Bachelorarbeit

Philipp Prögel

Dienstkomposition für kooperatives Arbeiten in der Lehre

Fakultät Technik und Informatik
Studiendepartment Informatik

Faculty of Engineering and Computer Science
Department of Computer Science

Philipp Prögel

Dienstkomposition für kooperatives Arbeiten in der Lehre

Bachelorarbeit eingereicht im Rahmen der Bachelorprüfung

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

Betreuender Prüfer: Prof. Dr. Martin Becke
Zweitgutachter: Prof. Dr. Thomas Lehmann

Eingereicht am: 27. Februar 2017

Philipp Prögel

Thema der Arbeit

Dienstkomposition für kooperatives Arbeiten in der Lehre

Stichworte

Online-Dienst, Dienst, Dienstkomposition, Node.js, Angularjs

Kurzzusammenfassung

In dieser Ausarbeitung wird eine *Middleware* für die Lehre entwickelt, die verschiedene Online-Dienste für kooperatives Arbeiten verbindet. Abschließend wird eine Webseite entwickelt, die die Funktionen der *Middleware* testet.

Philipp Prögel

Title of the paper

Service composition for cooperative work

Keywords

online-service, service, service composition, Node.js, Angularjs

Abstract

In this elaboration a middleware for teaching will be developed, which connects different online services for cooperative work. Conclusively, a website will be developed for the purpose of testing the functions provided by the middleware.

Inhaltsverzeichnis

1. Einleitung **1**
1.1. Motivation . . . 1
1.2. Problemstellung . . . 2
1.3. Zielsetzung . . . 2

2. Grundlagen **4**
2.1. Dienst . . . 4
2.2. Online-Dienst . . . 4
2.3. Dienstkomposition . . . 5
2.4. Dienstkategorien . . . 5
2.4.1. Cloud-Storage . . . 5
2.4.2. Team-Collaboration . . . 5
2.4.3. Instant-Messaging . . . 5
2.4.4. Versionsverwaltung . . . 6
2.4.5. Document-Collaboration . . . 7
2.5. SaaS . . . 7
2.6. Middleware . . . 7
2.7. Frontend . . . 7

3. *Middleware* **8**
3.1. Requirements Engineering . . . 8
3.1.1. Zielgruppenanalyse . . . 8
3.1.1.1. Aufbau . . . 9
3.1.1.2. Auswertung . . . 10
3.1.1.3. Schlussfolgerung . . . 17
3.1.2. Kriterienkatalog . . . 18
3.1.3. Dienstauswahl . . . 20
3.1.4. Funktionale Anforderungen . . . 21
3.1.5. Nichtfunktionale Anforderungen . . . 22
3.2. Architektur . . . 23
3.2.1. Monolithische Architektur . . . 23
3.2.2. Microservice Architektur . . . 23
3.2.3. Auswahl . . . 24
3.2.3.1. Vorteile . . . 25
3.2.3.2. Nachteile . . . 25
3.2.4. Architekturaufbau . . . 26

3.3. Technologie . . . 27
3.3.1. Entwicklungsplattform . . . 27
3.3.1.1. Node.js . . . 27
3.3.2. Datenbank . . . 29
3.3.2.1. MongoDB . . . 29
3.3.3. Kommunikation . . . 30
3.3.3.1. GRPC . . . 30
3.3.3.2. REST . . . 31
3.3.3.3. JSON und Protobuf . . . 31

4. Besondere Herausforderungen bei der Implementation 33
4.1. Asynchrone Programmierung . . . 33
4.2. Implementation der *File-Storage*-Dienste . . . 36
4.2.1. Vereinheitlichung . . . 36
4.2.2. Datei-Austausch . . . 38
4.3. Abstrakter *File-Storage*-Dienst . . . 39
4.4. Authentifizierung und Autorisierung . . . 42
4.4.1. OAUTH2 . . . 42
4.4.2. Basic Authentication . . . 44
4.4.3. Microservice für die Authentifizierung . . . 45
4.5. Erweiterbarkeit und Wartbarkeit der RESTful-API . . . 47

5. Anwendung der *Middleware* 50
5.1. *Frontend*-Entwicklung . . . 50
5.1.1. Anforderungen . . . 50
5.1.2. Auswahl des *Frameworks* . . . 51
5.1.3. Angular2 . . . 52
5.1.3.1. Architektur . . . 52
5.1.3.2. Just in Time Compilation . . . 54
5.1.3.3. Ahead of Time Compilation . . . 54
5.1.3.4. Lazy Loading . . . 55
5.1.4. Ergebnis . . . 55
5.2. Validierung der Anwendungsfälle . . . 59

6. Fazit 61
6.1. *Lessons Learned* . . . 61
6.2. Entwicklungsstand . . . 62
6.3. Stärken und Schwächen der Implementierung . . . 63
6.3.1. Stärken . . . 63
6.3.2. Schwächen . . . 64
6.4. Ausblick auf zukünftige Entwicklungen . . . 65

A. Zielgruppenanalyse **67**
A.1. Fragebogen 68

B. Frontend **69**
B.1. Ergebnis 69

Abkürzungsverzeichnis **71**

Listings

3.1. Beispieldatei eines GRPC Services . . . 30
3.2. Identifikation von Ressourcen in RESTful Architekturen . . . 31
3.3. Beispiel JSON Struktur . . . 31

4.1. Pseudocode Java . . . 33
4.2. Pseudocode Node.js . . . 33
4.3. Pseudocode für das serielle Ausführen mehrerer I/O-Operationen in *Node.js*. . . 34
4.4. Serielle Ausführung von I/O-Operationen mit benannten Callback-Funktionen 35
4.5. Serielle Ausführung von I/O-Operationen mithilfe eines *Promise* . . . 35
4.6. Schemata für den abstrakten *File-Storage*-Dienst . . . 41
4.7. JSON-Format für die Konfiguration eines Dienstes innerhalb der RESTful-API 48
4.8. JSON Format für einen Request . . . 48

5.1. Beispiel von *structural Directives* mit *ngIf und *ngFor . . . 53

1. Einleitung

1.1. Motivation

Aufgaben und Arbeiten während des Studiums werden immer öfter in Gruppen bearbeitet. Dabei hat die Kooperation innerhalb der Gruppe einen großen Einfluss auf das Ergebnis. Fehlende Kommunikation und planloses Vorgehen sind Hinweise auf eine ungenügende Organisation innerhalb der Gruppe, insbesondere bei Studenten.

