehenden Teams sinkt die Produktivität mit wachsender Codebasis, da die von einzelnen Entwicklern geschaffenen Komponenten ihre Unabhängigkeit verlieren und somit Änderungen im Programm stets mit dem gesamten Team abgestimmt werden müssen. Kleine Updates einzelner Programmbestandteile oder Module benötigen stets ein Redeployment der gesamten Software. Der vollständige Austausch einzelner Bestandteile oder der Einsatz neuer Frameworks funktionieren meist nicht ohne tiefgreifende Änderungen im gesamten Projekt und können somit nicht ohne weiteres vollzogen werden. Die Abhängigkeit von den initial gewählten Softwarekomponenten ist folglich groß und wächst mit der Größe des Projekts [NSS14, vgl. S.24].

Aufgrund der bei monolithischen Architekturen auftretenden Probleme haben sich in der Industrie diverse Alternativen zu großen Monolithen entwickelt, von welchen in dieser Arbeit der Ansatz der Microservices detailliert betrachtet wird.

2 Höhere Bereitstellung an Ressourcen (i.d.R. mehrere Server), die jeweils einen Teil der anfallenden Rechenlast bearbeiten [Wol16, vgl. S. 150].

3.3 MICROSERVICE ARCHITEKTUR

Die Definition von Microservices ist in der Literatur ebenfalls nicht eindeutig definiert. Einen allgemeinen Definitionsansatz liefern Fowler und Lewis:

"[...] Microservice-Stil [ist] ein Ansatz für die Entwicklung einer einzigen Anwendung in Form einer Reihe kleiner Services, die jeweils in einem eigenen Prozess laufen und die durch einfache Mechanismen kommunizieren [...]. Diese Dienste [...] sind durch vollautomatisches Deployment unabhängig voneinander deploybar. Es gibt nur ein Minimum an zentraler Verwaltung dieser Dienste, die unterschiedliche Programmiersprachen wie auch Datenspeicher-Technologien verwenden können"[FL15].

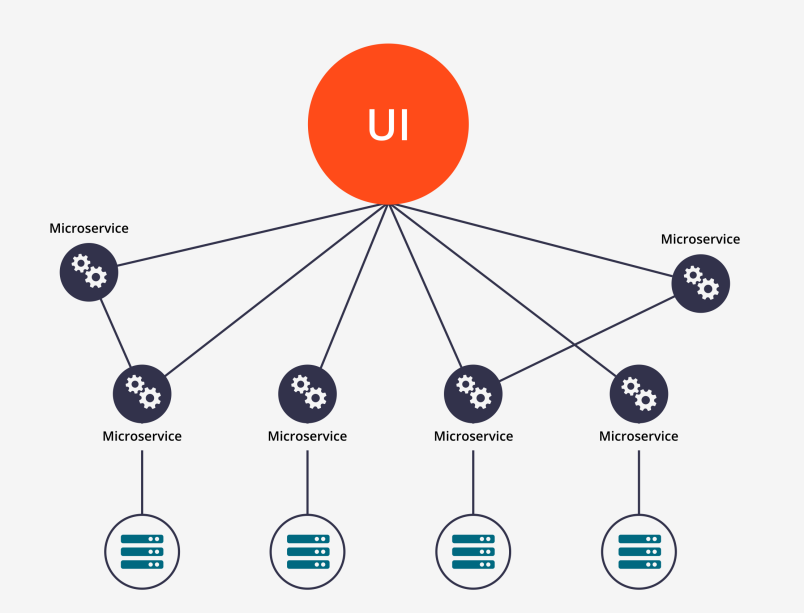

Abbildung 3: Aufbau einer Microservice-Architektur [Bue16]

Abbildung 3 zeigt eine mithilfe von Microservices realisierte Auflösung der in Abbildung 2 dargestellten monolithischen Struktur. In der Architektur bildet das UI die Schnittstelle zum Nutzer. Die Business Logic ist in viele voneinander unabhängige Microservices unterteilt, die in sich abgeschlossen sind, jedoch untereinander über Schnittstellen kommunizieren. Auch existiert in diesem Konzept nicht eine einzige Datenbank für das komplette System, sondern besitzen verschiedene Microservices jeweils ein eigenes, vom Rest des Systems separiertes Datenbanksystem.

Microservices stellen ein effektives Konzept zur Modularisierung von Software dar. Durch die Unterteilung in einzelne Services und den fest definierten Schnittstellen werden Abhängigkeiten innerhalb der Software minimiert, einzelne Bestandteile lassen sich unabhängig vom Gesamtpaket überarbeiten und ersetzen. Dies ist gerade in Zeiten agiler Softwareentwicklung und kürzeren Entwicklungszyklen ein großer Vorteil gegenüber monolithischen Strukturen, da einzelne Microservices separat entwickelt und insbesondere automatisch getestet und in Anwendung gesetzt werden können [Wol16, vgl. S. 59ff].

Microservices bieten zudem die Möglichkeit jeden Service individuell zu skalieren und auf einem oder mehreren Servern in Betrieb zu nehmen, was es ermöglicht, stark beanspruchten Services die notwendigen Kapazitäten zur Verfügung zu stellen [Wol16, vgl. S.64f]. Ebenfalls kann für die Realisierung jedes Microservice eine für den Anwendungsbereich passende Programmiersprache oder ein entsprechendes Framework genutzt werden. Dadurch können neue Technologien schneller übernommen werden, ohne ganze Strukturen zu verändern. Bedingung ist jedoch, dass die Kommunikation über fest definierte Schnittstellen beibehalten wird [New15, vgl. S.4f].

Die Verwendung von Microservices als Ersatz monolithischer Strukturen wird in der Literatur durchaus auch kritisch bewertet. So werden durch die notwendigen Aufrufe der einzelnen Microservices untereinander häufig Overhead[3]-lastige Kommunikationen aufgebaut, welche zulasten der Laufzeit und Netzwerkperformance gehen. Hingegen bieten die Methodenaufrufe in Monolithen bessere Laufzeiteigenschaften, da diese innerhalb derselben Laufzeitumgebung stattfinden [FL15, vgl. S.3]. Zusätzlich ist eine stabile Netzwerkumgebung im laufenden Betrieb notwendig, um die einzelnen Microservices zuverlässig einsetzen zu können, da bei einem Netzwerkausfall das gesamte System zum Stillstand kommen kann. Durch eine entsprechend hochverfügbare Hardware kann dieses Problem zwar minimiert werden, stellt jedoch auch einen Kostentreiber in Projekten dar. Auch birgt die Verwendung unterschiedlicher technologischer Ressourcen bei der Realisierung verschiedener Microservices neben einem großen Potenzial auch das Risiko steigender Komplexität des Gesamtsystems, sodass ähnlich wie bei monolithischen Strukturen es insbesondere für neue Mitarbeiter oder Projektgruppen schwer ist, das Gesamtprojekt zu überblicken [Wol16, vgl. S.76f].

3 Zusätzliche Informationen, die zur Nachrichtenübermittlung benötigt werden.

Fowler postuliert auf seinem Blog einen pragmatischen Ansatz zur Umsetzung der Software-Architektur, der den Einsatz von Monolithen als Basis neuer Projekte vorsieht und argumentiert, dass ein schlecht skalierbares, aber erfolgreiches monolithisches Software-System dem Einsatz von Microservices vorzuziehen ist. Gerade zu Beginn eines Softwareprojekts werden bei der Verwendung von Microservices hohe Anforderungen an eine Festlegung der Software-Architektur und die Koordination der involvierten Entwicklerteams gestellt, was zu einem langsameren Projektstart führt. Fowler beschreibt daher eine Monolith-First Strategie, die im Kern eines Software-Projekts eine monolithische Struktur vorsieht, welche sich über den Einsatz von definierten Schnittstellen und Microservices weiterentwickelt. Es gibt jedoch auch Stimmen, welche von Beginn an die Verwendung von Microservices fordern, unter anderem um die Entwickler an die Arbeit in kleinen Teams und die klare Abgrenzung von Zuständigkeiten zu gewöhnen [Fow15, vgl.].

Nach Abwägung der Vor- und Nachteile der vorgestellten Architekturansätzen wurde sich in der vorliegenden Arbeit für eine Orientierung an der Microservice Architektur entschieden. Insbesondere unter dem Aspekt der noch laufenden Entwicklung im Projekt AVARE und die daraus zu erwartende Ausweitung des Projekts auf weitere Anwendungsbereiche und Betriebssysteme ergab sich die Anforderung nach einer strikten Abgrenzung der Softwarekomponente und der dazugehörigen, klaren Definition von Schnittstellen. Zur Umsetzung einer solchen Architektur wurde das Spring Framework verwendet, welches auf Java basiert und in Kapitel 4 eingeführt wird. Auf die technologische Festlegung von Schnittstellen und Kommunikation geht der nachfolgende Abschnitt näher ein.

3.4 KOMMUNIKATIONSTECHNOLOGIEN FÜR MICROSERVICES

Der im Rahmen dieser Arbeit implementierte Microservice wird keine gesonderte Nutzeroberfläche enthalten und stattdessen eine grafische Oberfläche über das Dokumentationstool Swagger bereitstellen (vgl. 6.10). Daher werden an dieser Stelle die Möglichkeiten REST, SOAP und Messaging als Technologien zur Kommunikation mit den Microservices sowie derer untereinander vorgestellt.

3.4.1 *REST*

Representational State Transfer (REST) beschreibt einen Architekturstil für Webstandards und basiert auf dem Hypertext Transfer Protocol (HTTP), dem im Internet meistgenutzten Transportprotokoll, weshalb REST häufig fälschlicherweise selbst als Protokoll bezeichnet wird [HH10, vgl. S.367]. Grundsätzlich basiert REST auf den folgenden fünf Prinzipien:

- Ressourcen mit eindeutiger Identifikation: Der Zugriff auf einzelne Ressourcen erfolgt über sogenannte Uniform Resource Identifier (URI), denen ein konsistentes Schema zugrunde liegt. Jede Ressource ist eindeutig adressierbar über eine global individuelle ID [Til11, vgl. S.11f].
- Standardmethoden: Beim Zugriff auf einzelne Ressourcen wird der durch HTTP festgelegte Satz an Methoden genutzt (vgl. Anhang A), mit deren Hilfe festgelegt wird, wie mit einer einzelnen Ressource umgegangen wird. Dies ermöglicht einen Ressourcenzugriff über alle Anwendungen, die mit HTTP arbeiten können und denen die entsprechende URI bekannt ist [Til11, vgl. S.13ff].
- Hypermedia: Dieses Konzept basiert auf dem Prinzip von Links und ist unter dem Namen Hypermedia as the Engine of Application State (HATEOAS) bekannt. Bedingt durch die Tatsache, dass neben Menschen auch Maschinen in der Lage sind, Links zu folgen, ermöglicht HATEOAS es einem Server seinem Client über Links mitzuteilen, welche Aktionen er im Folgenden ausführen kann. Indem der Client einem solchen Link folgt überführt er den Server von einem Zustand in den nächsten [Til11, vgl. S.12f].
- Unterschiedliche Repräsentationen: Die Repräsentation von übermittelten Daten wird durch einzelne Datenformate festgelegt. Kann ein Client mit einem gewissen Datenformat arbeiten, so ist er in der Lage mit jeder Ressource zu interagieren, welche sich in diesem Datenformat repräsentieren lässt. Meist wird, sofern möglich, auf Standardformate wie JavaScript Object Notation (JSON) oder eXtensible Markup Language (XML) zurückgegriffen [Til11, vgl. S.16].

- Zustandslose Kommunikation: Dieses Prinzip sieht vor, dass der Server keine Zustände über die Anfrage eines Clients hinweg speichert. Nach Bearbeitung dieser besitzt der Server keinen clientspezifischen Zustand mehr, welcher Informationen über die erfolgte Anfrage und deren Ergebnis beinhaltet. Daraus ergibt sich die Anforderung an jede Anfrage, alle notwendigen Informationen zur Bearbeitung zu beinhalten [Til11, vgl. S.17].

Eine REST-Anfrage sei nun beispielhaft am Vorlesungsverzeichnis des KIT über KitHub[4] dargestellt. Die einzelnen Fakultäten, Institute und Vorlesungen sind jeweils eindeutig als Ressource ansprechbar, der folgende Link greift mittels eines GET-Aufrufs auf Details zur Vorlesung Web Science im Wintersemester 17/18 zu: `https://www.kithub.de/vvz/26419/events/55593`.

In diesem Aufruf ist das Institut für Angewandte Informatik und Formale Beschreibungsverfahren (AIFB) als Ressource mit der ID 26419 gekennzeichnet, durch den Anhang */events/* wird spezifiziert, dass auf Veranstaltungen dieses Instituts zugegriffen wird, unter welchen die Vorlesung Web Science die eindeutige ID 55593 besitzt. Als Rückgabe liefert der Webservice ein HTML-Dokument, welches alle grundsätzlichen Informationen enthält und von jedem Client mit HTML-Interpreter gelesen werden kann.

Das grundlegende Ziel dieser Arbeit beinhaltet die Erstellung einer Serveranwendung, realisiert als Microservice, zur Synchronisierung von Nutzereinstellungen. Hierbei handelt es sich um klar definierbare Ressourcen mit eindeutigen Abfrage- und Manipulationsaufträgen. Dies legt den Einsatz einer REST-Architektur nahe. Die Plattformunabhängigkeit der Schnittstelle und die weite Verbreitung von HTTP als Transportstandard sind weitere essentielle Vorteile von REST, weswegen sich für den Einsatz dieser Technologie entschieden wurde. Der Vollständigkeit halber werden in den nachfolgenden Abschnitten zwei Alternativen vorgestellt.

3.4.2 *SOAP*

SOAP ist ein leichtgewichtiges Kommunikationsprotokoll, welches auf XML, einer erweiterbaren Auszeichnungssprache, basiert. Ursprünglich als Akronym für Simple Object Access Protocol gewählt, wird SOAP mittlerweile als Eigenname verwendet, da die Bezeichnung nicht mehr der Definition entspricht. Die Spezifikation von SOAP legt fest, wie Nachrichten formuliert werden, aber stellt keine Vorgaben, wie diese übermittelt werden [HH10, vgl. S.92f].

4 Siehe auch `https://www.kithub.de/vvz`

SOAP kann als Ersatz für REST in einer Microservice-Architektur eingesetzt werden und ebenfalls HTTP als Kommunikationsprotokoll verwenden, wobei hier stets POST-Nachrichten gesendet werden (vgl. Anhang A). Beim Einsatz von SOAP werden Methoden von entfernten Objekten in anderen Prozessen aufgerufen, weshalb auch von einem Remote Procedure Call (RPC) gesprochen wird. Die verfügbaren Technologien ermöglichen unter anderem auch die Weiterleitung von Aufrufen von einem Microservice an andere Microservices, um beispielsweise zusätzliche Sicherheitsvorkehrungen wie Verschlüsselungen effektiv einzubinden. Aufgrund vieler vorhandener Erweiterungen für SOAP können komplexe Protokollschichten entstehen, welche die Isolation der Microservices aufweichen und der Projektübersicht schaden. Ein allgemeiner Nachteil von SOAP im Vergleich mit REST sind fehlende Mechanismen wie HATEOAS für eine flexible Handhabung der Schnittstellen [Wol16, vgl. S.182f].

Für den zu erzeugenden Microservice wäre ein Einsatz von SOAP anstelle von REST ebenso möglich gewesen. Die Nutzung von REST wurde aus den in Abschnitt 3.4.1 dargelegten Gründen einer Anwendung von SOAP jedoch vorgezogen.

3.4.3 *Messaging*

Mithilfe von Messaging-Systemen kann eine Kommunikation zwischen Programmen und Services über den Versand einzelner Nachrichten untereinander bewerkstelligt werden. Es wird auch von einer losen Kopplung zwischen Sender und Empfänger gesprochen, da beide Seiten abgesehen vom Nachrichtenformat und den Zieladressen keine Informationen über die jeweils andere Seite benötigen. In Java ist das Messaging beispielsweise über den Java Message Service realisiert [o.V13, vgl.].

Der Einsatz von Messaging unterscheiden sich von REST und SOAP durch eine asynchrone Kommunikation, d.h. nach dem Senden der Nachricht wartet der Sender nicht auf eine direkte Antwort vom Empfänger, diese kann später geschehen oder ausbleiben. Dies verhindert eine Blockade des Service, welcher bei längeren Antwortzeiten beim Einsatz einer synchronen Kommunikation in seinem Zustand verharrt. Insbesondere bei langen Aufrufketten zwischen Services ist dies vorteilhaft, da es zu keinen Einbußen der Performance aufgrund blockierender Services kommt. Auch kann trotz eines Netzwerkausfalls eine gesendete Nachricht durch Zwischenspeicherung in einem Messaging-System zu einem späteren Zeitpunkt weiterbearbeitet werden [Wol16, vgl. S.183ff].

Für die Implementierung wird ein Messaging-Server und somit zusätzliche Infrastruktur benötigt, welche eine hohe Verfügbarkeit aufweisen muss, um ein stabiles System zu gewährleisten. Die Struktur einer Software-Architektur ist bei der Verwendung von Messaging schwerer zu konzipieren als beim Einsatz einer synchronen Kommunikation, was die Komplexität eines Softwareprojekts gerade zu Beginn negativ beeinflussen kann [Wol16, vgl. S.186f]. Aus diesen Gründen empfiehlt sich der Einsatz eher in größeren Projekten mit langen Transaktionsketten zwischen den Microservices und wurde in der vorliegenden Arbeit nicht verwendet.

4 SPRING FRAMEWORK

4.1 EINFÜHRUNG IN SPRING

Das Spring Framework[1] für die Java-Entwicklung wurde zunächst als Alternative zur Java Enterprise Edition (JEE) geschaffen, einer Architektur zum Erstellen von Webanwendungen und verteilten Systemen. Besonders in ihren frühen Versionen[2] waren der Aufbau und die technischen Komponenten der JEE sehr komplex, was bei zunehmender Größe eines Projekts zu fehlender Übersichtlichkeit und Transparenz führte. Dies wiederum erhöhte die Fehleranfälligkeit entsprechender Softwaresysteme. Ein weiterer Kritikpunkt war die Schwergewichtigkeit der Servicekomponenten, den sogenannten Enterprise Java Beans (EJBs). Für die Umsetzung einzelner Funktionalitäten wie beispielsweise der Abbildung eines Kontos waren mehrere Artefakte notwendig, was nicht zuletzt auch Einfluss auf die Projektkosten nahm [Ie11, vgl. S.10f].

Das Spring Framework ist ein Open Source Framework[3] von Pivotal Software zur Vereinfachung der Entwicklung von JEE-Software. Durch einen modularen Aufbau und die Verwendung einfacher und konsistenter Programmierschnittstellen senkt das Spring Framework im Vergleich zu JEE die Komplexität von Softwareprojekten und mindert den Einsatz von Boilerplate-Code[4].

Abbildung 4: Spring Logo [spr17a]

1 Ein Framework besteht aus einer Sammlung von Bibliotheken, die bestimmte Aufgaben komplett oder teilweise lösen. Durch den Einsatz von Frameworks kann eine große Zeitersparnis in der Entwicklung erreicht werden [HRW13, vgl. S.189].
2 bis Version 2.1, ab Version 3.0 wurde der Aufbau verbessert.
3 Lizenziert unter der Apache-Lizenz 2.0
4 Codebausteine, die im Programm mehrfach ohne bzw. mit wenig Änderungen wiederverwendet werden.

Im Kern basiert das Spring auf drei Prinzipien: vereinheitlichten Schnittstellen, Dependency Injection und AOP. In Java sind im Laufe der Zeit viele Application Programming Interfaces (APIs)[5] entwickelt worden, welche auf verschiedenen Konzepten basieren, was deren Einsatz in Projekten verkompliziert. Spring stellt eine vordefinierte API-Schicht bereit, welche die Verwendung solcher Schnittstellen harmonisiert und somit vereinfacht [Wol11, vgl. S.2].

Ein großes Problem in komplexen Softwarearchitekturen stellt die Verwaltung von Objekten und deren Beziehungen untereinander dar. Insbesondere beim Einsatz von internen Services, wie sie auch in der vorliegenden Arbeit verwendet werden, kann es bei Änderungen in einzelnen Klassen zu weitreichenden Folgen kommen. Spring löst dieses Problem über Dependency Injection (DI), bei welchem Objekt-Referenzen an den jeweils benötigten Stellen automatisch von Spring induziert werden. Dies führt zu einer losen Kopplung der Komponenten in der Software, die sich nicht mehr selbst um die Erzeugung oder Suche von Referenzen auf andere Objekte kümmern müssen. [VB15, vgl. S.16]. Dies senkt die Abhängigkeit der einzelnen Objekte von ihrer Umgebung und ermöglicht einen flexiblen Einsatz in verschiedenen Bereichen wie Applikationsservern. Durch DI ist es beispielsweise möglich, angebundene Datenbanksysteme ohne große Änderungen am Code auszutauschen oder in Testsituationen gezielt Testobjekte in das Programm einzuspeisen [Wol11, vgl. S.2].