Um dem entgegenzuwirken, wurden verschiedene Online-Dienste für das kooperative Arbeiten entwickelt. Einige für diese Ausarbeitung wichtige Online-Dienste können in folgende Kategorien eingeteilt werden:

- Die Cloud-Storage-Dienste [1, 5] bieten elementare Funktionen für die Zusammenarbeit von Gruppen. Dateien können online gespeichert und über das Internet bereitgestellt werden. Dies ermöglicht das Teilen von Dateien mit den Gruppenmitgliedern.
- Eine weitere Kategorie an Online-Diensten für kooperatives Arbeiten bilden die Instant-Messaging-Dienste [2, 80]. Mithilfe dieser können sich Gruppen in Echtzeit austauschen und aktuelle Ereignisse besprechen.
- Vor allem in der Informatik spielen die Online-Dienste für Versionsverwaltung eine elementare Rolle [3]. Sie ermöglichen koordiniertes Arbeiten mehrerer Entwickler an einer Datei und die Protokollierung von Änderungen, wodurch nachverfolgt werden kann, welches Gruppenmitglied etwas geändert hat.
- Angelehnt an die bereits angesprochenen Cloud-Storage-Dienste existieren auch die Document Collaboration-Dienste [4, 12]. Diese erlauben eine gleichzeitige und gemeinsame Bearbeitung von Dokumenten.

1.2. Problemstellung

Die kooperative Zusammenarbeit in der Gruppe bedarf also einer Vielzahl an Diensten. Dabei steht der Anwender vor mehreren Problemen. Der Nutzer verliert an Ordnung und Übersicht, weil er für jeden Dienst eine Webseite besuchen oder die Applikation auf seiner Zielplattform starten muss, um dessen Funktionen zu nutzen. Außerdem bestehen keine sinnvollen Automatismen für die Verknüpfung von Diensten, weshalb der Anwender die Daten zwischen den Diensten eigenständig synchronisieren muss.

Es stellt sich die Frage, wie diese Probleme gelöst werden können. Eine mögliche Antwort darauf ist die Dienstkomposition. Dabei werden verschiedene Funktionen von mehreren Diensten miteinander verbunden. Im Zuge dessen können Verknüpfungen zwischen den Diensten geschaffen werden, welche in der autonomen Ausführung der einzelnen Dienste nicht existent sind. Es könnte beispielsweise bei jeder Änderung an einer Datei eine Nachricht per Messenger-Dienst verschickt werden, um das Team über aktuelle Entwicklungen zu informieren. Dateien aus einem Cloud-Storage-Dienst könnten an einen Document Collaboration-Dienst weitergeleitet werden, wodurch die gemeinsame Bearbeitung an der Datei ermöglicht wird. Es sind also Verknüpfungen zwischen unterschiedlichen Dienstarten vorstellbar, die allesamt die Kooperation und Organisation innerhalb der Gruppe fördern und somit bei der Problemlösung helfen.

1.3. Zielsetzung

Ziel der vorliegenden Ausarbeitung ist die Erstellung einer Dienstkomposition für Online-Dienste in der Lehre. Dabei soll der Anwendungsbereich der Dienste die Zusammenarbeit und Organisation innerhalb einer Gruppe enthalten. Um die relevanten Online-Dienste zu identifizieren, wird im Rahmen dieser Arbeit eine Umfrage erstellt, bei der die Studenten der HAW Hamburg im Fachbereich Informatik befragt werden. Des Weiteren werden Anforderungen an die Dienstkomposition aus den Umfrageergebnissen abgeleitet. Ein Kriterienkatalog für die Zusammensetzbarkeit von Online-Diensten ist ebenfalls Teil dieser Ausarbeitung. Dadurch wird festgestellt, ob sich ein Dienst für die Komposition eignet.

Für die Dienstkomposition wird zunächst eine *Middleware* konzipiert und entwickelt. Die *Middleware* hat die Aufgabe, einzelne Funktionen der Online-Dienste bereitzustellen. Die benötigten Daten werden durch verschiedene Web-Schnittstellen von den Online-Diensten zur Verfügung gestellt. Eine besondere Designvorgabe an die *Middleware* ist eine wiederverwend-

bare und plattformunabhängige Lösung. Im Zuge dessen können zukünftige Arbeiten, die sich mit dem Thema Dienstkomposition befassen, auf die *Middleware* zurückgreifen und ihre Arbeit darauf aufbauen.

Abschließend wird mithilfe der implementierten *Middleware* eine Webseite entwickelt. Auf dieser sollen die Funktionen der Dienste, die Teil der *Middleware* sind, bereitgestellt werden.

2. Grundlagen

In diesem Kapitel werden wichtige Begrifflichkeiten im Zusammenhang mit dieser Ausarbeitung diskutiert.

2.1. Dienst

Um im nachfolgenden Abschnitt den Begriff Online-Dienst definieren zu können, muss zunächst die übergeordnete Kategorie Dienst für sich betrachtet werden. Ein Dienst beschreibt in der Informatik zumeist ein in sich geschlossenes System, welches zusammenhängende Funktionen eines Themenfeldes bündelt und mithilfe einer Schnittstelle zur Verfügung stellt [5, 3].
Dienste können in unterschiedliche Arten beziehungsweise Aufgabenfelder eingeteilt werden, zum Beispiel Systemdienste, Online-Dienste oder auch Netzwerkdienste. Dabei bestimmt die Art des Dienstes, auf welche Weise die Schnittstelle erreichbar ist. Ein Systemdienst bietet seine Schnittstelle für Dienste auf dem jeweiligen System an, ein Netzwerkdienst für das gesamte Netzwerk und ein Online-Dienst ist von überall durch das Internet zu erreichen. Die Spezifikationen eines Dienstes können dabei in verschiedenen Bereichen vorgenommen werden, wie zum Beispiel Sicherheit, Zuverlässigkeit oder *Performance* [6, 89].

2.2. Online-Dienst

Online-Dienst ist ein noch unzureichend diskutierter Begriff in der Informatik. Um einer Definition näher zu kommen, können zunächst die Begriffe *Online* und *Dienst* für sich betrachtet werden. *Online* bedeutet für einen Benutzer, dass er sowohl aktiv als auch passiv mit einem Empfangsgerät mit dem Internet verbunden ist [7]. In Verbindung mit dem Begriff *Dienst* (siehe 2.1) bedeutet es, dass ein *Online-Dienst* seine Funktionen über das Internet bereitstellt und Nutzer *online* sein müssen, um diesen zu nutzen.