Als drittes Element unterstützt Spring die Aspektorientierte Programmierung (AOP). Als Grundprinzip verbirgt sich dahinter die Identifikation und Separierung sogenannter Cross-Cutting Concerns[6]. Typische Cross-Cutting Concerns stellen Fehlerhandling, Sicherheitschecks oder Logging dar. Bei Anwendung von AOP werden diese Anforderungen in separate Module gekapselt, den sogenannten Aspekten. Dies ermöglicht eine erleichterte Wiederverwendung von Code und unterstützt die zentrale Entwicklung solcher Module [Gor11, vgl. S.186].

Abbildung 5 gewährt einen Überblick über die etwa 20 Module umfassende Struktur des Frameworks. Die einzelnen Module sind grundsätzlich voneinander getrennt, was eine leichtgewichtige Entwicklung begünstigt, da eine Einbindung des kompletten Frameworks inkl. aller Module nicht notwendig ist. Ein detaillierterer Überblick über die Kernelemente des Frameworks findet sich in der offiziellen Dokumentation [Je14]. Aufbauend auf dem ursprünglichen Spring Framework sind diverse andere Projekte im Umfeld von Spring entstanden, die ebenfalls modular aufgebaut sind und dem jeweiligen

5 deutsch: Programmierschnittstellen
6 Anforderungen, die sich über mehrere Anwendungsschichten erstrecken

Projekt entsprechend unabhängig voneinander oder gemeinsam eingesetzt werden können. Spring setzt dabei auf einen gering invasiven Ansatz, d.h. der vom Anwender geschriebene Code weist nur geringe Abhängigkeiten mit den jeweils verwendeten Modulen aufs. Dadurch kann sich verstärkt auf die Implementierung der eigenen Geschäftslogik konzentriert werden anstatt auf die Einhaltung der durch APIs vorgegebene Konventionen [Sta11, vgl. S.2f].

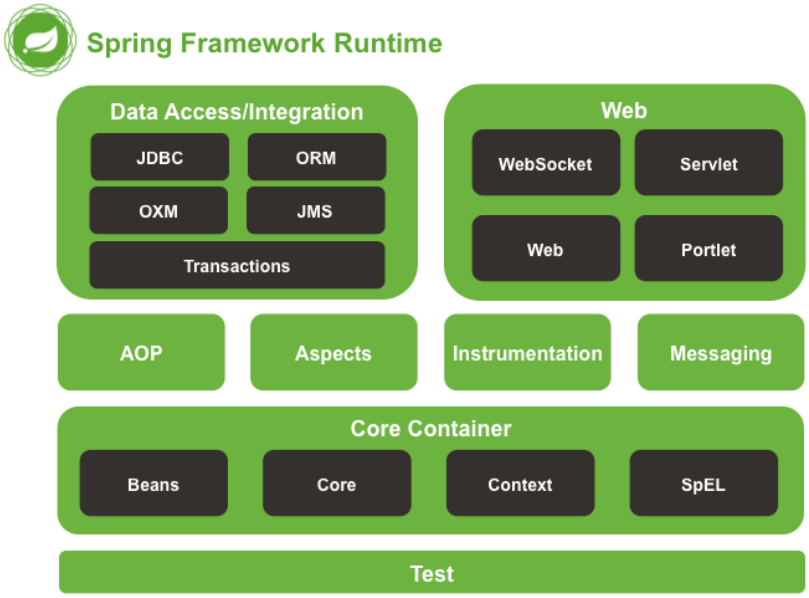

Abbildung 5: Überblick über die Bestandteile des Spring Frameworks [Je14]

Im Folgenden werden die zwei Hauptbestandteile des im Rahmen dieser Arbeit entstandenen Microservices näher vorgestellt. Dabei handelt es sich um die Spring Projekte Spring Boot sowie Spring Web MVC.

4.2 SPRING BOOT

Spring Boot stellt den Kern der Anwendung dar und ermöglicht die schnelle Entwicklung lauffähiger Software basierend auf Spring. Von Herstellerseite aus wird das Spring-Projekt folgendermaßen charakterisiert:

„Spring Boot makes it easy to create stand-alone, production-grade Spring based Applications that you can 'just run'."[7] [spr17b]

7 Deutsch etwa: Spring Boot erleichtert die Entwicklung von produktionsreifen Standalone-Anwendungen, basierend auf Spring und bereit zur sofortigen Ausführung.

Abbildung 6: Spring Boot Logo [o.V17a]

Ein großer Vorteil von Spring Boot gegenüber der herkömmlichen Entwicklung mit dem Spring Framework ist die Möglichkeit der Auto-Konfiguration. Üblicherweise werden Spring Anwendungen über XML-Dateien oder spezielle Java-Klassen konfiguriert. Die Festlegung verschiedener Konfigurationen über ein Projekt hinweg benötigt in herkömmlichen Spring Anwendungen viel Zeit und erzeugt häufig Boilerplate-Konfigurationen. Spring Boot setzt stattdessen auf Convention over Configuration[8], d.h. solange man sich an spezielle Konventionen des Frameworks hält, übernimmt Spring Boot die Konfiguration, sodass der Anwender keine zusätzlichen Einstellungen vornehmen muss [Mül17, vgl.].

Durch die Verwendung sogenannter Starter Dependencies[9] wird die Auswahl passender Bibliotheken stark vereinfacht. Spring stellt hierzu verschiedene Pakete bereit, die über Building-Tools wie Maven oder Gradle eingebunden werden können und die eine Grundausstattung an Bibliotheken mitbringen. Beispielsweise liefert das in der vorliegenden Arbeit verwendete Paket Spring Boot Starter Web insgesamt sechs Dependencies[10] zur Erstellung von unter anderem REST-Services. In diesen Paketen sind die einzelnen Versionen der Bestandteile aufeinander abgestimmt, sodass eine reibungslose Funktion untereinander gewährleistet wird. Grundsätzlich ist die Verwendung der Starter Dependencies freigestellt, sodass die Bibliotheken bei Bedarf auf manuell ausgewählt werden können [Wal16, vgl. S.5f.].

Actuator Endpoints erlauben, über eine Abfrage per Browser oder Shell, das Monitoring und die Interaktion mit der laufenden Anwendung. Dazu stellt Spring Boot eine Reihe bereits implementierter Endpunkte bereit wie beispielsweise `/health`, mit welchem sich der aktuelle Zustand des Programms abfragen lässt. Vorgefertigte Endpunkte lassen sich konfigurieren und um eigens definierte Endpunkte ergänzen. Eine Übersicht über alle möglichen Endpunkte liefert die offizielle Spring Boot Dokumentation [Wea, vgl.].

8 Deutsch: Konvention vor Konfiguration
9 Einbindung externer Bibliotheken
10 Aktueller Build: `https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web`.

Auf Basis der genannten Kernkompetenzen von Spring Boot eignet sich das Projekt insbesondere auch für die Umsetzung von Microservices. Zur Festlegung von REST-Schnittstellen zur Kommunikation mit anderen Microservices bzw. Clients wird zusätzlich das Projekt Spring Web MVC eingesetzt.

4.3 SPRING WEB MVC

Spring Web MVC bzw. häufig nur Spring MVC genannt, ist von der ersten Stunde an Bestandteil des Spring Frameworks und dient zur Erstellung von Webanwendungen wie beispielsweise REST-Services. Der Name leitet sich vom Entwurfsmuster Model View Controller (MVC) ab, an welchem sich die Aufteilung der Komponenten von Spring MVC orientiert.

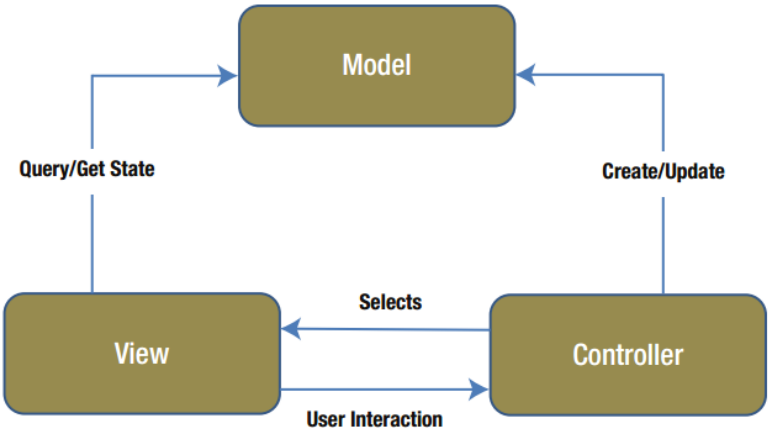

Abbildung 7: Model View Controller Schema [VB15, vgl. S.18]

Abbildung 7 zeigt die Untergliederung von Software gemäß MVC in die drei Bestandteile Model, View und Controller. Das Model repräsentiert die Daten bzw. den Status eines Programms. Dazu gehören im aktuellen Projekt die Profildaten der Anwender sowie die intern anfallenden Daten. Der Controller kümmert sich um die Umsetzung von Nutzeranfragen wie sie über REST kommuniziert werden. Diese Anfragen werden anschließend an interne Services übergeben zur Bearbeitung der vorliegenden Daten. Views stellen die grafische Repräsentation des Datenmodells dar, beispielsweise über eine Weboberfläche [VB15, vgl. S.18]. Der Bestandteil View wird in der Realisierung der Serveranwendung nicht weiter verwendet, da der Microservice keine grafische Oberfläche bieten wird[11].

11 Ein möglicher Ersatz bietet die Dokumentation über Swagger UI, siehe Kapitel 6.10.

Der strukturelle Aufbau eines REST-basierten Services, umgesetzt mithilfe von Spring MVC, ist in Abbildung 8 dargestellt. Die HTTP-Anfrage eines Clients stellt stets den Beginn einer Transaktion dar. Diese Anfragen werden von einem Front-Controller, dem sogenannten DispatcherServlet, abgefangen. Über Handler Mapping wird der Anfrage eines Clients entsprechend ein Controller aufgerufen, welcher die Anfrage bearbeitet. Hierzu kann der Controller auf interne oder externe Services zurückgreifen, insbesondere auch auf andere Microservices. Liegt ein Ergebnis der Transaktion vor, so wird dieses zurück an den Client gesendet, der die Antwort anschließend lokal auswerten kann [o.V17i, vgl.].

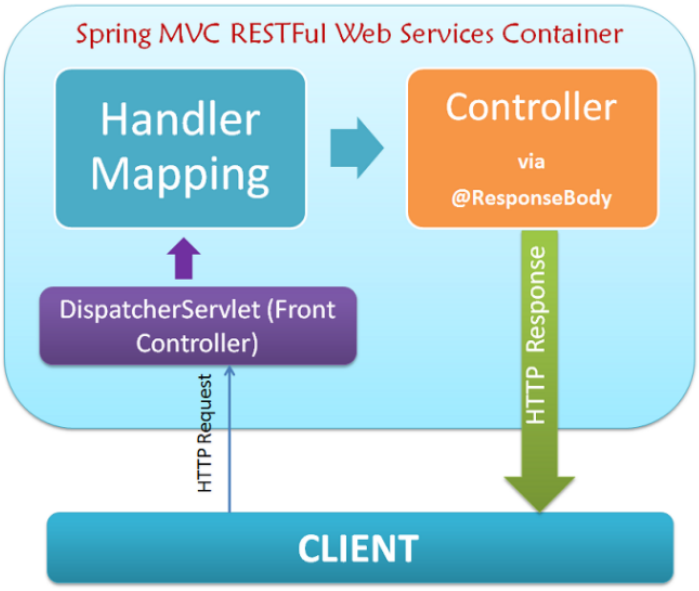

Abbildung 8: Spring Web MVC Architektur [Dar14]

Spring MVC liefert alle notwendigen Funktionalitäten zur Realisierung eines REST-Services. Durch die Verwendung von Spring Boot als Basis entfallen dank Autokonfiguration viele der ansonsten notwendigen Konfigurationen. Spring MVC ist unter anderem als Starter Dependency verfügbar[12] und ermöglicht somit einen schnellen Einstieg in die Entwicklung.

Mithilfe des Spring Frameworks können die grundlegenden Anforderungen an den Microservice umgesetzt werden, eine permanente Speicherung der Nutzerdaten erfordert zusätzlich eine weitere Systemkomponente, ein Datenbanksystem. Das nächste Kapitel nimmt einen Vergleich verfügbarer Technologien vor und geht näher auf das ausgewählte Datenbanksystem CouchDB ein.

12 Verfügbar unter `https://mvnrepository.com/artifact/org.springframework.boot/spring-boot-starter-web`.

5 DATENBANKSYSTEME

5.1 RELATIONALE DATENBANKEN

Die Aufgabe der entstandenen Serveranwendung ist grundsätzlich die Verwaltung von Nutzereinstellungen in Form von Strings. Dies erfordert neben der entsprechenden Bearbeitung von Anfragen eine persistente[1] Datenhaltung. Dafür wird ein Datenbanksystem[2] benötigt, welches für die Beschreibung, Speicherung und Verwaltung von Daten zuständig ist. Den Kern einer Datenbank bildet das sogenannte Datenbankmanagementsystem (DBMS), welches die grundlegende Funktionalität bereitstellt. Hierzu gehören das Suchen, Lesen und Schreiben von Daten sowie die Definition von Schnittstellen, welche einen externen Datenzugriff ermöglichen. Mithilfe einer Datenbanksprache kann der Anwender über die Schnittstellen des DBMS auf die gespeicherten Daten zugreifen [Ste14, vgl. S.6f].

Im Modell der relationalen Datenbanken wird zur Abfrage die Datenbanksprache Structured Query Language (SQL) verwendet, weshalb häufig auch von SQL-Datenbanken gesprochen wird. Das seit 1970 existierende Modell entspricht einer Menge von Tabellen, in welchen die Daten in vordefinierten Kategorien abgespeichert werden. Jede Kategorie ist in einer Spalte dargestellt, eine Tabelle kann stets mehrere solcher Spalten beinhalten. Jede Zeile einer Tabelle entspricht einem Datensatz und ist durch einen einzigartigen Bestandteil der Daten, dem Schlüssel, eindeutig identifizierbar. Aus der Datenhaltung ergeben sich die formalen Anforderungen der atomicity, consistency, isolation und durability, den ACID-Eigenschaften[3]. Relationale Datenbanken sind besonders für strukturierte Daten, d.h. Daten mit stets gleichem Format, geeignet [Lea10, vgl. S.12].

1 Langfristige Datenhaltung auf einem Speichermedium zur späteren Weiterverwendung [Kle16, vgl. S.2].

2 In der vorliegenden Arbeit werden die Begriffe Datenbank und Datenbanksystem synonym verwendet.

3 deutsch: Atomarität (Abgeschlossenheit), Konsistenzhaltung, Isolation und Dauerhaftigkeit. Daraus ergibt sich, dass eine Transaktion entweder komplett oder gar nicht ausgeführt wird (atomicity), jede Transaktion die Datenbank in einen zulässigen Zustand überführt (consistency), die einzelnen Transaktionen sich gegenseitig nicht beeinflussen können (isolation) und die Wirkung abgeschlossener Transaktionen dauerhaft sind (durability) [SSH13, vgl. S.388].

Grenzen im Einsatz ergeben sich bei der horizontalen Skalierung, d.h. bei der Verteilung von einzelnen Datenbanken auf mehrere Server, da es zu Problemen beim Joinen von Tabellen[4] kommen kann. Ein weiteres Problem stellt die Verarbeitung komplexer Datenstrukturen dar, die sich nicht in einzelnen Tabellen strukturieren lassen. Ebenso ist SQL auf die Abfrage strukturierter Daten ausgelegt, was problematisch bei der Arbeit mit unstrukturierten Daten, insbesondere bei Verwendung verschiedener Datenformate im selben Datensatz, ist [Lea10, vgl. S.12].

5.2 NOSQL-DATENBANKEN

In den letzten Jahren wurde eine Reihe alternativer Konzepte zu relationalen Datenbanken entwickelt, auch im Hinblick auf die wachsenden Anforderungen aus Big Data und Web 2.0. Viele dieser neuen Systeme werden unter dem Begriff Not Only SQL (NoSQL) geführt. Ein zentraler Aspekt solcher Datenbanksysteme ist die Möglichkeit einer einfachen horizontalen Skalierung sowie die Replikation und Verteilung von Daten auf mehrere Server. Zur Datenabfrage werden je nach Datenbanklösung verschiedene Schnittstellen oder Protokolle verwendet, die in der Regel eine geringere Komplexität als SQL aufweisen und dadurch schneller praktisch eingesetzt werden können. Zur parallelen Verarbeitung einer großen Anzahl an Anfragen und Daten sowie einer entsprechend kurzen Antwortzeit setzen NoSQL-Datenbanken häufig auf eine effiziente Indexierung[5] und Nutzung des Arbeitsspeichers. Hierzu wird zum Teil auch von ACID-Transaktionen, wie sie in SQL-Datenbanken in der Regel festgelegt sind, abgewichen. Im laufenden Einsatz der Datenbanksysteme können dynamisch neue Attribute zu einzelnen Datensätzen hinzugefügt sowie die Struktur von Daten angepasst werden, bei relationalen Datenbanken ist dies in der Regel nicht ohne weiteres möglich [Cat11, vgl. S.1].

Neben den genannten Vorteilen bestehen auch bei NoSQL-Datenbanken diverse Einschränkungen. Durch den Verzicht auf SQL als Abfragesprache und den Einsatz individueller Lösungen steigt die Komplexität größerer Abfragen an, was sich negativ auf die Laufzeit auswirkt. Auch die fehlenden ACID-Eigenschaften senken die Verlässlichkeit des Systems und können zu Problemen bei der Konsistenz der Daten führen. Im Einsatz von NoSQL-Lösungen können ACID-Eigenschaften zusätzlich manuell hinzugefügt werden, was wie bei

4 Zusammenfügen von Daten aus verschiedenen Tabellen

5 Erstellung von Indizes, die verwendet werden, um die Suche nach Datensätzen zu beschleunigen [ENS03, vgl. S.185].

relationalen Datenbanken jedoch zu Einbußen bei Performance und Skalierbarkeit führt. Im Unternehmensbereich ist meist ein breites Wissen über den Einsatz von SQL und zugehöriger, relationaler Datenbanken vorhanden, die neueren Technologien sind meist noch unbekannt und erfordern eine Schulung der Mitarbeiter. Auch fehlen Open-Source-Anwendungen häufig ein Kundensupport sowie Verwaltungstools, was den Einsatz im Tagesgeschäft zumindest in der Anfangszeit erschwert [Lea10, vgl. S.14].

Für den Zweck der Speicherung von Nutzerdaten wurde sich trotz der genannten Nachteile für den Einsatz einer NoSQL-Datenbank entschieden. Ein wichtiger Aspekt für die Entscheidung stellt die bessere Unterstützung einer agilen Softwareentwicklung dar, da flexibler auf sich ändernde Anforderungen hinsichtlich der Datenstruktur reagiert werden kann. Auch für die Anforderung der Verwendung von Open-Source-Software unter Apache-Lizenz 2.0 bietet der Markt von NoSQL-Systemen ein größeres Angebot im Gegensatz zu relationalen Datenbanken.

5.3 VERGLEICH VERFÜGBARER NOSQL-LÖSUNGEN

Im Bereich der NoSQL-Datenbanken wird grundsätzlich zwischen folgenden vier Klassen differenziert:

- Key-Value: Grundlage sind Schlüssel-Wert-Paare, d.h. über definierte Schlüssel kann auf einzelne Daten zugegriffen werden. Daten können in Form einfacher Datentypen, Strings oder komplexeren Strukturen wie Listen oder Sets vorliegen, je nach verwendetem Datenbanksystem. Key-Value-Datenbanken zeichnen sich durch eine schnelle und effiziente Datenverwaltung aufgrund des einfachen Datenmodells aus. Der Nachteil dieser Vereinfachung ist die fehlende Möglichkeit, komplexe Abfragen zu generieren, ohne sich dabei auf entsprechend vordefinierte Schnittstellen zu verlassen [Edl11, vgl. S.7].

- Document Stores: Gespeichert werden Daten in strukturierten Datensammlungen wie z.B. JSON-Dokumenten. Jedes Dokument ist über eine eindeutige ID ansprechbar. Die Strukturen der einzelnen Dokumente unterliegen dabei kaum Einschränkungen von Seiten des Datenbanksystems, sodass hier der Aufbau flexibel gestaltet werden kann [Edl11, vgl. S.8]. Im Gegensatz zu Key-Value-Systemen bieten Document Stores somit die Möglichkeit, komplexere Daten abzubilden, da auch eingebettete Dateien oder Dokumente im Bereich des Möglichen liegen. Es wird jedoch auch wie bei anderen NoSQL-Systemen meist auf ACID-Eigenschaften bei Transaktionen verzichtet [Cat11, vgl. S.16f].

- Spaltenorientiert: Die Daten werden als Schlüssel-Wert-Relation in Spalten abgelegt, wobei einzelne Spalten separat gespeichert werden. Bei Abfragen werden nur jene Spalten gelesen, die auch für die Abfrage benötigt werden. Dies ermöglicht eine bessere Kompression und erhöht den Durchsatz bei I/O-Operationen[6] [He11, vgl. S.364].
- Graphdatenbanken: Zur Speicherung von Daten werden Konzepte der Graphentheorie[7] verwendet, bei welchen die gespeicherten Elemente untereinander verknüpft sind. Dies ermöglicht eine Nachbildung realer Beziehungen der Inhalte wie sie beispielsweise in sozialen Netzwerken eine Rolle spielen. Ein Vorteil der Systeme ist die schnelle Traversierung von Relationen einzelner Elemente [Edl11, vgl. S.8f.].

Für die vorliegenden Anforderungen eignen sich Key-Value Datenbanken weniger, da sie nur eingeschränkte Abfragen zulassen und somit die Umsetzung komplexer Anwendungsfälle, wie sie im Laufe der Entwicklung zu erwarten sind, erschweren. Auch Graphdatenbanken sind aufgrund der Verknüpfung der Daten untereinander ungeeignet, da die einzelnen zu speichernden Profile voneinander unabhängig sind und eine künstliche Abhängigkeit keinen sinnvollen Mehrwert bieten würde.

Zwischen den verbleibenden Systemen der Document Stores und der Spaltenorientierten Datenbanken wurde sich für den Einsatz eines Document Stores entschieden, da diese eine relativ freie Gestaltung der abzuspeichernden Daten gewährleisten und somit auch bei nachträglichen Änderungen im Datenformat flexibel angepasst werden können. Weiterhin bietet sich die Verwendung in Kombination mit Spring MVC an, da das Framework bereits von Haus aus Unterstützung von häufig verwendeten Dokumentformaten wie JSON mitbringt.

Bei der Suche nach passenden Document Stores wurde zunächst Wert auf eine Bereitstellung unter Apache-Lizenz 2.0 gelegt. Des Weiteren wurde darauf geachtet, dass die Systeme eine gewisse Entwicklungsreife besitzen, sodass ein zuverlässiger Betrieb gewährleistet ist. Gerade im NoSQL-Bereich gibt es eine Vielzahl an kleineren und größeren Projekten mit großem Potenzial, deren Entwicklung sich aber noch in einem experimentellen Stadium befindet. Eine ausführliche Dokumentation sowie das Vorhandensein von entsprechender Fachliteratur stellt eine zusätzlich wünschenswerte Eigenschaft dar.

6 Kurzform für Input/Ouput, bezeichnet Interaktionen, welche die Ein- und Ausgabe von Daten umfassen.

7 Ein Graph besteht aus Knoten, welche die Speicherinhalte repräsentieren, sowie Kanten, welche die Verbindung der einzelnen Knoten untereinander beschreiben.

Nach ausführlicher Recherche haben sich insbesondere die beiden Systeme Couchbase sowie CouchDB als den Anforderungen entsprechend herauskristallisiert. Couchbase bildete hierbei zunächst die erste Wahl, unter anderem auch durch eine native Anbindung an das Spring Framework, was eine Kommunikation mit dem Microservice stark vereinfacht hätte. Couchbase bietet jedoch lediglich als Open-Source-Edition eine Nutzung unter Apache-Lizenz 2.0 an, diese Variante ist jedoch als experimentell deklariert und kommt ohne Support oder Garantien daher [o.V17g, vgl.]. Dies ist unter den Ansprüchen des Projekts nicht tragbar. Daher wurde sich schließlich für den Einsatz von CouchDB entschieden, welche im folgenden Abschnitt näher beleuchtet wird.

5.4 APACHE COUCHDB

Das Projekt CouchDB ist 2005 von Damien Katz ins Leben gerufen worden und seit 2008 ein Projekt der Apache Software Foundation, woraus auch die Bereitstellung unter Apache-Lizenz 2.0 resultiert. Die Software ist in der funktionalen Programmiersprache Erlang geschrieben, die eine hohe Parallelität und Zuverlässigkeit der Software verspricht. Im Kern ist CouchDB ein Document Store und verzichtet auf vordefinierte Schemata der Dokumente. Der Zugriff auf die Datenbank erfolgt über eine REST-Schnittstelle, sodass bei der Entwicklung unabhängig von vordefinierten APIs für einzelne Programmiersprachen gearbeitet werden kann. Als Dokumentenformat wird JSON verwendet, sodass auch komplexere Datenstrukturen wie Listen oder verschachtelte Dokumente gespeichert werden können. Ein Limit für die Größe oder Anzahl von Texten oder Elementen wird von CouchDB nicht gesetzt. Daneben stellt CouchDB unter dem Titel Project Fauxton[8] eine grafische Weboberfläche zur Verwaltung des Systems bereit [o.V17b, vgl.].

Abbildung 9: CouchDB Logo [o.V16]

8 Bei default-Einstellungen im laufenden Betrieb lokal erreichbar unter: `http://127.0.0.1:5984/_utils/`

Das Datei- und Commitment-System berücksichtigt alle ACID-Eigenschaften, was eine hohe Zuverlässigkeit verspricht. Das Update-Modell von CouchDB arbeitet optimistisch[9] und ohne Locking[10] und setzt stattdessen auf den Einsatz von Multiversion Concurrency Control (MVCC). Hierbei werden alle Dokumente versioniert und erhalten bei Schreibvorgängen jeweils eine neue Revision. Dadurch arbeiten Anwender stets mit Momentaufnahmen, sogenannte Snapshots, von Dokumenten und müssen bei Schreibprozessen jeweils die von ihnen bearbeitete Revision angeben. Kommt es nun zu Konflikten bei parallelen Schreibprozessen im selben Dokument, erhält der spätere Prozess eine Konfliktmeldung, weil er nicht mehr die aktuellste Revision bearbeitet [o.V17b, vgl.].

Die Indexierung erfolgt in B-Bäumen[11] über eine für jedes Dokument individuelle ID. Aufgrund der Verwendung von Dokumenten als Speicherformat liegen zusammengehörende Daten gebündelt an einer Stelle im Speicher. Um ein einzelnes Dokument zu aktualisieren muss jeweils ein vollständiges Dokument übermittelt werden. Dies verursacht zunächst einmal mehr Traffic über das Netz, stellt aufgrund der geringen Anzahl an notwendigen Informationen zur Synchronisierung der Nutzerprofile keinen merklichen Nachteil in der Verwendung dar [o.V17b, vgl.].

Um Speicherplatz zu sparen bietet CouchDB die Möglichkeit zur Verdichtung. Dabei werden alle aktiven Daten in eine neue Datei kopiert und die alte Datei anschließend gelöscht. Während des kompletten Prozesses bleibt die Datenbank erreichbar und kann weiterhin Lese- und Schreibprozesse ausführen.

Mithilfe von JavaScript und MapReduce[12] können sogenannte Views erzeugt werden, welche einen gezielten Abruf von Informationen aus Dokumenten der Datenbank ermöglicht. Views werden in sogenannten Design Documents definiert und werden dynamisch erzeugt. Dabei wird mit Kopien der einzelnen Dokumente gearbeitet, sodass diese vollständig unabhängig von Views sind und sich auch während Update-Prozessen oder der Verdichtung der Datenbank abrufen lassen [o.V17b, vgl.].

9 Gültigkeit einer Transaktion wird erst beim Schreibprozess geprüft.
10 Jeder Anwender kann unabhängig von anderen Prozessen auf Dokumente zugreifen.
11 Ausgeglichene Baumstruktur, die eine mehrstufige Zugriffshierarchie zur Suche von Daten anbietet [ENS03, vgl. S.199ff.].
12 Konzept zur Datenabfrage. Zunächst werden Daten nach vordefinierten Kriterien aus der Datenbank ausgelesen (Map-Schritt) und anschließend aggregiert (Reduce-Schritt) [WK11, vgl. S.37ff].

Eine Replikation der Datenbank kann über CouchDB Replication umgesetzt werden, was die Synchronisation von mehreren Systemen zum Aufbau redundanter Datenbanken ermöglicht. Die Kommunikation der einzelnen Datenbanken zur Synchronisation untereinander erfolgt ebenso über REST [o.V17c, vgl.]. Mittlerweile ist auch eine horizontale Skalierung, sogenanntes Sharding, durch Verteilung einer Datenbank auf mehreren Servern möglich. Zusammen mit der Replikation können verteilte Systeme mit geringer Ausfallwahrscheinlichkeit aufgebaut werden [New16, vgl.].

Die realisierte Anbindung von CouchDB an den Microservice wird im nachfolgenden Kapitel im Abschnitt 6.7 beschrieben. Die Anbindung erfolgt hierbei ausschließlich über die Verwendung der REST-Schnittstelle.

6 SOFTWARELÖSUNG

6.1 VERWENDETE SOFTWARE

Im folgenden findet sich eine Auflistung der für die Umsetzung verwendeten Programme und Frameworks[1]:

- Java Development Kit (JDK): Compiler, Laufzeitumgebung, Debugger und Dokumentationswerkzeuge für die Entwicklung mit Java. Verwendete Version: 1.8.0.
- Eclipse Java EE IDE for Web Developers: Integrierte Entwicklungsumgebung für die Entwicklung mit Java. Verwendete Version: 4.7.0 Oxygen.
- Spring Boot: Framework zur Realisierung des Microservices, siehe Kapitel 4.2. Verwendete Version: 1.5.6.
- Springfox: Framework zur Umsetzung einer Dokumentation der Schnittstellen über Swagger, siehe Abschnitt 6.10. Verwendete Version: 2.7.0.
- Apache Maven: Build-Management-Tool zur Verwaltung von Java Software Projekten. Verwendete Version: 3.5.0.
- Apache CouchDB: Dokumentenorientiertes NoSQL-Datenbanksystem, siehe Kapitel 5.4. Verwendete Version: 2.1.1.
- Postman: Programm zum Testen von HTTP-Schnittstellen. Verwendete Version: 5.3.2.
- Github Desktop: Desktop-Anwendung zur Arbeit mit GitHub und Versionsverwaltung. Verwendete Version: 1.0.10.

1 Unvollständige Übersicht: Kleinere Frameworks wurden ausgenommen. Verwendete Versionsnummern beziehen sich auf den Stand während der Erstellung dieser Arbeit.

6.2 REPLIKATION UND SYNCHRONISATION

Vor der Implementierung der Software ist zunächst eine Strategie hinsichtlich der Art der Replikation und Synchronisation festzulegen. Eine Replikation von Daten führt zu einer Datenredundanz und entspricht dem erstmaligen Pull-Vorgang eines Clients. Über diesen erhält das Gerät eine vollständige Kopie des Profils aus der Datenbank [MS04, vgl. S.79]. Sind auf dem Gerät bereits Daten vorhanden, so wird das Konzept der Synchronisation angewandt.

Im Bereich der Synchronisation in verteilten Anwendungen unterteilt man den Synchronisationsprozess in zwei Phasen: Reintegration und Rückübertragung. Reintegration wird bei einem Abgleich von Änderungen auf einer Replikationssenke (Clientgerät) durchgeführt. Werden auf dem Gerät Änderungen an den Daten durchgeführt, so müssen diese auf der Replikationsquelle (Datenbank) eingebracht bzw. reintegriert werden [MS04, vgl. S.86f.]. Dies entspricht einem Push-Vorgang (UC 3) nach Anpassungen des Profils einem Clientgerät. In Abbildung 10 ist dieser Vorgang im linken Anwendungsfall skizziert.

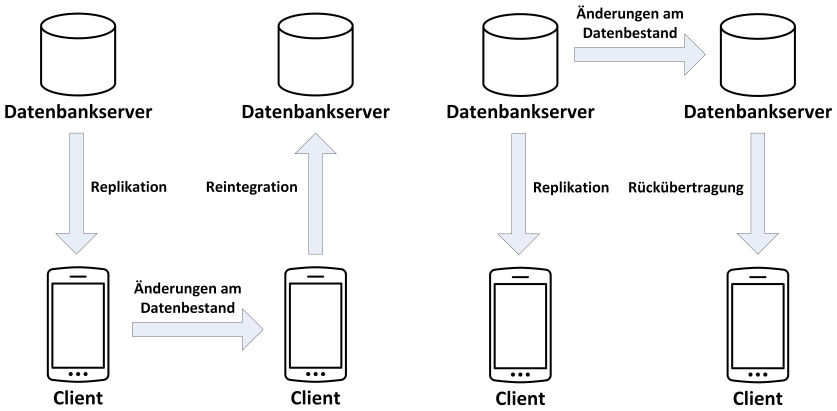

Abbildung 10: Reintegration und Rückübertragung [Welo9, vgl. S.41]

Eine Rückübertragung findet statt, wenn von der Replikationsquelle ein aktueller und konfliktbereinigter Datenzustand an die Replikationssenken übertragen wird [MS04, vgl. S.86f.]. Dieser Fall tritt ein, nachdem der Anwender über ein Endgerät Anpassungen an seinem Profil vorgenommen hat und dieses im Zuge einer Reintegration auf den Server übertragen hat. Daraufhin folgt eine Rückübertragung der neuen Daten auf die anderen Endgeräte des Anwenders, umgesetzt durch eine Pull-Anfrage der Clientgeräte (UC 4), um einen geräteübergreifend konsistenten Datensatz zu gewährleisten. Dies entspricht dem rechten Anwendungsfall in Abbildung 10. In der Realisierung des Microservice finden Reintegration und Rückübertragung unabhängig voneinander auf Anfrage eines Clients statt.

Man spricht hierbei auch von einem unidirektionalen Synchronisationsprozess [MS04, vgl. S.87]. Da der Synchronisationsprozess von den Clientgeräten angestoßen wird, kann eine konsistente Datenlage vonseiten der Serveranwendung nicht garantiert werden. Die Endgeräte sind demnach in der Pflicht, regelmäßig ihre Daten auf Aktualität zu prüfen bzw. Änderungen zeitnah an den Server zu übertragen.

6.3 SOFTWARE-ARCHITEKTUR DES MICROSERVICES

Der interne Aufbau des Microservice orientiert sich an der Software-Architektur der Schichtenbildung, dem sogenannten Layering. Hierbei stellt jede Schicht eine Art virtuelle Maschine dar, die den darüberliegenden Schichten einzelne Dienste anbietet. Die Implementierung jeder Schicht ist nach außen hin verborgen. Dabei ist es möglich in einzelnen Schichten komplexe Subsysteme aufzubauen bei gleichzeitiger Kapselung der technischen Aspekte. Wie beim Aufbau einer Zwiebel verbirgt jede Schicht die darunterliegen Schichten. Ein Zugriff ist nur von außen nach innen möglich, eine tieferliegende Schicht kann in der Regel nicht auf höhere Schichten zugreifen. Bei wechselseitigen Beziehungen zwischen Komponenten sind diese möglichst in derselben Schicht zu anzusetzen. Das Durchreichen von Aufrufen durch alle Schichten kann zu einer schlechteren Performance führen, weshalb es möglich ist, einzelne Schichten zu überspringen, sogenanntes Layer-Bridging [Sta11, vgl. S.144ff].

Die praktische Umsetzung des Layering ist in Abbildung 11 skizziert. Grundsätzlich untergliedert sich der Aufbau in drei Schichten (three tier architecture). Als oberste Schicht steht der Front Controller, welcher sich nochmals in drei Controller untergliedert, welche im Abschnitt 6.5 näher beleuchtet werden. Zugriff auf diese Ebene haben grundsätzlich alle Clientgeräte, die mit REST arbeiten können. Administratoren erhalten über eine passwortgeschützte Verbindung (vgl. 6.9.2) zusätzliche Möglichkeiten der Interaktion. Auf derselben Ebene, aber unabhängig von der Front Controller Schicht, befindet sich der Scheduler, welcher zeitgesteuerte Aufgaben umsetzt.

In der zweiten Schicht befinden sich Services zur Erstellung neuer Profile, Verwaltung bestehender Profile sowie zum Aufräumen und zur Verdichtung der Datenbank. Weitere Details zu den Services werden in Abschnitt 6.6 erläutert.

Die innerste Schicht, die Repository Ebene, stellt Schnittstellen zu Datenbanksystemen bereit. Wie bereits in Kapitel 5 besprochen, wird CouchDB als Datenbanksystem verwendet, mit welchem die Kommunikation über REST stattfindet. Ein detaillierter Blick auf die Anbindung dieses wird in Abschnitt 6.7 geworfen.

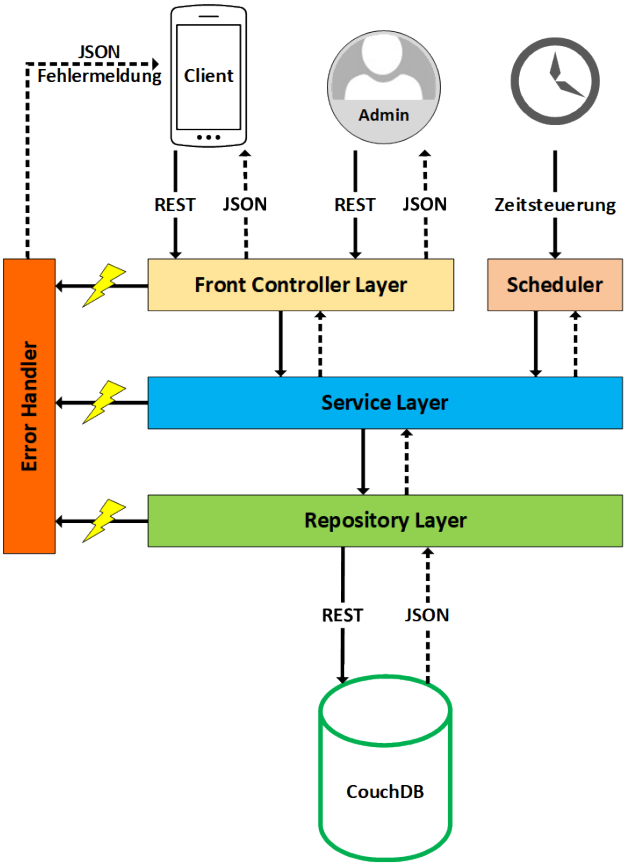

Abbildung 11: Dreischichtige Architektur des Microservice

Bei erfolgreich bearbeiteten Anfragen liefern die Front Controller JSON-Dokumente als Ergebnisse an den Aufrufer zurück. Kommt es während der Bearbeitung von Transaktionen zu Fehlern, so werden diese von einem Error Handler abgefangen und behandelt. Es handelt sich hierbei um eine Anwendung des AOP und eine Umsetzung des Cross-Cutting Concerns des Fehlerhandlings. Basierend auf den abgefangenen Fehlern werden JSON-Dokumente mit Details zum Auftreten des Fehlers erzeugt und an den Client bzw. Administrator zurückgeliefert. Die Implementierung wird in Abschnitt 6.8 besprochen.

6.4 STRUKTUR DER DATENBANK-DOKUMENTE

Für das bessere Verständnis der nachfolgenden Abschnitte sei an dieser Stelle kurz der Aufbau der JSON-Dokumente, wie sie in der CouchDB-Datenbank abgelegt werden, aufgezeigt. Jedes Dokument beinhaltet folgende Schlüssel-Werte-Paare:

```
{
        "_id": "38nky6ij6248q654",
        "_rev": "5-2fce91f7364dc97c0ea5fc46017274e5",
        "lastProfileChange": 1510906815312,
        "lastProfileContact": 1510906819032,
        "preferences": "encData"
}
```

Hierbei entspricht der Schlüssel *_id* der generierten ProfileID, worüber der Microservice und CouchDB das jeweilige Dokument eindeutig identifizieren. Über *lastProfileChange* und *lastProfileContact* werden die Zeitpunkte der letzten Profiländerung bzw. des letzten Profilzugriffs in Millisekunden seit dem 1. Januar 1970[2] festgehalten. In *preferences* schließlich liegen die gespeicherten Einstellungen in Form von Strings vor, im Fall AVARE sind diese zusätzlich vom Client verschlüsselt worden.

Die Eigenschaft *_rev* dient der internen Versionierung durch CouchDB, die in Abschnitt 5.4 besprochen wurde. Die Version entspricht der MD5-Repräsentation[3] eines Dokuments. Dieser vorangestellt ist eine fortlaufende Nummer, an welcher erkennbar ist, wie häufig das Dokument bereits geändert wurde. Die Version ist beim Update und Löschen von Dokumenten stets anzugeben. Damit wird gewährleistet, dass Änderungen nur auf Basis der aktuellsten Version eines Dokuments stattfinden [WK11, vgl. S.77].