2.3. Dienstkomposition

Eine Dienstkomposition beschreibt den Zusammenschluss mehrerer Dienste zu einer neuen Anwendung oder auch einem neuem Geschäftsprozess. Dabei kann die entstandene Komposition wieder als neuer Dienst verstanden werden [8, 4]. Idealerweise weist die Komposition einen Mehrwert auf, der aus der Wechselwirkung der Dienste resultiert und insofern nicht isoliert durch einen einzelnen Dienst der Komposition erzielt werden kann [9]. Eine Dienstkomposition kann sowohl proaktiv als auch reaktiv aufgebaut sein. Proaktiv bedeutet, dass die Dienste, aus denen sich die Komposition zusammensetzt, vor der Laufzeit bekannt sind, wohingegen eine reaktive Komposition die Dienste dynamisch zusammenstellt [8, 5].

2.4. Dienstkategorien

Um verschiedene Dienste im Laufe der Ausarbeitung zusammenfassen und kategorisieren zu können, werden in diesem Abschnitt einige Dienstkategorien eingeführt. Die Kategorien wurden anhand der in 3.1.1 vorgestellten Umfrage ausgewählt, weil die meisten der angegebenen Online-Dienste den gewählten Dienstkategorien zuzuordnen sind.

2.4.1. Cloud-Storage

Bei *Cloud-Storage*-Diensten werden Dateien auf *Server* abgelegt, die über das Internet zu erreichen sind [1]. Der Zugriff auf die Dateien findet wie in einem klassischen Dateisystem statt. Der Vorteil liegt hier bei der Erreichbarkeit. Der Anwender kann von überall auf seine Daten zugreifen und sie bearbeiten. Des Weiteren bieten die meisten *Cloud-Storage*-Dienste das Teilen von Dateien an. Dadurch wird das kooperative Arbeiten in Teams ermöglicht.

2.4.2. Team-Collaboration

Team-Collaboration ist ein noch unzureichend diskutierter Begriff in der Wissenschaft. Diese Art von Diensten versucht die Zusammenarbeit in Teams zu vereinfachen. Dabei werden verschiedene Funktionen wie zum Beispiel Terminplanung, Kommunikation oder Speicherung von Dateien auf einer Plattform angeboten. Ein *Team-Collaboration*-Dienst kann also bereits als eine Dienstkomposition verstanden werden

2.4.3. Instant-Messaging

Instant-Messaging-Dienste ermöglichen den Nachrichten-Austausch in Echtzeit. Dabei können oft auch Gruppen-Gespräche gestartet werden, um die Kommunikation innerhalb einer Gruppe

zu erleichtern [2]. In Abbildung 2.1 ist der Versand von Short Message Service (SMS) im Vergleich zu *Instant-Messaging* Nachrichten zu sehen.

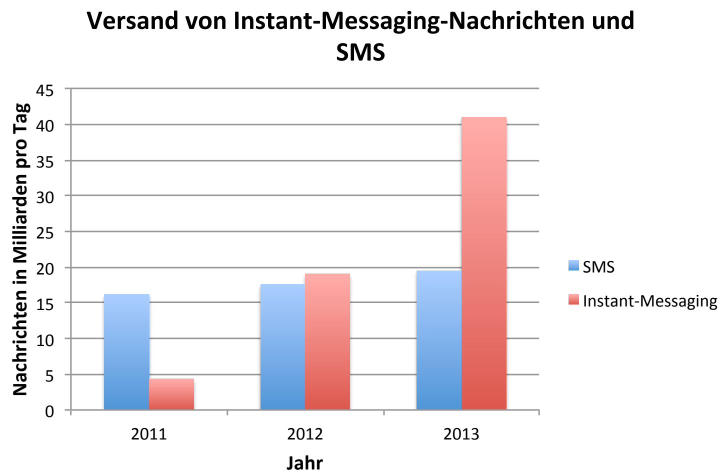


Abbildung 2.1.: Versand von Instant-Messaging-Nachrichten und SMS weltweit 2011 und 2012 und Prognose für 2013 (in Milliarden pro Tag) [10]

Der Versand von *Instant-Messaging* Nachrichten ist im Jahr 2012 im Vergleich zu 2011 um circa das Vierfache gestiegen. Während das Versenden von SMS in der Anzahl stagniert, ist ein klarer Wachstum für den Versand von *Instant-Messaging* Nachrichten zu erkennen. *Instant-Messaging*-Dienste gewinnen also immer mehr an Bedeutung.

2.4.4. Versionsverwaltung

Mit Hilfe der Versionsverwaltung werden Änderungen an Dateien und Dokumenten gespeichert. Dies ermöglicht ein Zurücksetzen auf ältere Versionen und die Zusammenarbeit mehrerer Entwickler. Die Dateien werden in einem sogenannten *Repository* abgelegt. Eine besondere Form der Versionsverwaltung ist die verteilte Versionsverwaltung. Dabei ist das *Repository* dezentralisiert und ermöglicht Entwicklern *offline* zu arbeiten [11]. Besonders in der Zusammenarbeit unterscheiden sich Dienste der Versionsverwaltung von *Cloud-Storage*-Diensten. Durch sogenannte *Branches*, kann die Entwicklung von Projekten in verschiedene Zweige aufgeteilt werden, und fördert somit die kooperative Arbeit in Teams.

2.4.5. Document-Collaboration

Document-Collaboration ermöglicht das gemeinsame und gleichzeitige Bearbeiten von Dokumenten. Anders als bei *Cloud-Storage*-Diensten steht hier die gleichzeitige Bearbeitung in Echtzeit im Vordergrund [12]. Dadurch kann schon beim Editieren von Dateien auf die Änderungen eingegangen werden, anstatt zu einem späteren Zeitpunkt.