6.5 FRONT CONTROLLER-EBENE

Die drei Java-Klassen, welche die Schicht Front Controller bilden, repräsentieren die REST-Schnittstellen, welche von außen sichtbar sind. Die Klassen sind hierzu mit der Annotation @RestController markiert, wodurch diese von Spring automatisch als Controller erkannt werden [o.V17m, vgl.]. In den Klassen selbst können einzelne Methoden mithilfe von @RequestMapping zu Handler-Methoden und somit als Endpunkte für REST-Anfragen deklariert werden [o.V17k, vgl.]. Die Basis der Implementierung bildet dabei Spring MVC (vgl. 4.3).

2 Im Microservice wird hierzu auf die Klasse Calendar zurückgegriffen, siehe: `https://docs.oracle.com/javase/8/docs/api/java/util/Calendar.html`

3 Kryptografischer Fingerabdruck, welcher der Abbildung eines beliebig langer Textes auf einen 128 Bit Hash-Wert entspricht [Ert07, vgl. S.100f].

6.5.1 *Controller zum Anlegen neuer Profile*

Der Controller, umgesetzt in der Klasse *NewProfileController*, stellt grundsätzlich die Funktionalität zur Generierung eines neuen Profils inkl. dessen eindeutiger ProfileID in der Datenbank bereit und entspricht somit dem in Kapitel 2.1 vorgestellten UC 1. Hierbei wird zwischen zwei Anwendungsfällen unterschieden: Der Generierung einer neuen ProfileID und der Verwendung einer bereits existierenden ProfileID, deren zugehöriges Profil nicht länger in der Datenbank vorhanden ist. Auf Basis dieser ProfileID wird dann ein neues Profil angelegt.

Für beide Funktionen wird ein POST-Aufruf über HTTP an die URI `/v1/newProfiles/` gesendet, wobei im Falle einer vorhandenen ProfileID diese in der URI als Parameter mitangegeben wird: `/v1/newProfiles/{profileId}`. Die Funktionalitäten der Generierung einer ID und der Erzeugung eines neuen Profils werden durch die Service- Ebene (vgl. 6.6) bereitgestellt. Abbildung 12 verbildlicht dies nochmals. Ebenso sind die HTTP-Codes bei erfolgreicher Transaktion dargestellt, die vom Client zusätzlich ausgewertet werden können.

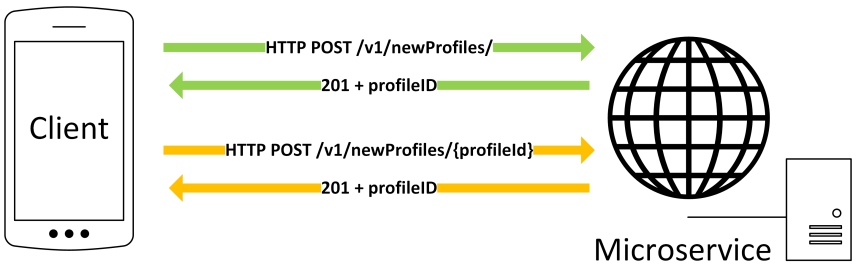

Abbildung 12: REST-Schnittstellen der Klasse NewProfileController

6.5.2 *Controller zur Interaktion mit Profilen*

Als zweiter Bestandteil des Front Controllers bietet die Klasse *ExistingProfileController* REST-Schnittstellen zur Interaktion mit bereits in der Datenbank bestehenden Profilen. In Abbildung 13 sind die verfügbaren Schnittstellen ersichtlich und werden im Folgenden der Reihenfolge nach vorgestellt.

Als erste Funktionalität wird das Pushen, d.h. abspeichern von Profilen bereitgestellt (UC 3). Hierzu ruft der Client eine URI auf, welche sowohl die ProfileID als auch das letzte Änderungsdatum des Profils auf Clientseite als Parameter enthält. Über den HTTP-Nachrichtenrumpf und die HTTP-Methode PUT wird das zu speichernde Profil in Form eines Strings übertragen.

Im Kontext von AVARE ist dieser String durch das Clientgerät bereits verschlüsselt, es kann grundsätzlich aber auch ein lesbarer Text übertragen werden. Eine Überprüfung der Aktualität der Profile wird durch den Profile Service bereitgestellt, welche in 6.6.2 beschrieben ist.

Um dem Anwender die Möglichkeit zu geben, auch nach dem Vergleich der Zeitstempel als veraltet angesehene Profile in die Datenbank zu schreiben, wird eine zweite Schnittstelle mit dem zusätzlichen Parameter overwrite bereitgestellt, über welchen in Form einer booleschen Variable festgelegt wird, ob das Profil auch unabhängig vom Zeitstempel in die Datenbank geschrieben wird. Grundsätzlich entspricht der Aufruf der ersten Schnittstelle ohne Angabe von overwrite dem Aufruf der zweiten Schnittstelle mit dem gesetzten Parameter *overwrite = false*. Zur besseren Übersichtlichkeit werden dem Anwender jedoch beide Möglichkeiten angeboten.

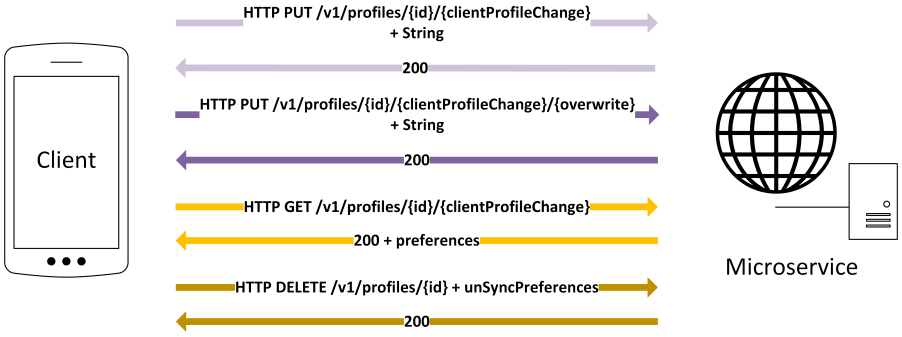

Abbildung 13: REST-Schnittstellen der Klasse ExistingProfileController

Um Profile aus der Datenbank auslesen zu können, wird über die dritte Schnittstelle die Möglichkeit eines Pulls über einen GET-Aufruf angeboten (UC 4). Auch hier werden in der URI die Parameter ProfileID und das letzte Änderungsdatum des Profils auf Clientseite erwartet und intern ausgewertet. Bei aktuellerem Datenbankprofil wird dieses entsprechend als String in einem HTTP-Nachrichtenrumpf zurückgeliefert. Für den Fall, dass der Client auch bei veraltetem Serverprofil dieses auslesen möchte, wird keine separate Schnittstelle angeboten, der Client hat jedoch die Möglichkeit, über eine entsprechende Angabe eines älteren Zeitpunktes in der URI den Vergleich der Zeitstempel zu manipulieren.

Die letzte Funktionalität entspricht dem Löschen von Profilen in der Datenbank (UC 2). In diesem Fall sendet der Client ein verschlüsseltes Profil an die entsprechende URI, woraufhin der Wert von *preferences* in der Datenbank ersetzt und der Zeitpunkt *lastProfileChange* angepasst wird. Der genaue Ablauf wird in 6.6.2 erläutert.

Eine direkte Umsetzung des Löschauftrags, d.h. ein sofortiges Entfernen des Dokuments aus der Datenbank, ist mit dieser Schnittstelle nicht gegeben. Der Grund hierfür ist die Annahme, dass mehrere Clientgeräte auf dasselbe Profil zugreifen und durch das Ablegen verschlüsselter Anweisungen als Preferences die Geräte untereinander indirekt kommunizieren können. Möchte der Endanwender sein Profil löschen lassen, so sendet er über eines seiner Geräte eine verschlüsselte Anweisung zum Ende der Synchronisierung und speichert diese im Feld *preferences*. Alle anderen Geräte, die nun lesend auf dieses Profil zugreifen, können anhand der Anweisung den Auftrag zur Beendigung der Synchronisierung auslesen. Zum besseren Verständnis wird als HTTP-Methode trotzdem DELETE verwendet und nicht PUT.

6.5.3 *Controller für Administration/ Entwicklung*

Der dritte Bestandteil des Front Controllers, die Klasse *DevController*, stellt keine Funktionalität für anonyme Clientgeräte bereit, sondern zielt auf die Anwendung während der Entwicklung sowie zur Administration des laufenden Microservices und CouchDB ab. Daher ist der Zugriff auf die REST-Schnittstelle auch über eine Basic Authentication (vgl. 6.9.2) geschützt. Anwender müssen sich dabei über eine Kombination aus Nutzernamen und Password authentifizieren. Nur wenn der Server die entsprechende Anfrage validieren kann, erhält der Anwender Zugriff auf die Schnittstellen [Sir14, vgl.]. Im vorliegenden Microservice werden die Zugangsdaten über die Datei APPLICATION.PROPERTIES festgelegt (vgl. 6.9.1).

Die Schnittstellen der Klasse lassen sich grob in zwei Bereiche unterteilen. Der erste Bereich befasst sich mit zusätzlicher Funktionalität zur Interaktion mit Profilen in der Datenbank. So existiert zum Beispiel eine Schnittstelle zur Erzeugung von Profilen mit einem weit in der Vergangenheit liegenden Zeitpunkt *lastProfileContact*, um das automatische Löschen ungenutzter Profile testen zu können (vgl. 6.6.3). Weiterhin stehen Schnittstellen bereit, um eine Liste aller Profile in der Datenbank zu erhalten sowie von einzelnen Profilen die Eigenschaften *lastProfileContact* und *lastProfileChange*. Eine weitere Schnittstelle ermöglicht das Auslesen von gespeicherten Preferences ohne Angabe eines Zeitstempels. Abbildung 14 listet alle Schnittstellen aus dem ersten Bereich auf:

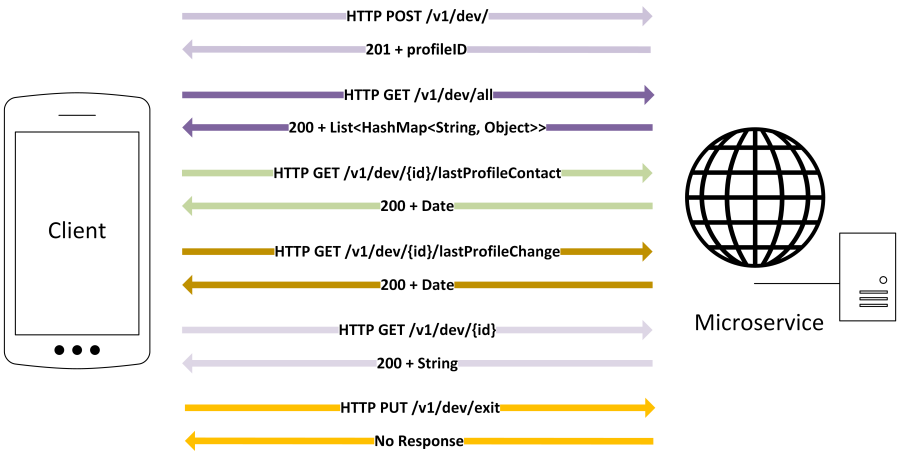

Abbildung 14: REST-Schnittstellen der Klasse DevController (1 von 2)

Der zweite Bereich ist in Abbildung 15 abgebildet und befasst sich mit der Interaktion mit dem angebundenen Datenbanksystem und ist zum Teil speziell auf die Verwendung von CouchDB ausgelegt. Der im normalen Betrieb zu fest vorgegebenen Zeiten startenden Säuberungsprozess der Datenbank lässt sich auch über *v1/dev/clean* manuell ausführen.

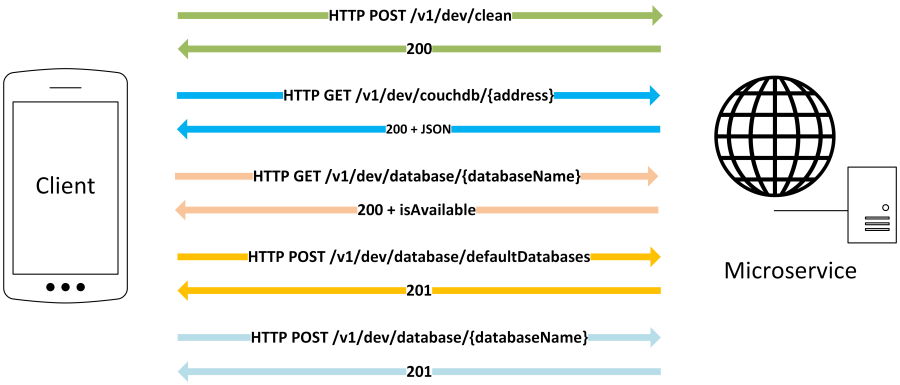

Abbildung 15: REST-Schnittstellen der Klasse DevController (2 von 2)

Für die Verwendung von CouchDB stehen zusätzlich vier Schnittstellen zur Verfügung. Die erste dient dazu zur Überprüfung, ob über die in den APPLICATION.PROPERTIES festgelegten Verbindungseinstellungen eine laufende Instanz des Datenbanksystems erreichbar ist. Die zweite Schnittstelle ermöglicht eine Überprüfung, ob eine Datenbank mit einem im Parameter festgelegten Namen bereits existiert. Bei der Verwendung von Docker (vgl. 8.1) für das Deployment einer CouchDB-Instanz fehlen vier grundsätzliche Datenbanken, die bei einer regulären Installation automatisch generiert werden. Um diese nicht von Hand erstellen zu müssen, wurde auch hierfür eine Schnittstelle implementiert.

Ebenso lassen sich zusätzliche Datenbanken über eine weitere Schnittstelle erzeugen. In Abbildung 15 sind die aufgezählten Schnittstellen abermals grafisch dargestellt, eine weitaus detailliertere Beschreibung bietet die Dokumentation über Swagger (vgl. 6.10).

6.6 SERVICE-EBENE

Die zweite Schicht, die Service-Ebene, stellt insgesamt drei Dienste bereit, welche die interne Geschäftslogik implementieren und ein Bindeglied zwischen dem Front Controller Layer und dem Repository Layer bildet. Der ID Service stellt die grundsätzliche Funktionalität zur Generierung einer neuen ProfileID bereit. Der zweite Bestandteil, der Profile Service, verarbeitet alle Anfragen bezüglich des Umgangs einzelner Profile und stellt den Kernbestandteil der Ebene dar. Der Clearance Service als dritter Dienst beinhaltet die Logik zum Aufräumen der Datenbank.

6.6.1 *ID Service*

Das Hauptaugenmerk der Klasse *IDService* liegt auf der Generierung einer individuellen ProfileID. Um den Anforderungen einer individuellen Kennung gerecht zu werden, wird eine Zeichenkette bestehend aus 10 Zahlen sowie 6 Kleinbuchstaben generiert. Abbildung 16 zeigt beispielhaft eine Generierung am 20.11.17 um 09:14 Uhr. Zunächst wird der aktuelle Serverzeitpunkt beim Aufruf der Methode ermittelt (oranger Kasten) und eine zehnstellige Zahlenkette bestehend aus dem aktuellen Tag, Monat, Jahr sowie der aktuellen Stundenzahl (24h-Format) und der Minutenzahl gebildet, wobei auf die einzelnen Zahlen jeweils eine festgelegte Konstante aufaddiert oder subtrahiert wird, um jeweils eine zweistellige Zahl zu garantieren (blauer Kasten).

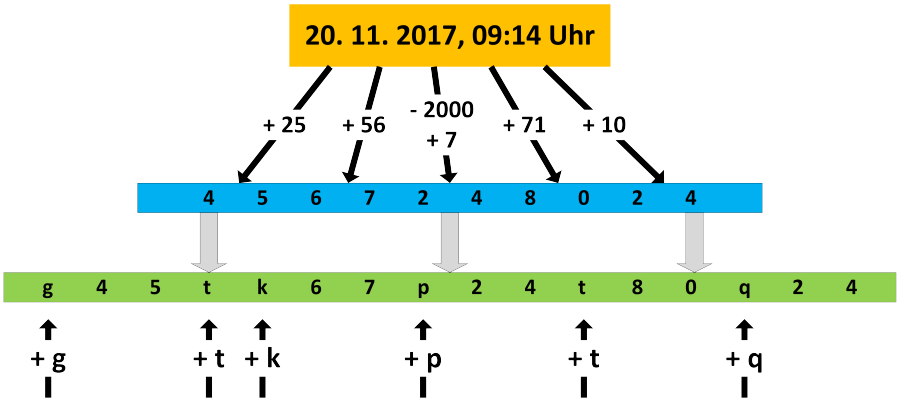

Abbildung 16: Generierungsprozess einer ProfileID

Anschließend werden nacheinander sechs zufällige Kleinbuchstaben erzeugt und an zufälliger Stelle in der Zahlenkette eingefügt (grüner Kasten). Durch Verwendung des Serverzeitpunktes in der ProfileID kann eine doppelte ID lediglich innerhalb einer Minute generiert werden. Zur besseren Lesbarkeit werden nur Kleinbuchstaben verwendet. Als Rückgabe liefert die Methode die fertig generierte ProfileID, ein direkter Zugriff auf die Datenbank erfolgt nicht. Im Beispiel ergibt sich als Rückgabewert *g45tk67p24t80q24*.

Als zusätzliche Funktion bietet der Dienst die Möglichkeit zu überprüfen, ob in der Datenbank ein Dokument mit einer bestimmten ID bereits vorhanden ist. Als dritte Methode offeriert der Service die Möglichkeit zur Validierung einer ProfileID, welche ebenfalls bei der Erzeugung eines neuen Profils auf Basis einer bestehenden ProfileID verwendet wird. Hierzu überprüft der Service, ob die Syntax eines übergebenen Parameters vergleichbar ist mit der einer gültigen ProfileID.

6.6.2 *Profile Service*

Der Profile Service kümmert sich im Microservice um die interne Verwaltung und Verarbeitung der Profile. Hierzu stehen insgesamt dreizehn Methoden zur Verfügung, die an dieser Stelle nicht alle detailliert beschrieben werden. Grundsätzlich werden zur Umsetzung aller über die REST-Schnittstellen[4] gestellten Anfragen Methoden bereitgestellt. Dazu zählen insbesondere das Anlegen, Speichern, Aktualisieren, Abrufen und Löschen von Profilen.

Mithilfe von Abbildung 17 soll die Arbeitsweise des Microservices beispielhaft anhand einer Pull-Anfrage dargestellt werden. Diese beginnt über eine HTTP-Anfrage in Form einer GET-Methode an die URI `/v1/profiles/{id}/{lastChange}`[5], über welche der Client dem Server seine *ProfileID* und den letzten Aktualisierungszeitpunkt seines lokalen Profils (bzw. einen selbstgewählten Zeitpunkt) mitteilt. Das Format des Zeitstempels im Parameter besitzt die auch für Menschen lesbare Struktur *yyyy-MM-dd'T'HH-mm-ss-SSS*.[6]

4 Ausnahme: Der DevController implementiert seine Funktionalitäten zum Teil selbst.

5 *lastChange* ist in der Dokumentation mit *clientProfileChange* benannt, wurde aber aufgrund des begrenzen Platzes in der Grafik umbenannt.

6 Java kann das Format mithilfe der Klasse *SimpleDateFormat* verarbeiten, siehe auch: `https://docs.oracle.com/javase/8/docs/api/java/text/SimpleDateFormat.html`

Zur Bearbeitung der Anfrage werden die Parameter aus der URI von der Klasse *ExistingProfileController* in einen String für die ID und ein Date-Objekt[7] für den Zeitpunkt konvertiert. Spring bietet hierzu bereits eine passende Konvertierung über Annotations an. Die aufgenommenen Daten werden anschließend über den Methodenaufruf *getProfileByIdComparingLastChange(id, lastChange)* an eine Instanz von *ProfileService* weitergereicht.

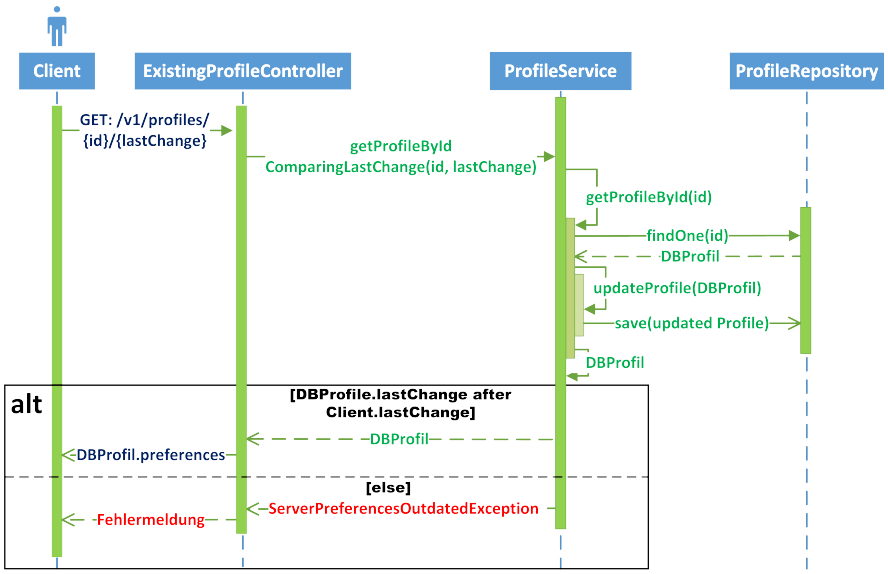

Abbildung 17: Sequenzdiagramm einer Pull-Anfrage

Im *ProfileService* wird über die Methode *getProfileById(id)* das passende Profil aus der Datenbank gesucht (DBProfil) und zurückgeliefert. Zugleich wird der letzte Kontaktzeitpunkt mit dem Profil über den Aufruf *updateProfile(DBProfil)* in der Datenbank aktualisiert. Kann kein Profil mit entsprechend der übergebenen ID gefunden werden, so wird eine *ProfileNotFoundException* geworfen und über den Error Handler (vgl. 6.8) an den Client eine entsprechende Fehlermeldung gesendet. Der Fall, dass ein Profil nicht gefunden werden kann ist in der Grafik zur besseren Übersicht nicht abgebildet.