2.5. SaaS

SaaS oder *software as a service* beschreibt die Bereitstellung von Software als Dienst über das Internet. SaaS steht damit im Kontrast zu dem klassischen Modell, Software vor Ort für Kunden zu installieren und dort betreiben zu lassen. Denn die Idee hinter SaaS ist, einen Dienst als Abo-Modell oder nutzungsabhängig zu monetarisieren, und den Zugriff über das Internet, zum Beispiel durch einen Internet-*Browser*, zu ermöglichen. Dadurch kann der SaaS-Anbieter die Sicherheit und Verfügbarkeit der Daten und Anwendungen garantieren. [13]

2.6. Middleware

Eine *Middleware* beschreibt eine Klasse von *Software*-Systemen, die dazu entwickelt wurde, komplexe und heterogene verteilte Systeme zu verwalten. Einzuordnen ist die *Middleware* über dem Betriebssystem und unter der eigentlichen Applikation. Eine *Middleware* hat dabei die Aufgabe, Funktionen zu abstrahieren und diese verteilten Systemen bereitzustellen. Nutzer der *Middleware* können die angebotenen Funktionen nutzen, ohne sich mit den Technologien unter der *Middleware* auseinander setzen zu müssen. [14]

2.7. Frontend

Ein *Frontend* bezeichnet eine Benutzerschnittstelle mit einer grafischen Oberfläche [15]. Dabei kommuniziert das *Frontend* mit einem Server der die Daten liefert, welcher oftmals in diesem Zusammenhang auch als *Backend* bezeichnet wird.

3. *Middleware*

Dieses Kapitel befasst sich mit der Umsetzung der *Middleware* für die Dienstkomposition. Es werden Anforderungen aufgestellt, eine Übersicht und Auswahl zur Technologie besprochen und die Architektur definiert.

3.1. Requirements Engineering

3.1.1. Zielgruppenanalyse

Im Rahmen der Ausarbeitung wurde eine Umfrage durchgeführt, mit folgenden Zielen:

- Online-Dienste identifizieren, die Studenten für kooperatives Arbeiten nutzen
- Auswahl der zu komponierenden Online-Dienste anhand der Auswertung
- Anforderungen für die *Middleware* ableiten
- Anforderungen für das beispielhafte *Frontend* ableiten

Die Grundgesamtheit [16, 146] für die Befragung bilden die Studenten der Fachrichtung Informatik an der HAW Hamburg. Durch die Einschränkung auf eine Fachrichtung soll der Umfang an infrage kommenden Online-Diensten verringert werden. In Abbildung 3.1 ist die Verteilung der Probanden auf die jeweiligen Studiengänge zu sehen.

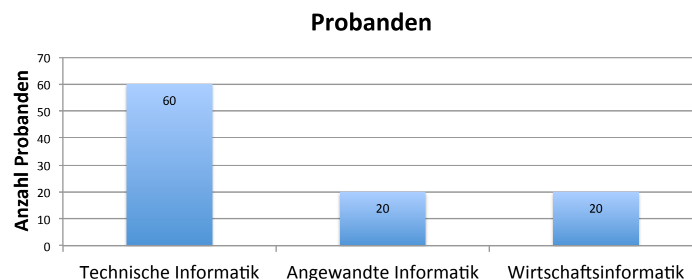


Abbildung 3.1.: Verteilung der Probanden

Insgesamt wurden 100 Studenten befragt, wobei die Fachrichtung Technische Informatik mit 60 Studenten am stärksten vertreten ist. Angewandte Informatik und Wirtschaftsinformatik sind mit jeweils 20 Studenten vertreten. Im Sommersemester 2016 waren insgesamt 968 Studenten in den erwähnten Studiengängen [17] eingeschrieben. Also wurden 9,68% der infrage kommenden Probanden abgedeckt.

3.1.1.1. Aufbau

Am Anfang des Fragebogens wurden Studienfach und Fachsemester des Probanden abgefragt. In die erste Spalte des Fragebogens konnten die Probanden Online-Dienste eintragen, die sie im Rahmen ihres Studiums für kooperatives Arbeiten nutzen. Dies war nötig, weil die Anzahl der verfügbaren Online-Diensten für kooperatives Arbeiten zu groß für eine Vorauswahl ist. Einzig die Online-Dienste *Owncloud* und *EMIL* wurden als Vorauswahl angegeben, weil diese im Rahmen eines Studiums an der HAW Hamburg in der Fachrichtung Informatik vorgegeben werden. Zu jedem Online-Dienst wurden die gleichen sieben Fragen gestellt, mit Einfach- oder Mehrfachauswahl. Dabei sollten folgende Dimensionen im Nutzungsverhalten untersucht werden:

- Häufigkeit: Ein Online-Dienst der von vielen Probanden angegeben, aber nur selten genutzt wird, weißt unter Umständen eine geringere Gesamtnutzung auf, als ein Online-Dienst, der nicht so oft angegeben, aber täglich genutzt wird. Selten genutzte Online-Dienste können identifiziert und möglicherweise aussortiert werden.

- Zufriedenheit: Eine hohe Zufriedenheit bei der Benutzung eines Online-Dienstes kann darauf hindeuten, dass der Dienst auch noch eine längere Zeit genutzt wird. Wohingegen Unzufriedenheit zurückgehende Nutzerzahlen prognostizieren kann.

- Antrieb: Der Antrieb soll in Verbindung mit anderen Eigenschaften im Nutzungsverhalten gesehen werden. Zum Beispiel ob Probanden mit aufgezwungenen Online-Diensten unzufrieden sind. Oder wie stark die Wahl der genutzten Online-Dienste auf den eigenen Antrieb zurückzuführen sind.

- Umfeld: Beim Umfeld soll untersucht werden, wo Online-Dienste genutzt werden. Eher im privatem Umfeld, an der Hochschule oder auch im Beruf.

Des Weiteren sollte das Gerät für die Nutzung des Online-Dienstes angegeben sowie differenziert werden, ob über den *Browser* oder eine Applikation auf den Online-Dienst zugegriffen wird. Zum Schluss sollte für jeden Dienst angegeben werden, ob der Befragte Zugriffsrechte an

Drittanbieter freigeben würde. Diese Frage durfte auch mit „Weiß nicht" beantwortet werden, weil die Beantwortung unter anderem von der Sensibilität der hinterlegten Daten abhängig ist. Der vollständige Fragebogen liegt als Anlage bei (A.1).

3.1.1.2. Auswertung

Im folgenden Abschnitt werden die Ergebnisse der Auswertung vorgestellt. Dabei werden neben relevanten Ergebnissen vor dem Hintergrund dieser Ausarbeitung auch interessante Ergebnisse für zukünftige Arbeiten dargelegt. In Abbildung 3.2 ist die Korrelation zwischen dem Fachsemester des Studierenden und der Anzahl der von ihm genutzten Online-Dienste zu sehen.

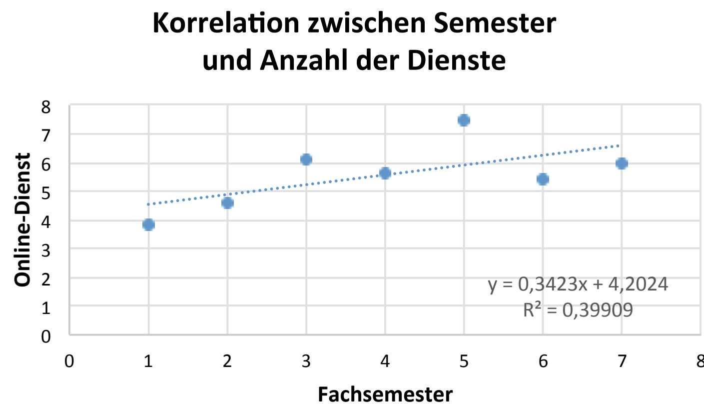


Abbildung 3.2.: Korrelation zwischen Fachsemester und Anzahl der genutzten Online-Dienste

Ein Student im ersten Semester nutzt durchschnittlich vier Online-Dienste für kooperatives Arbeiten. Bis zum fünften Fachsemester steigt diese Anzahl auf ungefähr sieben an, während im sechsten und siebten Semester circa sechs Online-Dienste verwendet werden. Die aufgeführte Auswertung und das Ergebnis der Korrelationsformel nach Pearson [18, 85] zeigen eine Korrelation zwischen Fachsemester und Anzahl der Online-Dienste. Je höher das Fachsemester, desto höher die Anzahl der genutzten Online-Dienste. Ein Grund für diese Kausalität könnten wechselnde Gruppen-Zusammensetzungen im Verlauf des Studiums sein. Dienste werden untereinander besprochen und ausgetauscht. Der Rückgang von genutzten Diensten im sechsten und siebten Semester kann mit dem reduziertem Aufkommen von Gruppenaufgaben erklärt werden. Aufgrund von Bachelorarbeiten belegen die Studenten weniger Kurse und haben infolgedessen eine geringere Anzahl an kooperativen Arbeiten, bei denen auf Online-Dienste

zurückgegriffen werden könnte.

Wie im Abschnitt 3.1.1.1 erklärt, sollte jeder Befragte die von ihm genutzten Online-Dienste für kooperatives Arbeiten auflisten. In der Abbildung 3.3 sind alle angegebenen Online-Dienste und der zugehörige Anteil der Nutzer in Prozent zu sehen.

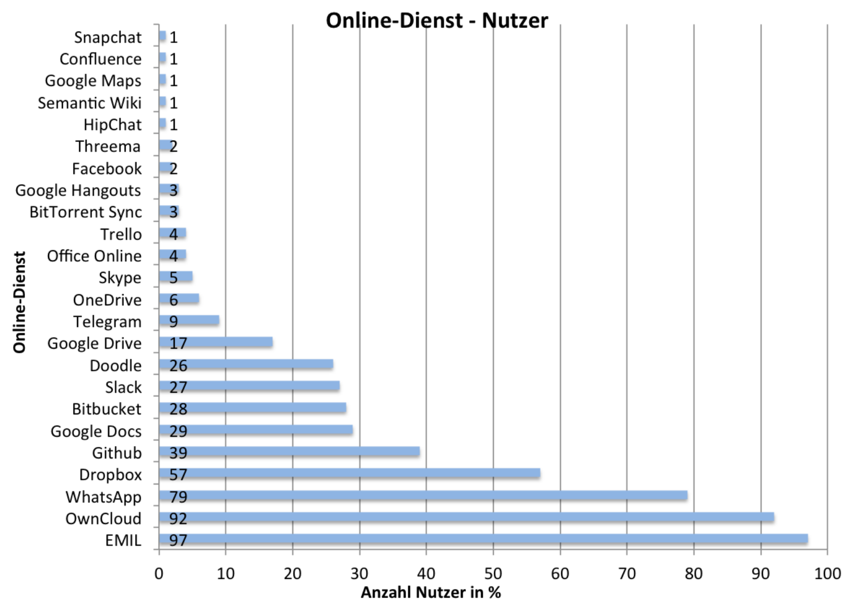


Abbildung 3.3.: Liste der Online-Dienste und die Anzahl der Nutzer in Prozent

Insgesamt wurden 23 verschiedene Online-Dienste für kooperatives Arbeiten angegeben. Die beiden von der HAW Hamburg vorgegeben Online-Dienste, *Owncloud* und *EMIL*, werden von 97% beziehungsweise 92% der Informatik Studenten genutzt. Einige Studenten verzichten also auf die vorgegebenen Online-Dienste. Der *Instant-Messaging*-Dienst *WhatsApp* ist mit 79% der am meisten genutzte Online-Dienst, der nicht vorgegeben wurde. Der am häufigsten genutzte Online-Dienst für Versionsverwaltung ist *Github* mit 39%. *Google Docs* ist mit 29% der am weitesten verbreitete *Document-Collaboration*-Dienst unter den Informatikstudenten der HAW Hamburg.

Die angegebene Zufriedenheit bei der Verwendung der jeweiligen Online-Dienste ist in Abbildung 3.4 zu sehen. Als Antwortmöglichkeiten bei der Evaluation wurde eine Skala von 1-5 gewählt, bei der eine Eins zufrieden und eine Fünf unzufrieden bedeutet. Für die Auswertung wurde der Durchschnittswert für jeden Online-Dienst ermittelt und danach in einen

Prozentwert umgeformt. Je höher der Prozentwert, desto besser wurde die Zufriedenheit beim Umgang mit dem jeweiligen Online-Dienst bewertet. Sortiert sind die Online-Dienste nach der Anzahl an Nutzern (siehe 3.3), wobei Dienste mit weniger als fünf Angaben nicht berücksichtigt wurden.