Liegt ein Profil aus der Datenbank vor, so wird die Eigenschaft *lastProfileContact* mit dem in der URI übergebenen Zeitpunkt verglichen. Ist das Profil aus der Datenbank (DBProfil) aktueller als das des Clients, so wird zunächst das DBProfil an den *ExistingProfileController* zurückgeliefert und von dort die Preferences aus dem Profil an den Client gesendet. Ist das letzte Änderungsdatum des DBProfils älter als der vom Client in der URI spezifizierte Zeitpunkt, so wird eine *ServerPreferencesOutdatedException* geworfen und über den Error Handler eine Fehlermeldung an den Client gesendet (in der Grafik

7 Siehe auch `https://docs.oracle.com/javase/8/docs/api/java/util/Date.html`

vereinfacht dargestellt ohne Error Handler). Um Ungenauigkeiten beim Vergleich der Zeitpunkte zu vermeiden, gilt ein Profil genau dann als aktueller, wenn der letzte Änderungszeitpunkt um eine in den APPLICATION.PROPERTIES (vgl. 6.9.1) festgelegte Konstante nach dem Zeitstempel des anderen Profils liegt.

6.6.3 *Clearance Service*

Der dritte Service kümmert sich um das Aufräumen der Datenbank, d.h. um die endgültige Entfernung ungenutzter Profile. Hierzu wird zunächst aus den APPLICATION.PROPERTIES eine Zeitspanne ausgelesen, nach welcher ein Profil ohne Zugriff aus der Datenbank gelöscht werden kann (vgl. 6.9.1). Im Falle eines fehlerhaften Zugriffs auf die Datei sind als Standardwert 18 Monate festgelegt.

Über eine Instanz von *ProfileRepository* (vgl. 6.7) werden alle Profile aus der Datenbank ausgelesen, bei welchen der letzte Kontaktzeitpunkt die festgelegte Zeitspanne überschreitet. Da CouchDB keine dynamischen Abfragen über Views erlaubt (vgl. 5.4), werden zunächst alle Profile aus der Datenbank geladen und entsprechend im Microservice ausgewertet. Da die einzelnen Profile an sich nur eine geringe Größe von einigen Kilobyte[8] aufweisen, kann auch mit einer größeren Anzahl an Profilen aus der Datenbank ohne spürbare Einbußen der Performance gearbeitet werden. Nach der Auswertung der letzten Zugriffszeitpunkte sendet der Clearance Service den Befehl zum Löschen der entsprechenden Profile an das *ProfileRepository*. Zugleich wird auf der Konsole eine Nachricht mit Informationen bezüglich des Löschvorgangs ausgegeben.

Nach dem Löschvorgang wird aus dem Service heraus direkt über eine REST-Anfrage die Verdichtung (vgl. 5.4) der Datenbank angestoßen.[9] Dieser Befehl ist speziell auf die Verwendung von CouchDB ausgelegt, sollte das System nicht verfügbar oder durch ein anderes Datenbanksystem ersetzt worden sein, so wird eine entsprechende Fehlermeldung auf der Konsole ausgegeben, der Betrieb des Microservices jedoch nicht gestört, da der Fehler abgefangen wird.

Um die Belastung des Servers möglichst gering zu halten, wird der Service wöchentlich Montags um 03:00 Uhr morgens gestartet. Die Festlegung des Zeitplans erfolgt über die separate Klasse *ScheduledTasks*, wobei Spring über eine Annotation die Möglichkeit bietet, einen zeitgesteuerten Methodenaufruf auszuführen.

8 Ein Dokument, welches einen 5000 Zeichen langen String als Preferences gespeichert hat und 100 mal überschrieben worden ist, besitzt eine Größe von 143,09 KB.

9 Nach Durchführung des *compact*-Befehls besitzt das Dokument noch eine Größe von 8,47 KB.

6.7 REPOSITORY-EBENE

Die innerste Ebene im Microservice liefert die Anbindung an das Datenbanksystem. Spring stellt unter seinem Projekt Spring Data diverse vorgefertigte Interfaces bereit, die grundlegende Datenbankoperationen definieren. Im vorliegenden Programm wurde das Interface *CrudRepository*[10] verwendet, welches für generische Create, Read, Update, Delete (CRUD)-Operationen[11] gedacht ist. Die Operationen wurden im Interface *ProfileRepository* um einige zusätzliche Methoden erweitert, die Spezialfälle in der vorliegenden Anwendung darstellen.

Grundsätzlich bietet Spring Data bereits native Anbindungen an diverse Datenbanklösungen an, unter anderem MongoDB oder Redis [o.V17h, vgl.]. Für CouchDB sind keine nativen Module verfügbar, auf bestehende Community-Lösungen wurde aufgrund der fraglichen Zuverlässigkeit der Software sowie möglicher Lizenzprobleme verzichtet. Daher wurde die Implementierung des Interfaces manuell umgesetzt.

Zur Instanziierung des Interfaces kann auf Dependency Injection (DI) zurückgegriffen werden, wobei Spring eine passende Implementierung des *CrudRepository* entsprechend dem angebundenen Datenbanksystems bereitstellt. Durch die Verwendung von DI ist ein späterer Wechsel des Datenbanksystems leicht umzusetzen, da lediglich die implementierende Klasse zu ersetzen ist. Der folgende Quellcode zeigt die Verwendung von DI mittels der Annotation *Autowired*, wie sie in allen implementierenden Services verwendet wird. Spring liefert für das angegebene Interface automatisch eine passende Klasse, welche dieses implementiert.

```
@Autowired
ProfileRepository profileRepository;
```

Die Implementierung des Interfaces findet in der Klasse *ProfileRepositoryCouchDBImpl* statt. Zur Kommunikation mit einem CouchDB-System wird ausschließlich auf die Verwendung von REST gesetzt.[12] Insgesamt werden dreizehn Methoden implementiert für diverse CRUD-Operationen sowie zusätzlich CouchDB-spezifische Anforderungen wie der Erstellung von Datenbanken. Als Beispiel dient im folgenden die Implementierung zur Abfrage eines Profils aus der Datenbank über seine ID:

10 Siehe auch `https://docs.spring.io/spring-data/commons/docs/current/api/org/springframework/data/repository/CrudRepository.html`

11 Operationen zum Erstellen, Lesen, Schreiben und Löschen von Daten.

12 Eine Übersicht über die REST-API von CouchDB liefert die offizielle Dokumentation: `http://docs.couchdb.org/en/2.1.1/api/`

```
@Override
public Profile findOne(String id) {
        RestTemplate restTemplate = new RestTemplate();
        Profile profile;
        try {
                profile = restTemplate.getForObject(url + id,
                    Profile.class);
        } catch (HttpClientErrorException e) {
                profile = null;
        }
        return profile;
}
```

Spring MVC stellt mit der Klasse *RestTemplate* eine fertige Implementierung für REST-Anfragen bereit. Über diese wird durch Aufruf von `http://couchdb:5984/profiles/{id}` ein Dokument mit der in der URI festgelegten ProfileID gesucht und zurückgeliefert. Findet sich kein Profil, so wird die geworfene Exception abgefangen und stattdessen eine Nullreferenz zurückgeliefert. Die Reaktion darauf wird von den aufrufenden Klassen individuell vorgenommen. Die Adresse des Datenbanksystems, der verwendete Port sowie die Bezeichnung der Datenbank (in diesem Fall *profiles*) werden aus der Konfiguration aus den APPLICATION.PROPERTIES (vgl. 6.9.1) entnommen.

Zur internen Verarbeitung der Dokumente werden Objekte der Klasse *Profile* verwendet, bei denen es sich um Plain Old Java Objects (POJOs)[13] handelt und welche die Eigenschaften der Dokumente aus der Datenbank abbilden. Weiterhin bietet die Klasse diverse Getter- und Setter-Methoden sowie die Möglichkeit, alle Eigenschaften als *String* oder *HashMap* zurückzuliefern. Die Methode *getForObject(URL, Class<T>)* der Klasse *RestTemplate* stellt eine GET-Anfrage an die im Parameter spezifizierte URI und erhält von CouchDB als Antwort zunächst eine Repräsentation des Profils als JSON-Dokument. Dieses wird automatisch von Spring in ein POJO der im Parameter festgelegten Klasse konvertiert und kann im vorliegenden Fall direkt einer Variable vom Typ *Profile* zugewiesen werden [o.V17l, vgl.].

6.8 EXCEPTION HANDLING

Mit wachsender Größe und Komplexität eines Programms steigt auch die Anfälligkeit für Fehler, die den regulären Programmablauf stören können. In Java werden auftretende Fehler als Exceptions bezeichnet. Um ein unkontrolliertes Programmverhalten, im schlimmsten Fall einen Ausfall des Microservices zu verhindern, ist eine entsprechende Behandlung solcher Fehler, sogenanntes Exception Handling, notwendig.

13 Einfache Java-Objekte ohne besondere Funktionalitäten oder Abhängigkeiten.

Spring MVC bietet hierzu drei Ansätze: Die Verwendung von HTTP-Statuscodes, ein Controller-basiertes Exception Handling sowie ein globales Exception Handling.

Standardmäßig liefert jede unbehandelte Exception, die infolge einer REST-Anfrage auftritt, den HTTP-Statuscode 500 zurück, der auf einen internen Server Fehler hindeutet [Feo4, vgl.]. Über eigene Fehlerklassen, welche bestehende Exception-Klassen erweitern, kann jeweils separat ein Statuscode festgelegt werden, der bei Auftritt der Exception an den Client zurückgesendet wird. Über Festlegung entsprechender Statuscodes kann dem Client grob die Art des Fehlers mitgeteilt werden, beispielsweise fehlende Authentifizierung oder eine fehlerhaft aufgebaute Nachricht. Eine differenziertere Fehlerbeschreibung ist hingegen nicht möglich [Cha13, vgl.].

Der zweite Ansatz, das Controller-basierte Exception Handling, ermöglicht die Bearbeitung von Exceptions in denselben Klasse wie die Handler-Methoden zum Abfangen von REST-Anfragen liegen. So können innerhalb eines Controllers für verschiedene Arten von Exceptions unterschiedliche Handler-Methoden geschrieben werden, die entweder einen einfachen Statuscode zurückliefern, den Client auf eine spezielle Fehlerseite weiterleiten[14] oder eine vollständig individuelle Lösung anbieten [Cha13, vgl.]. Ein Problem stellt die steigende Komplexität der Klassen dar, die nun neben Handler-Methoden auch das Exception Handling beinhalten. Im vorliegenden Programm sind drei verschiedene Controller in der Front Controller Ebene im Einsatz, sodass für jede dieser drei Klassen ein separates Exception Handling notwendig wäre, was zu umfangreichem Boilerplate-Code führen würde und insbesondere bei Anpassungen einzelner Controller die Gefahr von Inkonsistenzen bezüglich des Exception Handlings steigt.

Aufgrund der genannten Nachteile der ersten beiden Ansätze wurde sich für ein globales Exception Handling an zentraler Stelle entschieden, die dem Ansatz der Aspektorientierte Programmierung (AOP) (vgl. 4.1) entspricht und die Separierung des Exception Handlings als Cross-Cutting Concern vornimmt, wie auch in der Struktur des Microservices in Abbildung 11 dargestellt . Hierzu wurden zunächst diverse eigene Exceptions erstellt, um ein qualitativ verbessertes Feedback zu erhalten. Alle selbst definierten Exceptions wurden als Unterklassen der *RuntimeException* entworfen, sodass ein Abfangen im Code selbst zwar möglich, aber nicht vorgeschrieben ist, da es sich um vom Compiler ungeprüfte Exceptions handelt [Ull12, vgl. S.635]. In den Klassen selbst werden neben der Festlegung des HTTP-Statuscode keine weiteren Anpassungen vorgenommen.

14 Einsatz nur bei Verwendung grafischer Oberflächen sinnvoll.

Die Kernfunktionalität des globalen Exception Handlings stellt die Klasse *ExceptionHandlingController* bereit. In dieser werden die eigens definierten ebenso wie eine Reihe vordefinierter Exceptions programmweit abgefangen und daraus eine standardisierte Fehlernachricht generiert und an den aufrufenden Client gesendet. Hierfür wird jeweils eine Instanz des Data Transfer Objects (DTOs)[15] *ErrorInformation* erzeugt und in dieser verschiedene Details zum aufgetretenen Fehler gesammelt. Anschließend werden mithilfe einer Instanz von *ResponseEntity* die gesammelten Informationen in Form eines JSON-Dokuments an den Client gesendet. Beispielhaft sei an dieser Stelle ein JSON-Dokument vorgestellt, welches ein Client bei Abfrage eines nicht vorhandenen Profils empfängt:

```
{
        "title": "Profil nicht gefunden",
        "exception": "de.privacy_avare.exeption.
            ProfileNotFoundException",
        "status": 404,
        "detail": "Kein Profil mit entsprechender ID gefunden.",
        "requestedURI": "/v1/profiles/4w66p62d480q5dl4/
                2017-11-22T10:16:44,413",
        "timestamp": "2017-11-23T09:48:51,877",
        "additionalInformation": ""
}
```

Wie zu sehen ist, beinhaltet das Dokument Informationen bezüglich der Art des Fehlers, die genaue Bezeichnung der Exception, den zugehörigen HTTP-Statuscode und Details zur Exception. Weiterhin werden die aufgerufene URI, die zum Fehler geführt hat, sowie der Zeitpunkt des Fehlerauftritts in für Menschen lesbarer Form festgehalten[16]. Das letzte Feld für zusätzliche Informationen wird im aktuellen Stand der Arbeit nicht verwendet, wurde jedoch im Hinblick auf mögliche, zukünftige Erweiterungen als zusätzliche Eigenschaft definiert.

6.9 KONFIGURATION

Spring Boot erleichtert über Autokonfiguration die Festlegung von Einstellungen für den Microservice. Trotz dieser Eigenschaft sind einige Konfigurationen weiterhin manuell durch den Entwickler festzulegen oder werden absichtlich nicht automatisiert. Hierzu zählen unter anderem die Festlegung der Adresse des Datenbanksystems, Erstellung von Zugangsdaten oder diverse Parameter für das Verhalten der Anwendung zur Laufzeit. Spring Boot bietet eine manuelle

15 Ein Objekt, welches Informationen kompakt zwischen zwei Prozessen transportiert, im vorliegenden Fall zwischen Server und Client [FR11, vgl. S.401f].

16 Format entspricht demselben wie es bei REST-Anfragen erwartet wird. Dies ermöglicht den Einsatz eines konsistenten Konverters für das Zeitformat auf Clientseite.

Konfiguration über eine separate Konfigurationsdatei, Umgebungsvariablen und Kommandozeilenparameter an [Gut16, vgl. S.62f]. Zusätzlich kann auf die Konfiguration über spezielle Konfigurationsklassen, wie sie im Spring Framework allgemein vorkommen, zurückgegriffen werden.

In den nachfolgenden Abschnitten werden die Konfigurationen für die Sicherung der REST-Schnittstellen sowie der Aufbau einer verschlüsselten Verbindung zwischen Client und Server über das Verschlüsselungsprotokoll Secure Sockets Layer (SSL) genauer beschrieben. Die Konfiguration von Swagger als Dokumentationstool für REST-Schnittstellen wird in Abschnitt 6.10 betrachtet. Im Microservice werden weitere Konfigurationen vorgenommen, deren Betrachtung in dieser Arbeit jedoch aufgrund fehlender Relevanz ausgelassen wird. Zunächst wird ein Blick auf die Externalisierung von Einstellungen über die Datei APPLICATION.PROPERTIES geworfen.

6.9.1 *Externalisierung von Programmeinstellungen*

Für den praktischen Einsatz des Microservice wurde der Wunsch geäußert, an zentraler Stelle diverse Parameter festlegen zu können, sodass auch nicht mit der Implementierung vertraute Personen nach kurzer Einarbeitungszeit in der Lage sind, grundlegende Konfigurationen vorzunehmen. Daher wurde in der Arbeit auf den Einsatz einer Properties-Datei gesetzt, welche in Spring standardmäßig den Namen APPLICATION.PROPERTIES besitzt. Eine aktuelle Version der Datei findet sich in Anhang B.

In der Datei selbst werden Programmparameter über ein Präfix, gefolgt von der Bezeichnung der Eigenschaft und Zuweisung eines Wertes festgelegt. So wird beispielsweise über die Zeile

```
server.port=8443
```

der Port, über welchen der Server von außen erreichbar ist, gesetzt. Spring Boot bietet eine Vielzahl solcher vordefinierter Eigenschaften, welche über die Datei manuell eingestellt werden können. Die offizielle Dokumentation gewährt einen Überblick über häufig genutzte, von Spring Boot bereits implementierte Eigenschaften, siehe hierzu [Web].

Zusätzlich zu den bereits vorhandenen Eigenschaften wurden für die Konfiguration der CouchDB-Anbindung die drei Parameter Adresse, Port und Datenbanknamen angelegt, über welche der REST-Zugriff auf CouchDB erfolgt. Änderungen bezüglich der Anbindung können ebenfalls in der APPLICATION.PROPERTIES-Datei vorgenommen werden, woraus die entsprechenden Klassen die Werte entnehmen.

Zusätzlich wurden Parameter definiert für die Festlegung des Zeitraums beim Vergleich der Aktualität von Profilen (vgl. 6.6.2) sowie einer Zeitspanne, nach welchem Profile ohne Nutzung gelöscht werden (vgl. 6.6.3).

6.9.2 *Schnittstellensicherung über Basic Authentication*

Wie bereits im Abschnitt 6.5.3 erläutert, soll der Zugriff auf die REST-Schnittstelle für die Entwicklung und Administration nur autorisierten Anwendern zur Verfügung stellen. Um dies zu gewährleisten, wird zunächst eine Authentifikation[17] des anfragenden Clients vorgenommen, worauf aufbauend eine mögliche Autorisierung[18] stattfindet. Hierzu stellt HTTP ein allgemeines Framework zur Zugriffskontrolle bereit, welches von Spring in Form von Basic Authentication über die Klasse *HttpSecurity* umgesetzt wurde [Ae17, vgl.].

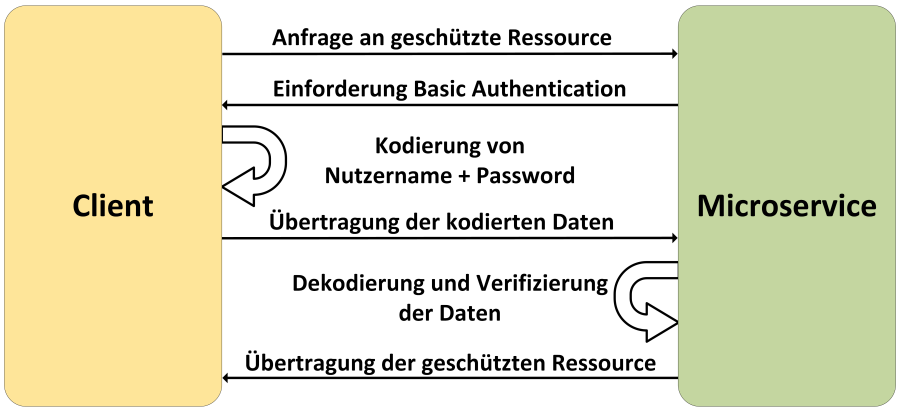

Abbildung 18: Ablauf einer HTTP Basic Authentication [VB15, vgl. S.123]

Der schematischen Ablauf von Basic Authentication ist in Abbildung 18 grafisch aufbereitet. Zunächst sendet der Server bei Clientanfragen bezüglich einer geschützten Ressource eine Aufforderung zur Authentifikation. Diese geschieht über Angabe eines Nutzernamens sowie eines Passworts. Diese werden an den Server gesendet, welcher diese verifiziert. Kann der Client identifiziert werden, so kann er entsprechend seinen Berechtigungen die Freigabe für die REST-Schnittstelle erhalten. Bei jeder weiteren Anfrage des Clients sendet dieser die Details der Authentifikation mit, wodurch das Prinzip der Zustandslosen Kommunikation von REST ein Stück weit verletzt wird.