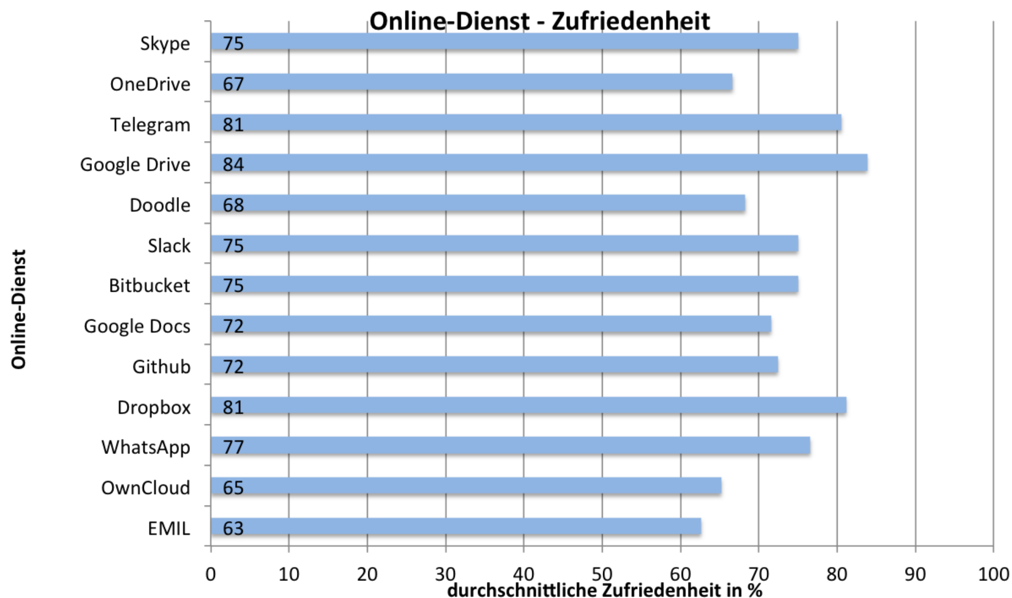


Abbildung 3.4.: Zufriedenheit bei der Verwendung der jeweiligen Online-Dienste

Generell ist zu erkennen, dass die Probanden mit allen Online-Diensten die sie benutzen auch zufrieden sind. Am wenigsten zufrieden sind die Befragten beim Umgang mit *EMIL* und *Owncloud*, die von dem Department Informatik vorgegebenen werden. Ob die Qualität der Dienste oder die Vorschrift, diese Online-Dienste zu verwenden, der Grund für die geringere Zufriedenheit ist, kann mithilfe der Umfrage nicht abschließend geklärt werden.

Wie häufig ein Online-Dienst genutzt wird ist in Abbildung 3.5 dargestellt. Als Antwortmöglichkeiten wurden täglich, wöchentlich und seltener genutzt. Um die Ergebnisse besser einordnen zu können, wird die Häufigkeit als Prozentzahl dargestellt. Je höher der Prozentwert, desto häufiger wird der jeweilige Online-Dienst genutzt. Ein Ergebnis von 100% würde zum Beispiel bedeuten, dass der Dienst ausschließlich täglich genutzt wird, also sehr häufig.

Am häufigsten werden *Instant-Messaging*-Dienste genutzt. Denn *Whatsapp* und *Telegram* haben mit 98% beziehungsweise 89% die beiden höchsten Werte und werden fast ausschließlich täglich genutzt. Im Vergleich dazu werden Dienste für die Versionsverwaltung, wie *Github*

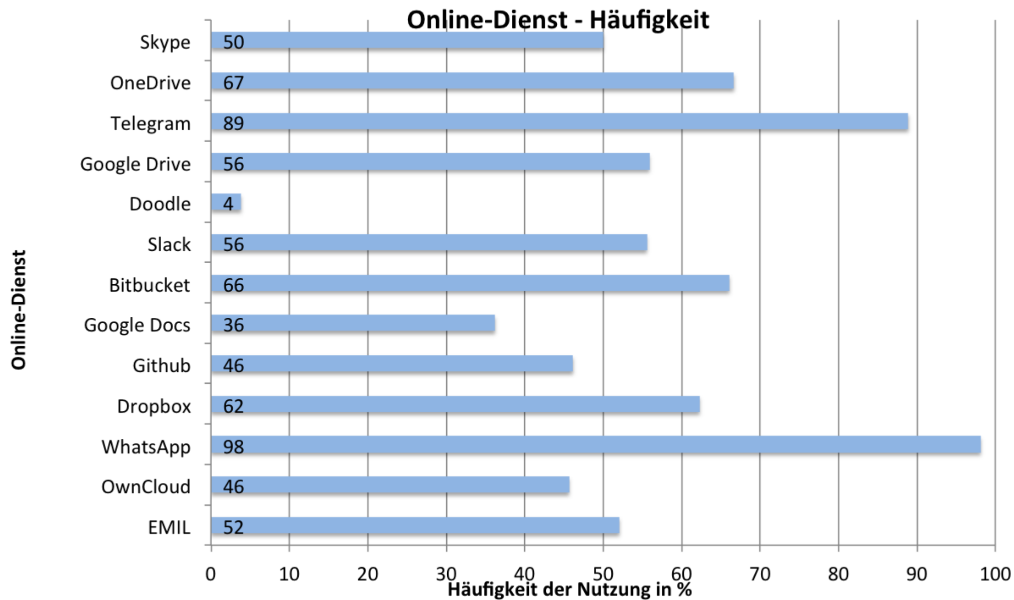


Abbildung 3.5.: Häufigkeit mit der ein Online-Dienst genutzt wird

und *Bitbucket*, eher auf einer wöchentlichen Basis genutzt. Auffallend ist *Doodle*, ein Dienst zur Terminfindung, der mit einem Wert von 4% seltener als wöchentlich genutzt wird. Eine Erklärung dafür könnte eine geringe Anzahl an Szenarien sein, bei der ein Dienst zur Terminfindung nötig ist.

Nachdem die Online-Dienste für sich betrachtet wurden, ist in Abbildung 3.6 eine Aufteilung der angegebenen Online-Dienste in Dienstkategorien zu sehen, um eine bessere Übersicht zu schaffen. Verwendet werden die im Abschnitt 2.3 eingeführten Kategorien. Die Prozentwerte beziehen sich auf die 534 im Zuge der Umfrage angegeben Online-Dienste. Jeder Dritte angegebene Online-Dienst ist ein *Cloud-Storage*-Dienst. Damit sind die *Cloud-Storage*-Dienste die am häufigsten vertretene Dienstkategorie. *Instant-Messaging*-Dienste wurden mit 24% am zweithäufigsten angegeben. Ebenfalls relevant sind *Document-Collaboration* und Versionsverwaltungs-Dienste mit 19 respektive 13 Prozent. Die Kategorien *Document-Collaboration* und Sonstige haben mit jeweils 6% eine untergeordnete Bedeutung. In die Kategorie Sonstige fallen Dienste, deren Anwendungsbereich spezieller ist und daher nicht in die anderen fünf Kategorien eingeordnet werden können. Beispiele hierfür sind *Doodle* [19], ein Dienst für Terminfindung, und *Google Maps*.

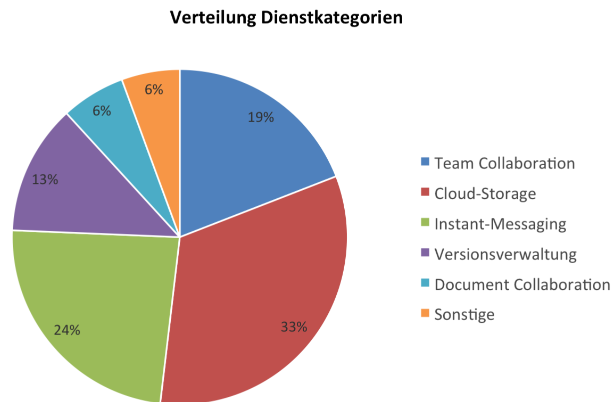


Abbildung 3.6.: Verteilung der angegebenen Dienste auf Kategorien

Nachdem die Verteilung der Online-Dienste auf Dienstkategorien untersucht wurde, stellt sich die Frage, welche Kombinationen von Dienstkategorien ein Informatikstudent der HAW Hamburg nutzt. In Abbildung 3.7 ist zunächst dargestellt, wie viel Prozent der Studenten einen oder mehrere Dienste der jeweiligen Dienstkategorie nutzt. Dabei werden anhand von Abbildung 3.6 nur die vier am stärksten vertretenen Dienstkategorien untersucht.

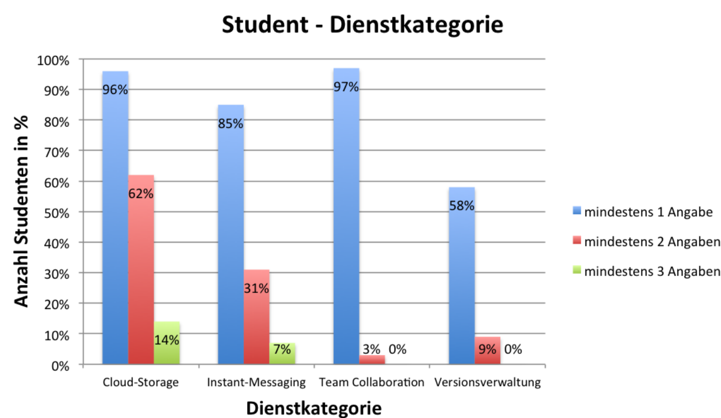


Abbildung 3.7.: Prozentuale Verteilung der jeweiligen Dienstkategorie

97% und 96% der Studenten nutzen mindestens einen *Team-Collaboration*-Dienst beziehungsweise *Cloud-Storage*-Dienst. Dies ist vor allem auf die beiden Dienste *EMIL* und *Owncloud* zu-

rückzuführen, die von der HAW Hamburg vorgegeben werden und somit erwartbar zu einem hohen Nutzungsanteil ihrer Dienstkategorie beitragen (siehe 3.3). Im Falle der *Cloud-Storage*-Dienste ist die Dichte an Diensten allerdings deutlich höher, denn 62% der Befragten nutzen mehr als einen *Cloud-Storage*-Dienst. Im Vergleich dazu nutzen nur 3% der Befragten mehr als einen *Team-Collaboration*-Dienst. Mindestens einen *Instant-Messaging*-Dienst nutzen 85% der Befragten. Bei dieser Kategorie ist ebenfalls eine höhere Dienstdichte zu beobachten, denn 31% nutzen mindestens zwei Dienste dieser Art. 58% der Befragten nutzen mindestens einen Dienst für die *Versionsverwaltung*. Bei der *Versionsverwaltung* macht ein Befragter meistens von nur einem Dienst Gebrauch, nur 9% nutzen mindestens zwei Dienste dieser Kategorie.

Nachdem dargelegt wurde, wie viele Dienste einer Kategorie die Probanden jeweils nutzen, kann außerdem die Kombination an Diensten untersucht werden. Dies ist in Abbildung 3.8 dargestellt.

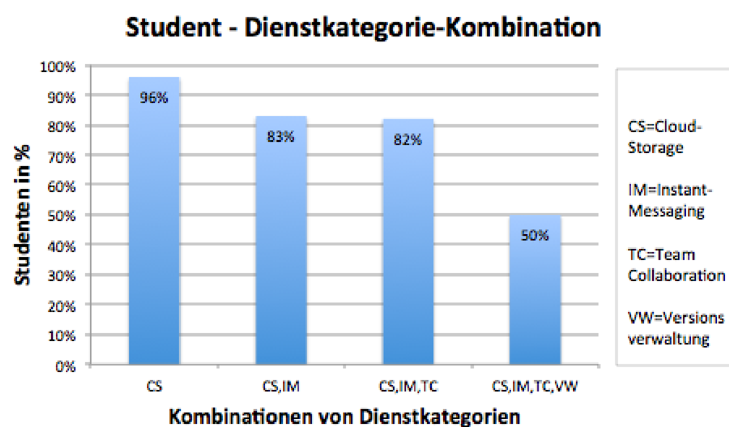


Abbildung 3.8.: Verteilung von Dienstkategorie-Kombinationen auf die Befragten

Als Ausgangspunkt für die Kombinationen soll die Kategorie *Cloud-Storage* dienen, weil sie am häufigsten angegeben wurden (siehe 3.6). Eine Kombination aus *Cloud-Storage*- und *Instant-Messaging*-Diensten nutzen 83% der Befragten. Eine Erweiterung dieser Kombination um *Team-Collaboration*-Dienste nutzen 82% der Befragten. Eine Kombination aus allen in der Abbildung 3.6 dargestellten Kategorien nutzen genau die Hälfte der Befragten. Die Informatikstudenten der HAW Hamburg nutzen also nicht nur Dienste für einen Anwendungsbereich, sondern Dienste aus verschiedenen Bereichen um kooperativ arbeiten zu können.

Wichtig für die Entwicklung der *Middleware* sind die vom Benutzer abzugebenden Zugriffsrechte. In Abbildung 3.9 ist zu sehen, welche Zugriffsrechte die Probanden einem Drittanbieter freigeben würden. Mit Drittanbietern sind Plattformen gemeint, die mithilfe der vom jeweiligen Online-Dienst bereitgestellten API, also mit externen Daten, einen eigenen Dienst anbieten.