17 Identifikation des Users anhand bereitgestellter Informationen [Ae17, vgl. S.2f].
18 Nach erfolgreicher Authentifikation des Clients wird geprüft, ob der Nutzer die Berechtigung für seine Anfrage besitzt [Ae17, vgl. S.3].

Anzumerken ist auch, dass bei dieser Art der Authentifizierung die Anmeldedaten des Clients nur codiert, aber nicht verschlüsselt übertragen werden, was eine Angriffsfläche für Hacker bietet [VB15, vgl. S.122f]. Die Verbindung zwischen Client und Server wird im vorliegenden Projekt zusätzlich über das Protokoll Transport Layer Security (TLS) verschlüsselt, sodass ein ungewolltes Mitlesen der Daten unterbunden wird.

Im Programm ist die Konfiguration für Basic Authentication in der Klasse *WebSecurityConfig* realisiert, welche von *WebSecurityConfigurerAdapter* aus dem Projekt Spring Data erbt. In dieser werden sämtliche Konfigurationen zur Festlegung der geschützten Bereiche sowie gültiger Nutzerdaten vorgenommen [o.V17d, vgl]. Bei Ausführung des Programms wird zunächst ein Servlet Filter erzeugt, welcher für die Sicherheit bezüglich aller eintreffenden Anfragen zuständig ist. Bei REST-Anfragen werden diese durch eine Reihe an Filtern geschickt, die u.a. prüfen, ob eine URI geschützt ist, der Client authentifiziert ist und falls ja, ob er die notwendige Berechtigung besitzt. Die in Abbildung 18 durchgeführten Operationen auf Seite des Microservices gehören somit zu den Aufgaben des Servlet Filters bzw. einer Schicht von diesem [Sca13, vgl. S.24ff].

6.9.3 *TLS/SSL-Verschlüsselung*

Für eine sichere Kommunikation zwischen Client und Server wird eine Verbindung über das Protokoll Hypertext Transfer Protocol Secure (HTTPS) verwendet. Dieses entspricht einer über SSL/TLS verschlüsselten Verbindung von HTTP. Das Protokoll Secure Sockets Layer (SSL) wurde zunächst zur Absicherung von HTTP-Verbindungen entworfen, ist mittlerweile aber auch für beliebige Anwendungen, die auf dem Transportprotokoll TCP basieren, verfügbar und wird häufig auch unter dem Namen des Nachfolgeprotokolls Transport Layer Security (TLS) geführt [Sch03, vgl. S.269f].

Der erste Bestandteil von SSL, das Handshake-Protokoll, kümmert sich um die Koordination und Konsistenz der Sitzungsinformationen von Client und Server. Beim ersten Kontakt sprechen beide Systeme ein Verfahren zur Kryptographie und Kompression der übertragenen Daten ab. Beim Einsatz von Zertifikaten sendet der Server dem Client sein Zertifikat zu, über welches er sich authentifiziert. Ebenfalls ist es möglich, dass der Server vom Client ein Zertifikat einfordert, sodass auch dessen Identität bestätigt werden kann. Ist die Authentifizierung beider Seiten erfolgreich abgeschlossen, können Datenpakete verschlüsselt übertragen werden [Eck05, vgl. S.382ff].

Hierfür ist das Record-Protokoll zuständig, welches auf Seite des Senders die zu übertragenden Daten fragmentiert, komprimiert, sie mit einer Prüfsumme ausstattet und anschließend verschlüsselt. Beim Empfänger werden die genannten Schritte in umgekehrter Reihenfolge vollzogen, die daraus resultierenden Daten können an weitere Schichten und Dienste zur Weiterverarbeitung übergeben werden [Ecko5, vgl. S.385f].

In der Implementierung von SSL im Microservice wird ein X.509-Zertifikat der KIT-Certification Authority[19] verwendet. Die Konfiguration von SSL erfolgt über die APPLICATION.PROPERTIES. In dieser werden der Keystore, in welchem das Zertifikat gespeichert ist, die Bezeichnung sowie das Passwort für den Zugriff festgelegt. Alternativ können über eine Konfigurationsklasse zusätzliche Einstellungen vorgenommen werden.

6.10 DOKUMENTATION DER REST-SCHNITTSTELLEN

Eine Dokumentation des Java-Quellcodes für die Implementierung des Microservice wird in Form einer JavaDoc-Datei bereitgestellt. Diese liefert eine detaillierte Beschreibung der einzelnen Komponenten und deren Funktionsweise. Für die praktische Anwendung des Microservice und dessen Schnittstellen bietet sich JavaDoc aufgrund seiner programmiertechnischen Sichtweise weniger an.

Daher wurde zur Dokumentation der REST-Schnittstellen (vgl. 6.5) eine weitere Dokumentation über Swagger umgesetzt. Das Framework wurde 2010 von Wordnik entwickelt und läuft aktuell unter dem Label von SmartBear Software. Neben der Möglichkeit zur Dokumentation bietet Swagger diverse Tools für die Generierung von Webservices basierend auf REST, die in der vorliegenden Arbeit jedoch nicht verwendet worden sind. Swagger ist dabei eine sogenannte Language-agnostic Spezifikation, d.h. zunächst einmal unabhängig von der verwendeten Programmiersprache und unterstützt somit auch Java [VB15, vgl. S.91]. Für die Verwendung von Swagger in Spring-Anwendungen steht mit Springfox eine Sammlung an Bibliotheken bereit, welche automatisch zur Laufzeit eine Swagger-basiert Dokumentation aus dem Quellcode erzeugt. Die Dokumentation an sich erfolgt über die Verwendung spezieller Annotations im Java-Code [KK17, vgl.].

Zur Visualisierung der Dokumentation wird zusätzlich Swagger UI eingesetzt, welches ebenfalls über Springfox bereitgestellt wird. Swagger UI ist ein Unterprojekt von Swagger und erzeugt aus den Dokumentationsdateien eine interaktive Weboberfläche. Diese liefert

19 Siehe auch: `https://www.ca.kit.edu/`

Details zu allen verfügbaren REST-Schnittstellen und integriert die Möglichkeit, die einzelnen Schnittstellen durch eigens definierte Anfragen zu testen. Insbesondere sind bei der Verwendung von Swagger UI keine Kenntnisse über die Implementierung der Funktionalität oder die Funktionsweise von Swagger notwendig [VB15, vgl. S.95f].

Ein Auszug aus der Dokumentation des Microservice ist in Abbildung 19 dargestellt. Für jeden der drei Controller wurde ein eigener Unterpunkt erstellt, unter welchem die jeweiligen REST-Schnittstellen aufgelistet sind. Zu jeder dieser Schnittstellen bietet die Dokumentation eine Beschreibung der Funktionalität, des Rückgabeformats sowie möglicher, auftretender Fehler. Zusätzlich kann aus Swagger UI heraus zu Testzwecken ein Aufruf an die jeweiligen Controller gesendet werden. Zur Laufzeit des Microservice ist die Dokumentation lokal erreichbar über die URI `localhost:8443/swagger-ui.html`. Um unbefugten Zugriff zu vermeiden, ist die Adresse zusätzlich über Basic Authentication gesichert (vgl. 6.9.2).

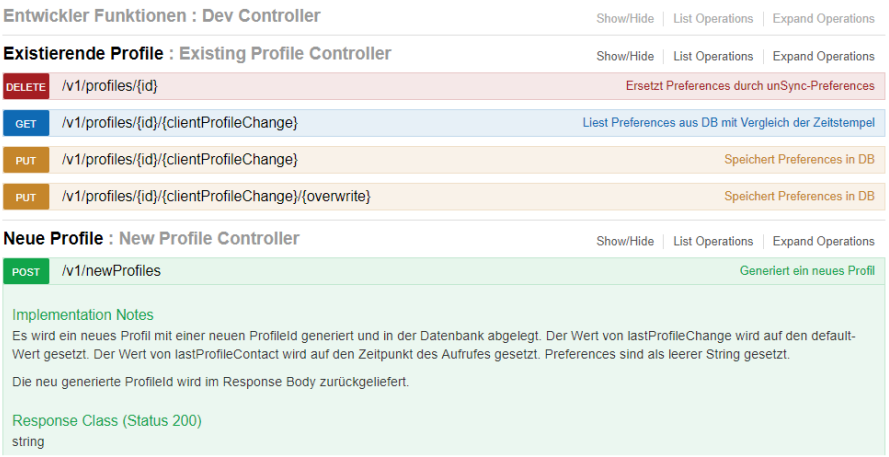

Abbildung 19: Oberfläche der Swagger UI-Dokumentation

7 SOFTWARETEST UND DEBUGGING

7.1 ÜBERBLICK SOFTWARETESTS

In diesem Kapitel werden die am Microservice durchgeführten Softwaretests erläutert. Der Standard *IEEE 829 Standard for Software and System Test Documentation* versteht unter einem Softwaretest folgendes:

„An activity in which a system or component is executed under specified conditions, the results are observed or recorded, and an evaluation is made of some aspect of the system or component.“[1] [o.V08, vgl. S.11]

Bei Softwaretests wird zwischen statischen Analysen und dynamischen Tests unterschieden. Bei einer statischen Analyse wird das zu prüfende Testobjekt nicht ausgeführt, sondern als solches selbst analysiert. Dabei wird dieses auf Anforderungs- und Entwurfsspezifikationen hin geprüft und der Quellcode ohne dessen Ausführung betrachtet. Bei einem dynamischen Test hingegen werden ausführbare Testobjekte verwendet, die dem Programmcode entspringen und unter kontrollierten Bedingungen zur Ausführung gebracht werden. Die Ergebnisse des ausgeführten Codes werden anschließend mit den eigentlich zu erwartenden Ergebnissen verglichen und bewertet [We13, vgl. S.33f.].

Im Bereich der dynamischen Softwaretests wird grundsätzlich zwischen zwei Prüfmethoden unterschieden: dem Blackbox-Verfahren und dem Whitebox-Verfahren. Beim Blackbox-Verfahren besitzt der Tester keine Kenntnis über den zugrundeliegenden Quellcode, daher werden die Testfälle aus den Spezifikationen erstellt und geprüft, ob diese eingehalten werden. Wie die Funktionalität an sich implementiert wurde, ist zunächst einmal nicht von Bedeutung, sondern vielmehr, ob sie ihre vorgesehenen Aufgaben erfüllen kann. Beim Whitebox-Verfahren hingegen steht dem Tester der Quellcode der Software zur Verfügung. Darauf aufbauend werden Testfälle entworfen, die alle möglichen Pfade der Software durchlaufen und auf Fehler hin untersuchen [Wit16, vgl. S.77f].

1 Deutsch etwa: Eine Aktivität, bei welcher ein System oder eine Komponente unter festgelegten Bedingungen ausgeführt wird, die Ergebnisse beobachtet oder aufgezeichnet werden und eine Bewertung anhand einiger Apsekte des Systems oder der Komponente vorgenommen wird.

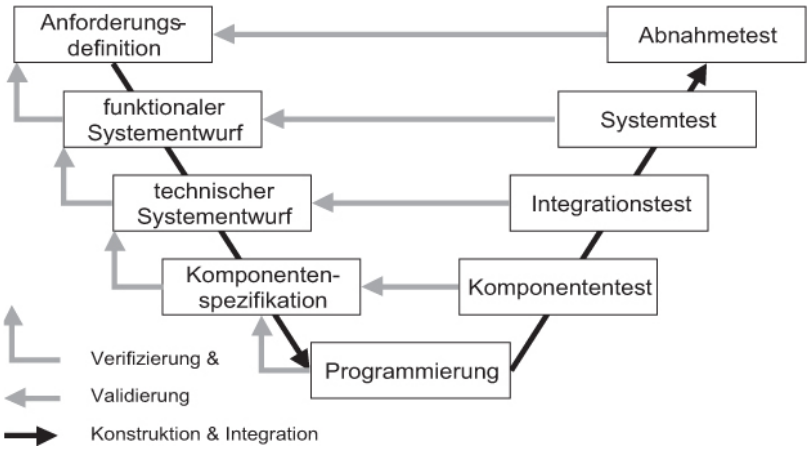

Abbildung 20: Übersicht V-Modell und Teststufen [Lin13, S.20]

Gemäß dem V-Modell[2] werden vier Teststufen, wie sie in Abbildung 20 dargestellt sind, unterschieden: Unit-Tests, Integrationstests, Systemtests und Abnahmetests. Bei Unit- bzw. Komponententests werden einzelne Elemente des Programms isoliert auf ihre Funktionalität hin getestet. Das Zusammenspiel mehrere Komponenten wird in einem Integrationstest betrachtet. Er findet auf mehreren Schichten statt und prüft, ob die Komponenten gemäß Spezifikation korrekt zusammenarbeiten. Der Systemtest prüft das System als Ganzes und teilweise auch die Interaktion mit anderen Systemen bzw. seiner Umgebung. Der letzte Test, der Abnahmetest, findet in der Zielumgebung statt und wird unter echten Einsatzbedingungen durchgeführt. Er soll dem Auftraggeber die korrekt spezifizierte Funktionalität der Software nachweisen [We13, vgl. S.4off].

7.2 REALISIERTE INTEGRATIONSTESTS

Für die Arbeit am Microservice wurden diverse Softwaretests durchgeführt, von denen die Integrationstests an dieser Stelle näher beleuchtet werden. Aufgrund der gerade zu Beginn der Arbeit wechselnden Bedingungen und Spezifikationen wurde auf die Implementierung reiner Unit-Tests verzichtet. Stattdessen wurden drei verschiedene Integrationstests aufgebaut, um die Funktionalität hinsichtlich der gestellten Anforderungen zu prüfen. Dabei wurde sich an den im Abschnitt Aufgabenstellung und Zielsetzung (vgl. 2.1) beschriebenen Use Cases (UCs) orientiert. Getestet wurden alle Funktionalitäten der Frontcontroller *NewProfileController* (vgl. 6.5.1) und *ExistingProfileController* (vgl. 6.5.2) über alle Ebenen bis zum Datenbanksystem, unter Verwendung von CouchDB.

2 Das V-Modell ist ein Vorgehensmodell in der Softwareentwicklung, bei dem der Softwareentwicklungsprozess in Phasen organisiert ist [Wit16, S.74].

Ein konkreter Test von *DevController* (vgl. 6.5.3) findet nicht statt, da dessen Funktionalität kein Bestandteil der konkreten Aufgabenstellung ist und in der Praxis eine untergeordnete Rolle spielen wird. Als dritter Integrationstest findet eine Überprüfung des *ClearanceService* (vgl. 6.6.3) statt. Da der Microservice in Form eines Schichtenmodells konzipiert wurde, besteht eine Abhängigkeit zwischen den Ebenen nur in eine Richtung. Aufgrund dessen wurde als Integrationsstrategie eine „Big Bang"-Strategie gefahren, d.h. eine direkte Zusammenschaltung aller Module [We13, vgl. S.158f].

Für die Realisierung der Integrationstests wurde JUnit 5 verwendet, ein Framework für das Testen von Java-Code. Dieses findet sich auch im Projekt Spring Testing wieder, welches zusätzlich verwendet wurde und diverse Annotations, Hilfsklassen und Mock-Objekte bereitstellt [VB15, vgl. S.156ff]. Die Realisierung der Integrationstests findet sich im Package *scr/test/java* des Microservice. Zu allen drei Tests existiert eine Beschreibung des jeweiligen Testkonzepts, siehe hierzu für den Test von UC 1 Anhang F, für die UCs 2, 3 und 4 Anhang G und für den UC 5 Anhang H. Die Struktur der Testkonzepte richtet sich grob nach IEEE 829 [Wit16, vgl. S.128f].

7.3 REALISIERTER SYSTEMTEST

Ein Systemtest des kompletten Projekts fand auf dem heimischen Rechner statt. Bei diesem Test wurde zunächst mithilfe von Postman[3], einem Programm zum Testen von REST-Schnittstellen, die Funktionalität des gesamten Systems geprüft. Das System blieb auch bei einer Reihe an paralleler Anfragen funktionsfähig und lieferte die erwarteten Ergebnisse. Die Anfragen wurden später mithilfe von einfachen Java-Anwendungen in vier Threads parallelisiert, sodass über einen längeren Zeitraum hinweg parallel Lese- und Schreibanfragen an den Microservice gesendet wurden. Die Anfragen konnten in der Regel beantwortet werden, teilweise kam es bei starker Belastung zu einem Fehler im Microservice, dieser stellte nach ca. 3 Sekunden fehlender Verfügbarkeit seine Dienste wieder zur Verfügung.

Aufgrund der unzuverlässigen Testumgebung auf dem heimischen Rechner mit diversen anderen Programmen im Hintergrund und geringerer Hardwareleistung, als auf dem späteren Hardware-Server zu erwarten ist, kann dieser Test nicht als vollkommen aussagekräftig angesehen werden, weshalb auf eine ausführliche Dokumentation verzichtet worden ist. Er zeigt jedoch, dass das System mit einer realistischen Belastung Anzahl an Anfragen keine Probleme auftreten sollten und im Fehlerfall kein Komplettabsturz des Systems zu erwarten ist.

3 Siehe auch: `https://www.getpostman.com/`

8 SOFTWARE DEPLOYMENT

8.1 SOFTWARE-CONTAINER

Für das Deployment des entworfenen Microservice sowie des Datenbanksystems CouchDB wird eine einfache Inbetriebnahme gefordert, die ohne umständliche Konfigurationen an der Software oder am Betriebssystem bzw. der Laufzeitumgebung auskommt. Für diesen Zweck wurde auf die Bereitstellung in Form von Software-Containern, kurz Container, zurückgegriffen. Die Idee solcher Containersysteme wird von Docker (siehe nachfolgenden Abschnitt) folgendermaßen beschrieben:

„A container image is a lightweight, stand-alone, executable package of a piece of software that includes everything needed to run it: code, runtime, system tools, system libraries, settings. Available for both Linux and Windows based apps, containerized software will always run the same, regardless of the environment.“[1] [o.V17p]

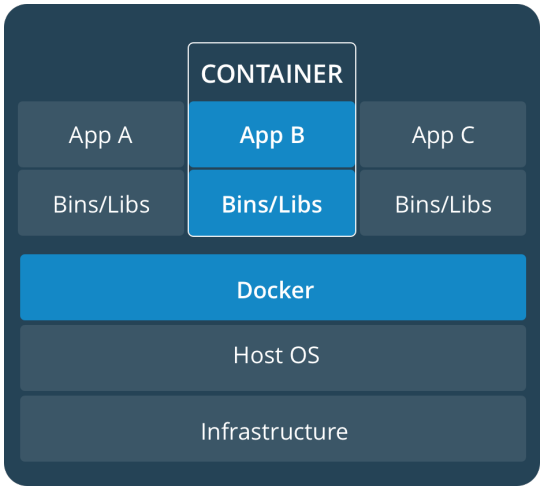

Abbildung 21: Struktur eines Containersystems [o.V17p]

1 Deutsch etwa: Ein Container-Image ist ein leichtgewichtiges, eigenständiges, ausführbares Softwarepaket, das alles beinhaltet, um die Software auszuführen: Quellcode, Laufzeitumgebung, System-Tools, System-Bibliotheken und Einstellungen. Aufgrund der Verfügbarkeit für Windows und Linux, wird in Container verpackte Software stets dasselbe Laufzeitverhalten besitzen, unabhängig von der Umgebung.

Abbildung 21 zeigt die Struktur eines Containersystems. Auf einem Betriebssystem wird hierzu eine Container-Engine aufgesetzt, in der Abbildung ist dies die Docker Engine, welche die Container in isolierten Prozessen ausführt. Ein Container selbst beinhaltet neben einer Anwendung die zugehörigen Bibliotheken und Einstellungen. Auf der Engine können mehrere Container parallel und unabhängig voneinander zum Einsatz gebracht werden [o.V17p, vgl.].

Die Verwendung von Containern bietet im Vergleich zu einem gewöhnlichen Deployment der Software bzw. zu einem Deployment innerhalb einer virtuellen Maschine verschiedene Vorteile. Zunächst einmal teilen sich Container Ressourcen mit dem Host-Betriebssystem, dies erhöht die Effizienz der Container verglichen mit einer Ausführung in virtuellen Maschinen, was eine starke Verkürzung der Startzeit nach sich zieht. Da jeder Container eine Art Laufzeitumgebung bereitstellt, bieten Container eine hohe Portierbarkeit zwischen verschiedenen Betriebssystemen. Aufgrund der leichtgewichtigen Strukturen der Container kann eine Vielzahl solcher Container gleichzeitig auf einem System ausgeführt werden, womit unter anderem verteilte Systeme ohne großen Aufwand emuliert werden können. Durch die Bereitstellung von verschiedener Software in fertigen Containern, wie beispielsweise Datenbanksysteme, können Endanwender und Entwickler diese ohne weitreichende Konfigurationen oder Installationen herunterladen und anwenden [Mou16, vgl. S.3f].

8.2 DOCKER

Für die Bereitstellung der Software im Containerformat wird Docker verwendet. Docker ist eine Open-Source-Software für die Entwicklung, Erstellung und Ausführung portabler, verteilter Anwendungen. Wie bereits im vorherigen Abschnitt beschrieben, packt Docker sämtliche benötigte Software und Abhängigkeiten in einen Container, das sogenannte Docker-Image, welches anschließend unabhängig vom Betriebssystem ausgeführt werden kann. Bei der Ausführung der Container laufen diese in einer isolierten Umgebung mit einem eigenen Dateisystem und eigenen Umgebungsvariablen. Somit sind Docker Container voneinander und vom zugrundeliegenden Betriebssystem zunächst isoliert. Mehrere Container können sich auch eine Umgebung teilen und somit untereinander in Kommunikation treten [Voh16, vgl. S.1].

Zur Erzeugung eigener Images, wie für den Microservice, wird eine einfache Textdatei, das Dockerfile, verwendet. In diesem wird unter anderem festgelegt, auf welchem Basisimage das neue Image entsteh, welche Schnittstellen nach außen verfügbar sind, welche Da-

teien in das Image hinzugefügt werden und welche Bestandteile des Programms bei der Ausführung des Images gestartet werden. Das Dockerfile für den Microservice findet sich in Anhang D.

Für das Datenbanksystem CouchDB stellt der Hersteller ein offizielles Docker-Image[2] über Docker-Hub, einem Cloud-Service zur Bereitstellung und Verteilung von Images, bereit.[3] Die Daten lassen sich über Docker Volumes persistent speichern. Im Gegensatz zur Anbindung von Verzeichnissen des Betriebssystems werden Volumes vollständig von Docker verwaltet und können leicht an Container angebunden oder zwischen Containern geteilt werden. Auch ist ein Backup der Daten mit geringem Aufwand möglich [o.V17o, vgl.].

8.3 DOCKER COMPOSE

Die Inbetriebnahme der Serverkomponente umfasst das Ausführen zweier Docker-Images: Eines für den Microservice und eines für CouchDB. Zusätzlich müssen beide Container in derselben Umgebung laufen, sodass eine lokale Kommunikation über REST möglich wird. Für CouchDB ist zusätzlich ein Volume für eine persistente Datenspeicherung zu erzeugen. Um diesen Prozess zu vereinfachen, kann mithilfe von Docker Compose eine Automatisierung umgesetzt werden. Docker Compose ist ein Tool, um Multi-Container Docker Anwendungen zu definieren und auszuführen. Hierzu wird über eine YAML-Datei, welche in Anhang E dargestellt ist, ein Anwendungsservice konfiguriert [o.V17n, vgl.].

In dieser Datei wird zunächst festgelegt, welche Docker-Images verwendet werden. Für jedes Image können u.a. separat die Schnittstellen, Verknüpfungen und Volumes konfiguriert werden. Nach Erstellung der Datei reicht ein Aufruf von *docker-compose up* in einer Kommandozeile zum Start der Anwendung. In der angehängten Datei wird hierbei auf Images aus dem Docker-Hub zurückgegriffen, die automatisch heruntergeladen werden, sofern sie noch nicht auf dem ausführenden Rechner vorhanden sind [o.V17n, vgl.].

2 Siehe: `https://hub.docker.com/_/couchdb/`

3 Zu Beginn der Arbeit mit Docker lag das offizielle CouchDB-Image eine Version hinter dem reinen Software-Release, mittlerweile ist auch die aktuellste Version als Docker-Image verfügbar. Es ist in Zukunft aber mit Verspätungen hinsichtlich der Auslieferung neuer Versionen über Docker zu rechnen.

9 REFLEXION

9.1 ALTERNATIVE LÖSUNGSANSÄTZE

Für die Umsetzung der gestellten Anforderungen wurde, wie im Laufe der Arbeit beschrieben, der Ansatz über eine Microservice-Architektur basierend auf Spring in Kombination mit einer Kommunikation über REST und CouchDB als Datenbank gewählt. Dies stellt nur einen von vielen möglichen Lösungsansätzen dar, die alternativ hätten gewählt werden können. In diesem Abschnitt werden einige von diesen vorgestellt.

Ein möglicher Ansatz stellt die Aufgliederung des Microservice in mehrere einzelne, voneinander unabhängige Microservices dar. Analog zum geschichteten Aufbau hätten die einzelnen Schichten in separate Programme aufgespalten werden können, die unabhängig voneinander gestartet werden und über Schnittstellen wie REST miteinander kommunizieren. Diese Aufteilung bietet den Vorteil, dass jeder Microservice für sich gesehen aufgrund der geringeren Größe im Vergleich zur realisierten Lösung übersichtlicher ist und leichter ausgetauscht bzw. erweitert werden kann. Schließlich wurde sich gegen die Aufspaltung des Microservice entschieden, da die Handhabung von drei oder mehr Programmen auch einen höheren Aufwand bezüglich Deployment und Verwaltung mit sich bringt, was insbesondere auch die Wartung in der Praxis erschwert. Weiterhin führt jede Kommunikation über Prozessgrenzen hinweg zu zusätzlichem Overhead sowie schlechterer Laufzeit im Vergleich zu einem internen Methodenaufruf, was insgesamt die Performance des Servers, wenn auch in geringem Umfang, beeinträchtigt. Beim vorliegenden Projektumfang ist eine gute Übersicht über den Microservice auch im aktuellen Zustand gegeben, sodass für Dritte eine Einarbeitung in die Struktur in angemessener Zeit möglich ist.

Anstelle einer Kommunikation über REST mithilfe von JSON wären auch XML-Dokumente als Transportmedium möglich gewesen. Außerdem hätte anstelle von REST auch SOAP (vgl. 3.4.2) eingesetzt werden können. Dies würde zunächst einmal Änderungen in der Schicht der Front Controller (vgl. 6.5) erfordern, bei welcher anstelle von Abrufen der Ressourcen gezielt Methoden aufgerufen worden wären (Remote Procedure Calls). Tiefere Schichten wären von der Umstellung nicht betroffen, sodass diese Lösungen auch ohne Probleme nachträglich realisiert werden könnten.

Im Kontext von AVARE, welches momentan in Form einer Android-Anwensung[1] als Prototyp implementiert wird, wären auch der Einsatz Java-spezifischer Kommunikationsprotokolle möglich gewesen, beispielsweise über die Verwendung von Sockets. Dies würde jedoch dem Prinzip einer geräteunabhängigen Kommunikation zuwiderlaufen und wäre nicht ohne Weiteres mit anderen Programmiersprachen und Betriebssystem kompatibel.

Im Bereich der Datenbanksysteme gibt es eine Vielzahl an Alternativen zu CouchDB. Im Bereich der dokumentenorientierten NoSQL-Systemen wurden anderweitig bereits gute Erfahrungen mit der Verwendung von MongoDB[2] und Couchbase in Konstellation mit einer Java-Anwendung gemacht. Beide Systeme stellen keine stabilen Versionen unter der Apache-Lizenz bereit, weshalb auf deren Einsatz verzichtet worden ist. Sie bieten jedoch den Vorteil einer nativen Unterstützung von Spring Data (vgl. 6.7), was gerade die Kommunikation mit dem Microservice vereinfacht und den Austausch des Datenbanksystems zusätzlich erleichtert hätte. Daneben wäre auch die Verwendung eines relationalen Datenbanksystems denkbar gewesen, da die Struktur der Profile in der Datenbank zum Ende des Projekts hin eine feste Form angenommen hat. Gerade zu Beginn der Entwicklung war der Aufbau der Profile noch von vielen Änderungen geprägt, auf welche mit NoSQL-Systemen besser reagiert werden konnte.

Grundsätzlich hätte auf die Verwendung eines Frameworks zur Implementierung des Microservice bzw. des Servers auch gänzlich verzichtet werden können. Dies hätte den Vorteil, in allen Bereichen des Projekts den vollständigen Überblick über die Implementierung der Funktionalitäten zu behalten. Jedoch wäre der Aufwand ein weitaus höherer gewesen und hätte kaum die Qualität eines mehrjährig entwickelten und umfangreich getesteten Frameworks erreichen können. Für Spring als Framework stehen diverse Alternativen wie Dropwizard oder Light-Rest-4J, mit denen vergleichbare Ergebnisse hätten erzielt werden können. Aufgrund der umfangreichen Dokumentation und dem Vorhandensein zahlreicher Literatur wurde Spring unter den Alternativen klar präferiert.

Am Ende stellt die Art und Weise der Implementierung eine schlichte Design-Entscheidung dar. Ein einziger, richtiger Lösungsansatz kann und möchte im Rahmen dieser Arbeit nicht gegeben werden.

1 Android-Apps basieren auf Java.

2 Verfügbar unter GNU AGPL v3.0-Lizenz, welche nicht in Apache-Lizenz 2.0 Projekten verwendet werden kann [0.V17e, vgl.].

9.2 GRENZEN UND PROBLEME

Auch wenn alle Anforderungen in Form des Microservice umgesetzt werden konnten, so gibt es doch einige Grenzen im praktischen Einsatz. Zunächst einmal kann jeder Benutzer über seine individuelle ProfileID lediglich einen einzigen Daten-String speichern. Dieser besitzt zwar aus Sicht des Datenbanksystems keine festgelegte Maximallänge, eine Speicherung unterschiedlicher Daten unter demselben Profil ist jedoch nicht möglich. Eine Verschlüsselung dieses Strings ist von Seiten der Clients umzusetzen, diese wird vom Server nicht geprüft, sodass unverschlüsselte Texte in der Datenbank lesbar abgelegt werden und grundsätzlich ein Datenschutzproblem aufweisen können. Komplexere Datenstrukturen müssen zunächst in einen String konvertiert und aus einem solchen auch wieder ausgelesen werden können. Multimedia-Inhalte sind grundsätzlich nicht mit dem Projekt kompatibel, entsprechen aber auch nicht dem Einsatzgebiet des Servers.

Aufgrund fehlender Serverinfrastruktur war ein Systemtest unter Realbedingungen nicht möglich. Die korrekte Funktionalität gemäß der Spezifikation konnte anhand der durchgeführten Integrationstests sowie durch einen Softwaretest auf dem heimischen Rechner festgestellt werden. Eine aussagekräftige Bewertung der Performance des Microservices bei Bearbeitung mehrerer, paralleler Anfragen ist nicht möglich. Für eine geringe Anzahl paralleler Anfragen sind keine merklichen Einbußen in der Performance zu erwarten, das Verhalten bei einer Vielzahl solcher Anfragen oder gar eines DDoS-Angriffs[3] lässt sich nur schwer vorhersagen.

Ein Problem der Sicherheit stellt die reine Authentifizierung über die ProfileID dar. Sobald eine Person oder Maschine Kenntnis über eine ID erlangt, hat diese Zugriff auf die gespeicherten Daten. Sind diese vom Client verschlüsselt, so sind die Inhalte zunächst vor unberechtigtem Lesezugriff geschützt. Ein Überschreiben und somit Löschen der Daten ist jedoch möglich, sodass es zu Datenverlust oder -Manipulation kommen kann. Es empfiehlt sich, den Server in Bereichen zu verwenden, in welchen mit verschlüsselten oder unsensiblen Informationen gearbeitet wird, deren Verlust verschmerzbar wäre.

3 Distributed-Denial-of-Service, gezielte Überlastung durch eine Vielzahl paralleler Anfragen.

9.3 FAZIT UND AUSBLICK

Im Zuge dieser Arbeit ist eine vollständig funktionsfähige Serveranwendung in Form eines Microservice entstanden, die alle gestellten Anforderungen erfüllt und zugleich dem Anspruch einer leichtgewichtigen, agilen Softwareentwicklung nachkommt. Von Dritten kann die fertige Software inklusive Datenbanksystem mithilfe von Docker mit einem einzeiligen Kommandozeilen-Befehl zur Ausführung gebracht werden, eine zusätzliche Installation oder Konfiguration von Software neben Docker ist nicht notwendig.

Im Kontext von AVARE gestartet, hat sich das Projekt zunächst stark an den daraus resultierenden Anforderungen orientiert, über die letzten Monate jedoch merklich in eine unabhängige Richtung entwickelt, sodass ein Einsatz in anderen Gebieten und Projekten durchaus vorstellbar ist. Grundsätzlich kann das Resultat in allen Bereichen eingesetzt werden, in welchen Daten in Form einzelner Strings ohne komplizierte Verfahren bezüglich der Authentisierung gespeichert werden können. Eine mögliche Zielgruppe stellt die Arbeit mit nicht sensiblen oder frei verfügbaren Informationen dar, für welche sich der Dienst bestens eignet. Solange eine Verschlüsselung der Daten auf Clientseite stattfindet, können auch sensiblere Daten abgelegt werden. Die Software kann auch als stark vereinfachter Cloud-Speicher verstanden werden, der im Gegensatz zu Anbietern wie Google oder Facebook keine automatische Analyse der Nutzerdaten vornimmt.[4]

Die Entwicklung des Microservice an sich ist im Rahmen der Arbeit abgeschlossen worden, Raum für Weiterentwicklung und Verbesserungen besteht trotzdem. Beispielsweise eine Erweiterung des Umfangs der zu speichernden Datenstrukturen. Möglich wäre der Einsatz von Feldern zur Speicherung mehrerer Strings unter einer gemeinsamen ProfileID. Diese könnten entweder alle gemeinsam an den Client zurückgeliefert werden, welcher sich dann die gesuchten Daten selbst auswählt oder eine Indexierung der Daten, wodurch der Anwender genau festlegen kann, welchen Datensatz er aus seinem Profil lesen bzw. schreiben möchte.

Für die Verwaltung der Datensätze könnte zusätzlich eine grafische Oberfläche für Endanwender bereitgestellt werden, über welche diese Zugriff auf ihre Daten erhalten. Besonders zusammen mit einer möglichen Erweiterung der Datenstrukturen könnte der Dienst sich zu einem vollständigen Webservice weiterentwickeln.

4 Eine manuelle Auswertung der Datenbank durch Menschen kann grundsätzlich nicht ausgeschlossen werden!

A

HTTP-METHODEN

Tabelle 1: Übersicht über HTTP-Verben [Til11, vgl. S.51ff]

Verb	**Beschreibung**	**Safe**	**Idempotent**
GET	Abrufen von Informationen einer Ressource, d.h. lesender Zugriff. Eine Veränderung der Daten ist nicht vorgesehen. Seiteneffekte wie Aktualisierung einer Logdatei gelten als akzeptierbar. GET ist die Operation, für welche das gesamte Web optimiert ist.	Ja	Ja
HEAD	Grundsätzlich ähnlich wie GET, liefert jedoch nur Metadaten, den HTTP-Header, nicht aber die Ressource selbst zurück. Kann verwendet werden um zu prüfen, ob eine Ressource vorhanden ist, ohne diese selbst zu übertragen.	Ja	Ja
PUT	Aktualisiert eine bestehende Ressource oder erzeugt diese, falls noch nicht vorhanden. Die Ressource wird über den Entity Body einer HTTP-Nachricht transportiert.	Nein	Ja
POST	Dient zur Generierung einer neuen Ressource unter einer festgelegten URI. Die Methode wird häufig auch dann eingesetzt, wenn es keine andere, passende Methode gibt. Im Gegensatz zu PUT wird eine URI zum Anlegen der Ressource angegeben, nicht eine URI, unter welcher die Ressource zu finden ist.	Nein	Nein
DELETE	Dient zum Löschen einer Ressource. Hierunter fällt auch das Setzen eines internen Attributs, welches den Zustand *gelöscht* repräsentiert.	Nein	Ja
OPTIONS	Liefert Metadaten über eine Ressource, welche die von der Ressource unterstützten Methoden enthalten.	Ja	Ja

B

APPLICATION.PROPERTIES

Zugangsdaten für Admin

```
admin.username = ******
admin.password = ******
```

Aktivierung Serverschnittstellen für zusätzliche Informationen sowie Einstellungen bezüglich der Verschlüsselung und Sicherheit

```
endpoints.enabled=true
management.health.defaults.enabled=true
management.health.diskspace.enabled=true
management.security.enabled=false
management.port=8443
management.ssl.enabled=true
management.ssl.key-store=jks
management.ssl.key-password=password
management.ssl.keyAlias=tomcat
```

Festlegung der TLS/SSL-Verschlüsselung für die REST-Schnittstellen

```
server.port=8443
server.ssl.keyStore=keystore.jks
server.ssl.enabled=false
server.ssl.keyPassword=password
server.ssl.keyStoreType=jks
server.ssl.keyAlias=tomcat
```

Informationen zum Verbindungsaufbau mit einem CouchDB-Container (Docker)

```
couchdb.adress=http://couchdb
couchdb.port=5984
couchdb.databaseName=profiles
```

Festlegung des kleinstmöglichen Zeitabstandes zwischen Serverprofil und Clientprofil (in Minuten)

```
server.minTimeDifference=5
```

Festlegung des Zeitraums, nach welchem ungenutzte Profile gelöscht werden (in Monaten)

```
server.monthsBeforeDeletion=18
```

C

CUSTOM EXCEPTIONS

- *ClientPreferencesOutdatedException*: Signalisiert den veralteten Stand eines Profils auf einem Clientgerät. Die Exception wird bei einer Push-Anfrage geworfen, wenn der vom Client angegebene Zeitstempel älter ist als die Eigenschaft *lastProfileChange* des entsprechenden Profils in der Datenbank.

- *ServerPreferencesOutdatedException*: Signalisiert den veralteten Stand eines Profils aus der Datenbank. Die Exception wird bei einer Pull-Anfrage geworfen, wenn der vom Client angegebene Zeitstempel neuer ist als die Eigenschaft *lastProfileChange* des entsprechenden Profils in der Datenbank.

- *MalformedProfileIdException*: Signalisiert die Angabe einer ID mit ungültigem Format im Parameter bei der Erzeugung eines neuen Profils.

- *NoProfilesInDatabaseException*: Signalisiert eine leere Datenbank beim Versuch eine Liste aller Profile aus dieser abzurufen.

- *ProfileAlreadyExistsException*: Signalisiert, dass bei der Erzeugung eines Profils eine ID angegeben wurde, die bereits einem anderen Profil in der Datenbank zugeordnet ist.

- *ProfileNotFoundException*: Signalisiert, dass kein Dokument mit einer spezifizierten ProfileID in der Datenbank vorhanden ist.

D

DOCKERFILE

```
FROM java:8

LABEL description="Microservice fuer die Datensynchronisation"

LABEL maintainer="Lukas Struppek"

LABEL contact="lukas.struppek@student.kit.edu"

EXPOSE 8443

ADD /target/AvareSyncServer-1.0.jar /AvareSyncServer-1.0.jar

ENTRYPOINT ["java","-jar","AvareSyncServer-1.0.jar"]
```

E

DOCKER-COMPOSE.YAML

```
version: '3'

services:
        couchdb:
                image: couchdb:latest
                volumes:
                        - couchdb_data:/opt/couchdb/data
                ports:
                        - "5984:5984"
                restart: always

        syncserver:
                image: lukasstruppek/syncserver:latest
                depends_on:
                        - couchdb
                ports:
                        - "8443:8443"
                restart: always

        volumes:
                couchdb_data:
```

F

TESTKONZEPT INTEGRATIONSTEST USECASE 1

Testkonzeptbeschreibung: Der Integrationstest umfasst das Testen aller Anforderungen, die sich aus Use Case 1, dem Anlegen von neuen Profilen, ergeben (vgl. 2.1).

Testobjekte: *NewProfileController*, *ProfileService*, *IdService*, *Profile*, *ProfileRepositoryCouchDBImpl*

Zu testende Leistungsmerkmale:

- Generierung einer neuen ID und Anlage eines Profils mit dieser ID inkl. Überprüfung der zugewiesenen default-Werte.
- Generierung eines neuen Profils basierend auf einer bereits existierenden, noch nicht vergebenen ProfileID. Geprüft wird eine korrekte Erzeugung bei Verwendung von Kleinbuchstaben sowie von Großbuchstaben.
- Generierung eines neuen Profils basierend auf einer bereits vergebenen ProfileID.
- Generierung eines neuen Profils basierend auf einer zu langen bzw. zu kurzen ProfileID.
- Generierung eines neuen Profils basierend auf einer ProfileID mit zu vielen Zahlen bzw. zu vielen Buchstaben, aber festgelegter Länge von 16 Zeichen.

Nicht getestete Leistungsmerkmale: Verwendung von Sonderzeichen in der ProfileID, Verhalten bei zufälliger Generierung zweier identischer IDs.

Teststrategie: Nutzung von REST-Schnittstellen für die Generierung neuer Profile bzw. deren IDs. Bei korrekten Erzeugungen wird das Dokument aus der Datenbank ausgelesen und dessen Inhalte geprüft. Bei fehlerhaften Erzeugungen werden die Art der Exception sowie der gesendete Statuscode geprüft. Am Ende des Tests wird die Datenbank geleert.

Abnahmekriterien: Inhalt der Profile in der Datenbank, HTTP-Statuscodes und auftretende Exceptions werden mit den in der Dokumentation spezifizierten Werten verglichen.

G

TESTKONZEPT INTEGRATIONSTEST USECASES 2, 3, 4

Testkonzeptbeschreibung: Der Integrationstest umfasst das Testen aller Anforderungen, die sich aus den Use Cases 2, 3 und 4 ergeben. Diese umfassen das Pushen, Pullen und Löschen von Profilen (vgl. 2.1).

Testobjekte: *ExistingProfileController*, *ProfileService*, *IdService*, *Profile*, *ProfileRepositoryCouchDBImpl*

Zu testende Leistungsmerkmale:

- Löschen von Profilen, d.h. Ersetzen der Preferences durch unsync-Nachrichten. Geprüft wird das erstmalige löschen, das erneute Löschen bereits auf unsync gesetzter Profile sowie das Löschen nicht vorhandener Profile in Form einer in der Datenbank nicht vorhandenen ProfileID, einer ungültigen ProfileID sowie einer nicht übergebenen ProfileID.
- Erstmaliges Pushen von Profilen, das Pushen von Profilen mit aktuellen Daten, veralteten Daten und leeren Preferences. Weiterhin wird das Verhalten für ungültige Parameter bezüglich des Zeitstempels und der ProfileID betrachtet.
- Beim Pushen werden alle drei möglichen Schnittstellen getestet, d.h. mit Angabe von Overwrite (true, false) bzw. ohne Angabe des Parameters.
- Pullen von Profilen mit ausreichendem Unterschied der Zeitstempel, zu geringem Unterschied des Zeitstempels sowie mit ungültigem Format bezüglich Zeitstempel und ProfileID.
- Beachtung der in den APPLICATION.PROPERTIES festgelegten minimalen Zeitabständen beim Vergleich der Zeitstempel.

Nicht getestete Leistungsmerkmale: Sonderzeichen bei der Angabe von Parametern. Verhalten beim Umgang mit Strings außerordentlicher Größe.

Teststrategie: Zunächst werden in der Datenbank für die einzelnen Anwendungsfälle vorgefertigte Profile mit entsprechend zu testenden Eigenschaften erzeugt und anschließend jeweils durch Verwendung der REST-Schnittstelle die einzelnen Methoden getestet. Am Ende des Tests wird die Datenbank von übriggebliebenen Profilen bereinigt. Zusätzlich wird aus der Datei APPLICATION.PROPERTIES der Wert für *minTimeDifference* ausgelesen.

Abnahmekriterien: Inhalt der Profile in der Datenbank, HTTP-Statuscodes und auftretende Exceptions werden mit den in der Dokumentation spezifizierten Werten verglichen.

H

TESTKONZEPT INTEGRATIONSTEST USECASE 5

Testkonzeptbeschreibung: Der Integrationstest umfasst das Testen aller Anforderungen, die sich aus Use Case 5, dem Aufräumen der Datenbank, ergeben (vgl. 2.1).

Testobjekte: *ClearanceService*, *ProfileRepository*

Zu testende Leistungsmerkmale:

- Korrektes erkennen und Löschen aller Profile mit einem Wert für *lastProfileContact*, der älter ist als die in APPLICATION.PROPERTIES festgelegte Zeitspanne *monthtsBeforeDeletion*.

Nicht getestete Leistungsmerkmale: Zeitgesteuerter Start des Prozesses zum Aufräumen, der Comptact-Befehl von CouchDB für die Verdichtung der Datenbank.

Teststrategie: Zunächst werden in der Datenbank 30 veraltete und 60 aktuelle Profile generiert. Anschließend wird der Aufräumprozess manuell gestartet und anschließend geprüft, ob die veralteten Profile korrekt gelöscht wurden und die aktuellen Profile in der Datenbank erhalten geblieben sind.

Abnahmekriterien: Die Datenbank darf nur noch die volle Anzahl aktueller Profile enthalten.

I

CD-INHALTE

Auf der beigelegten CD finden sich folgende Inhalte:

- Bachelorarbeit im pdf-Format
- Quellcode des Microservice als zip-File (Eclipse Projekt)
- Microservice als runnable-JAR-File
- Dockerfile
- docker-compose.yaml
- Apache-Lizenz 2.0 License-File

[Ae17] Alex, Ben ; et al. ; Pivotal Software, Inc. (Hrsg.): *Spring Security Reference: HttpSecurity*. https://docs.spring.io/spring-security/site/docs/5.0.0.BUILD-SNAPSHOT/reference/htmlsingle/#jc-httpsecurity. Version: 2017. – Stand: 22.11.2017

[BCK10] Bass, Len ; Clements, Paul ; Kazman, Rick: *Software architecture in practice*. 2. Auflage. Boston : Addison-Wesley, 2010

[Bue16] Buehrle, Anita ; Weave Cloud (Hrsg.): *What are Microservices?* https://www.weave.works/blog/what-are-microservices/. Version: 31.10.2016. – Stand: 01.11.2017

[Cat11] Cattell, Rick: Scalable SQL and NoSQL data stores. In: *ACM SIGMOD Record* 39 (2011), Nr. 4. https://dl.acm.org/citation.cfm?doid=1978915.1978919. – ISSN 01635808. – Stand: 10.12.2017

[Cha13] Chapman, Paul ; Pivotal Software, Inc. (Hrsg.): *Exception Handling in Spring MVC*. https://spring.io/blog/2013/11/01/exception-handling-in-spring-mvc. Version: 2013. – Stand: 21.11.2017

[Dar14] Darshini, Priya: *Spring MVC 4.0 RESTFul Web Services Simple Example*. http://www.programming-free.com/2014/01/spring-mvc-40-restful-web-services.html. Version: 2014. – Stand: 07.11.2017

[Eck05] Eckert, Claudia: *IT-Sicherheit: Konzepte - Verfahren - Protokolle*. Studienausgabe. München : Oldenbourg, 2005

[Edl11] Edlich, Stefan: *NoSQL: Einstieg in die Welt nichtrelationaler Web 2.0 Datenbanken*. 2. Auflage. München : Hanser, 2011

[ENS03] Elmasri, Ramez ; Navathe, Sham ; Shafir, Angelika: *Grundlagen von Datenbanksystemen*. 3. Auflage. München : Pearson-Studium, 2003

[Ert07] Ertel, Wolfgang: *Angewandte Kryptographie*. 3. Auflage. München : Hanser, 2007

[Eva17] Evans, Dave ; Cisco Internet Business Solutions Group (Hrsg.): *Das Internet der Dinge: So verändert die nächste Dimension des Internet die Welt*. https://www.cisco.com/c/

dam/global/de_de/assets/executives/pdf/Internet_of_Things_IoT_IBSG_0411FINAL.pdf. Version: 2017. – Stand: 24.11.2017

[Fe04] FIELDING, Roy ; ET AL. ; W3C (Hrsg.): *HTTP/1.1: Status Code Definitions*. https://www.w3.org/Protocols/rfc2616/rfc2616-sec10.html. Version: 2004. – Stand: 21.11.2017

[FL15] FOWLER, Martin ; LEWIS, James: *Microservices: Nur ein weiteres Konzept in der Softwarearchitektur oder mehr?* https://www.sigs-datacom.de/uploads/tx_dmjournals/fowler_lewis_OTS_Architekturen_15.pdf. Version: 01/2015, 2015. – Stand: 12.12.2017

[Fow15] FOWLER, Martin: *Monolith First*. https://martinfowler.com/bliki/MonolithFirst.html. Version: 2015. – Stand: 03.11.2017

[FR11] FOWLER, Martin ; RICE, David: *Patterns of enterprise application architecture*. 17. Auflage. Boston : Addison-Wesley, 2011

[FZI16] FZI FORSCHUNGSZENTRUM INFORMATIK (Hrsg.): *Projekt AVARE: Datenschutz auf allen Endgeräten*. https://www.fzi.de/de/aktuelles/news/detail/artikel/projekt-avare-datenschutz-auf-allen-endgeraeten/. Version: 2016. – Stand: 10.12.2017

[Gor11] GORTON, Ian: *Essential Software Architecture*. 2. Auflage. Heidelberg : Springer-Verlag, 2011

[Gut16] GUTIERREZ, Felipe: *Pro Spring Boot*. 1. Auflage. New York : Apress, 2016

[He11] HAN, Jing ; ET AL.: Survey on NoSQL database. In: HU, Bo (Hrsg.): *6th International Conference on Pervasive Computing and Applications (ICPCA), 2011*. Piscataway, NJ : IEEE, 2011

[HH10] HEUSER, Oliver ; HOLUBEK, Andreas: *Java-Web-Services in der Praxis*. 1. Auflage. Heidelberg : dpunkt-Verl., 2010

[HRW13] HÖLZL, Matthias ; RAED, Allaithy ; WIRSING, Martin: *Java kompakt: Eine Einführung in die Software-Entwicklung mit Java*. 1. Auflage. Heidelberg : Springer, 2013

[Ie11] IHNS, Oliver ; ET AL.: *EJB 3.1 professionell: Grundlagen- und Expertenwissen zu Enterprise JavaBeans 3.1 - inkl. JPA 2.0*. 2. Auflage. Heidelberg : dpunkt-Verl., 2011

[Je14] JOHNSON, Rod ; ET AL.: *Spring Framework Reference: Introduction to Spring Framework 4.0*. https://docs.spring.io/

spring/docs/4.0.x/spring-framework-reference/html/overview.html#overview-modules. Version: 2014. – Stand: 06.11.2017

[KK17] Krishnan, Dilip ; Kelly, Adrian ; springfox (Hrsg.): *Springfox Reference Documentation*. http://springfox.github.io/springfox/docs/current/#introduction. Version: 2017. – Stand: 24.11.2017

[Kle16] Kleuker, Stephan: *Grundkurs Datenbankentwicklung: Von der Anforderungsanalyse zur komplexen Datenbankanfrage*. 4. Auflage. Wiesbaden : Springer Vieweg, 2016

[Le13] Lipinski, Klaus ; et al. ; ITWissen.info (Hrsg.): *Monolithische Software-Architektur*. http://www.itwissen.info/Monolithische-Software-Architektur.html. Version: 02.11.2013. – Stand: 01.11.2017

[Lea10] Leavitt, Neal: Will NoSQL Databases Live Up to Their Promise? In: *Computer* 43 (2010), Nr. 2. http://ieeexplore.ieee.org/document/5410700/. – ISSN 0018–9162

[Lin13] Linz, Tilo: *Testen in Scrum-Projekten: Leitfaden für Softwarequalität in der agilen Welt*. 1. Aufl. Heidelberg : dpunkt.verlag, 2013

[Mou16] Mouat, Adrian: *Docker: Software entwickeln und deployen mit Containern*. 1. Auflage. Heidelberg : dpunkt.verlag, 2016

[MS04] Mutschler, Bela ; Specht, Günther: *Mobile Datenbanksysteme: Architektur, Implementierung, Konzepte*. Heidelberg : Springer, 2004

[Mül17] Müller, Melanie ; double Slash Blod (Hrsg.): *Convention over Configuration in Spring Boot*. https://blog.doubleslash.de/convention-over-configuration-in-spring-boot/. Version: 2017. – Stand: 06.11.2017

[New15] Newman, Samuel: *Building microservices*. 1. Auflage. Sebastopol, CA : O'Reilly, 2015

[New16] Newson, Robert: *CouchDB 2.0 Architecture*. https://blog.couchdb.org/2016/08/01/couchdb-2-0-architecture/. Version: 2016. – Stand: 11.11.2017

[NSS14] Namiot, Dmitry ; Sneps-Sneppe, Manfred ; International Journal of Open Information Technologies (Hrsg.): *On Micro-services Architecture*. http://injoit.org/index.php/j1/article/view/139/104. Version: 2, 2014. – Stand: 12.12.2017

[o.V08] o.V.: *IEEE standard for software and system test documentation*. New York : Institute for Electrical and Electronics Engineers, 2008

[o.V13] o.V. ; Oracle (Hrsg.): *Overview of the JMS API - The Java EE 6 Tutorial*. https://docs.oracle.com/javaee/6/tutorial/doc/bncdr.html. Version: 2013. – Stand: 04.11.2017

[o.V16] o.V. ; IBM (Hrsg.): *CouchDB Logo*. https://developer.ibm.com/dwblog/wp-content/uploads/sites/73/CouchDB-horizontal-logo-1.png. Version: 21.09.2016. – Stand: 11.11.2017

[o.V17a] o.V. ; Pivotal Software, Inc. (Hrsg.): *Spring Boot Logo*. https://spring.io/img/homepage/icon-spring-boot.svg. Version: 03.12.2017. – Stand: 11.12.2017

[o.V17b] o.V. ; Apache Software Foundation (Hrsg.): *Apache CouchDB 2.1 Documentation: Technical Overview*. http://docs.couchdb.org/en/2.1.1/intro/overview.html. Version: 07.11.2017. – Stand: 11.11.2017

[o.V17c] o.V. ; Apache Software Foundation (Hrsg.): *Apache CouchDB 2.1 Documentation: Why CouchDB? -*. http://docs.couchdb.org/en/2.1.1/intro/why.html. Version: 07.11.2017. – Stand: 11.11.2017

[o.V17d] o.V. ; Pivotal Software, Inc. (Hrsg.): *Class WebSecurityConfigurerAdapter (Spring Security 4.2.3.RELEASE API)*. https://docs.spring.io/autorepo/docs/spring-security/4.2.1.RELEASE/apidocs/org/springframework/security/config/annotation/web/configuration/WebSecurityConfigurerAdapter.html?is-external=true. Version: 08.06.2017. – Stand: 11.12.2017

[o.V17e] o.V. ; The Apache Software Foundation (Hrsg.): *Apache License v2.0 and GPL Compatibility*. https://www.apache.org/licenses/GPL-compatibility.html. Version: 19.09.2017. – Stand: 10.12.2017

[o.V17f] o.V. ; Apache Foundation (Hrsg.): *Apache License, Version 2.0*. https://www.apache.org/licenses/LICENSE-2.0. Version: 2017. – Stand: 11.12.2017

[o.V17g] o.V. ; Couchbase (Hrsg.): *Couchbase Server Editions*. https://www.couchbase.com/editions. Version: 2017. – Stand: 11.12.2017

[o.V17h] o.V. ; Pivotal Software, Inc. (Hrsg.): *Spring Data*. http://projects.spring.io/spring-data/. Version: 2017. – Stand: 11.12.2017

[o.V17i] o.V. ; Pivotal Software, Inc. (Hrsg.): *Spring Data Reference*. https://docs.spring.io/spring/docs/current/spring-framework-reference/web.html. Version: 2017. – Stand: 12.12.2017

[o.V17j] o.V. ; Google LLC (Hrsg.): *YouTube - Android-Apps auf Google Play*. https://play.google.com/store/apps/details?id=com.google.android.youtube&hl=de. Version: 2017. – Stand: 10.12.2017

[o.V17k] o.V. ; Pivotal Software, Inc. (Hrsg.): *Annotation Type RequestMapping (Spring Framework 5.0.1.RELEASE API)*. https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RequestMapping.html. Version: 24.10.2017. – Stand: 14.11.2017

[o.V17l] o.V. ; Pivotal Software, Inc. (Hrsg.): *Class RestTemplate (Spring Framework 5.0.1.RELEASE API)*. https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/client/RestTemplate.html. Version: 24.10.2017. – Stand: 20.11.2017

[o.V17m] o.V. ; Pivotal Software, Inc. (Hrsg.): *RestController (Spring Framework 5.0.1.RELEASE API)*. https://docs.spring.io/spring/docs/current/javadoc-api/org/springframework/web/bind/annotation/RestController.html. Version: 24.10.2017. – Stand: 14.11.2017

[o.V17n] o.V. ; Docker Inc. (Hrsg.): *Docker Documentation: Overview of Docker Compose*. https://docs.docker.com/compose/overview/. Version: 28.11.2017. – Stand: 11.11.2017

[o.V17o] o.V. ; Docker Inc. (Hrsg.): *Docker Documentation: Use volumes*. https://docs.docker.com/engine/admin/volumes/volumes/. Version: 28.11.2017. – Stand: 10.11.2017

[o.V17p] o.V. ; Docker Inc. (Hrsg.): *What is a Container*. https://www.docker.com/what-container. Version: 28.11.2017. – Stand: 10.12.2017

[Sca13] Scarioni, Carlo: *Pro Spring security*. New York : Apress / Springer, 2013

[Sch03] Schäfer, Günter: *Netzsicherheit: Algorithmische Grundlagen und Protokolle*. 1. Auflage. Heidelberg : dpunkt-Verl., 2003

[Sir14] Siriwardena, Prabath: *Advanced API Security: Securing APIs with OAuth 2.0, OpenID Connect, JWS, and JWE*. 1. Auflage. New York : Apress, 2014

[spr17a] PIVOTAL SOFTWARE, INC. (Hrsg.): *Spring Logo*. https://spring.io/img/spring-by-pivotal-9066b55828deb3c10e27e609af322c40.png. Version: 03.12.2017. – Stand: 11.12.2017

[spr17b] *Spring Boot*. https://projects.spring.io/spring-boot/. Version: 2017. – Stand: 06.11.2017

[SSH13] SAAKE, Gunter ; SATTLER, Kai-Uwe ; HEUER, Andreas: *Datenbanken - Konzepte und Sprachen*. 5. Auflage. Verlagsgruppe Hüthig Jehle Rehm, 2013

[Sta11] STARKE, Gernot: *Effektive Software-Architekturen: Ein praktischer Leitfaden*. 5. Auflage. München : Hanser, 2011

[Ste14] STEINER, René: *Grundkurs Relationale Datenbanken*. 8. Auflage. Wiesbaden : Springer Fachmedien, 2014

[Til11] TILKOV, Stefan: *REST und HTTP: Einsatz der Architektur des Web für Integrationsszenarien*. 2. Auflage. Heidelberg : dpunkt.verl., 2011

[Ull12] ULLENBOOM, Christian: *Java ist auch eine Insel*. 10. Auflage. Bonn : Galileo Press, 2012

[VB15] VARANASI, Balaji ; BELIDA, Sudha: *Spring REST*. 1. Auflage. Berkeley CA : Apress, 2015

[Voh16] VOHRA, Deepak: *Pro Docker*. 1. Auflage. Berkeley CA : Apress, 2016

[Wal16] WALLS, Craig: *Spring Boot in action*. 1. Auflage. Shelter Island, NY : Manning Publications, 2016

[Wea] WEBB, Phillip ; ET. AL. ; PIVOTAL SOFTWARE, INC. (Hrsg.): *Spring Boot Actuator: Production-ready features: Endpoints*. https://docs.spring.io/spring-boot/docs/current/reference/html/production-ready-endpoints.html. – Stand: 11.12.2017

[Web] WEBB, Phillip ; ET.AL.: *Spring Boot Reference Guide: Appendix A. Common application properties*. https://docs.spring.io/spring-boot/docs/current/reference/html/common-application-properties.html. – Stand: 22.11.2017

[We13] WINTER, Mario ; ET AL.: *Der Integrationstest: Von Entwurf und Architektur zur Komponenten- und Systemintegration*. 1. Auflage. München : Hanser, 2013

[Welo9] WELK, Kerstin: *Techniken für mobile Datenbanken in Informationssystemen*. Hamburg : Diplom.de, 2009 https://www.diplom.de/document/227158. – Stand: 11.12.2017

[Wit16] WITTE, Frank: *Testmanagement und Softwaretest: Theoretische Grundlagen und praktische Umsetzung*. 1. Auflage. Wiesbaden : Springer Vieweg, 2016

[WK11] WENK, Andreas ; KLAMPÄCKEL, Till: *CouchDB - Das Praxisbuch für Entwickler und Administratoren*. 1. Auflage. Bonn : Galileo Press, 2011

[Wol11] WOLFF, Eberhard: *Spring 3 (iX Edition): Framework für die Java-Entwicklung*. 3. Auflage. Heidelberg : dpunkt.verlag, 2011

[Wol16] WOLFF, Eberhard: *Microservices: Grundlagen flexibler Softwarearchitekturen*. 1. Auflage. Heidelberg : dpunkt.verlag, 2016

ERKLÄRUNG

„Ich versichere wahrheitsgemäß, die Arbeit selbstständig verfasst, alle benutzten Hilfsmittel vollständig und genau angegeben und alles kenntlich gemacht zu haben, was aus Arbeiten anderer unverändert oder mit Abänderungen entnommen wurde sowie die Satzung des KIT zur Sicherung guter wissenschaftlicher Praxis in der jeweils gültigen Fassung beachtet zu haben.“

Karlsruhe, 19. Dezember 2017

Lukas Struppek