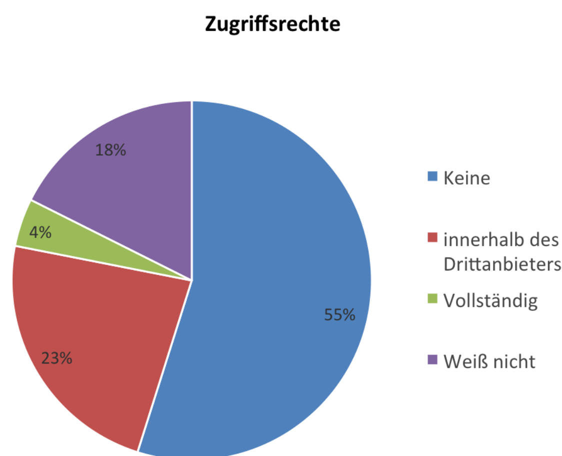
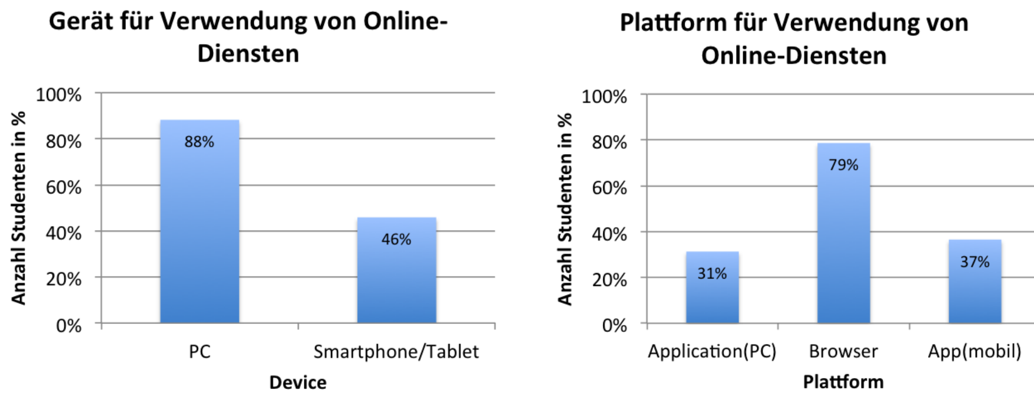


Abbildung 3.9.: Übersicht der Zugriffsrechte, die Befragte gegenüber eines Drittanbieters freigeben würden

55% der Befragten wollen einem Drittanbieter **keinen** Zugriff auf die jeweiligen Dienste freigeben. 23% hingegen würden den Zugriff **innerhalb des Drittanbieters** freigeben, also auf Daten und Dateien, die im Rahmen des Dienstes der die Rechte anfordert erstellt werden. Die im Rahmen der Ausarbeitung zu entwickelnde *Middleware* kann auch als Drittanbieter verstanden werden. **Weiß nicht** haben 18% der Befragten ausgewählt. Und **vollständigen** Zugriff auf die jeweiligen Dienste würden 4% der Befragten freigeben. Es sind also nur 27% der Befragten bereit, Zugriffsrechte oder Teile von diesen freizugeben. Zusammen mit dem hohen Anteil der Antwort **"Weiß nicht"** ist zu erschließen, dass der Aspekt der Sicherheit im Zusammenhang mit Online-Diensten noch wenig diskutiert ist.

Nachdem die einzelnen Online-Dienste und deren Kategorien diskutiert wurden, werden im Folgenden die Plattform und das Gerät für die Verwendung der Online-Dienste untersucht. Dazu sind in Abbildung 3.10 zwei Diagramme dargestellt. Die Angaben beziehen sich

auf alle von den Befragten angegeben Dienste. Beide Diagramme basieren auf Fragen mit Mehrfachauswahlen.



(a) Gerät für die Verwendung der Online-Dienste (b) Plattform für die Verwendung der Online-Dienste

Abbildung 3.10.: Zusammenfassung der Verwendungsarten von Online-Dienste

88% der angegeben Online-Dienste werden mit einem Personal Computer (PC) genutzt, wohingegen 46% der Befragten eine mobile Variante nutzen, um auf die angegeben Online-Dienste zuzugreifen. Die Internet-Browser stellen mit anteilig 79% die primäre Plattform für die Verwendung von Online-Diensten dar. Ein Grund dafür könnte die Plattformunabhängigkeit sein. Sie erlaubt die Verwendung der jeweiligen Online-Dienste auf verschiedenen Geräten, ohne Einfluss auf die vertraute Benutzerumgebung zu nehmen. Mobile *Apps* und die Applikation am PC verwenden 37% beziehungsweise 31% der Befragten.

3.1.1.3. Schlussfolgerung

In diesem Abschnitt werden die Schlussfolgerungen der Auswertung (3.1.1) formuliert und gesammelt.

- Die Befragten nutzen mehr Online-Dienste, je weiter sie in ihrem Studium fortgeschritten sind. Allerdings sinkt der Nutzungsumfang von Online-Diensten ab dem sechsten Fachsemester [3.2].
- Die Befragten sind mit der Verwendung ihrer genutzten Online-Dienste zufrieden, wobei die vom Department Informatik vorgegebenen Online-Dienste etwas schlechter bewertet wurden [3.4].

- *Instant-Messaging*-Dienste werden als einzige Dienstkategorie täglich genutzt. Andere Dienstarten werden eher wöchentlich in Anspruch genommen [3.5].
- *Cloud-Storage*, *Instant-Messaging*, *Team-Collaboration* und *Versionsverwaltung* sind die am häufigsten angegeben Dienstkategorien [3.6]. *Document-Collaboration*-Dienste hingegen werden von einem geringen Anteil der Befragten genutzt.
- Viele der Befragten nutzen mehrere *Cloud-Storage*-Dienste, wohingegen für andere Dienstarten zumeist auf einzelne Dienste zurückgeriffen wird [3.7].
- Die Befragten nutzen eine Reihe von Diensten verschiedener Anwendungsbereiche, um kooperativ in der Lehre zu arbeiten. Eine Kombination von *Cloud-Storage*, *Instant-Messaging*, *Team-Collaboration* und *Versionsverwaltungs*-Diensten nutzen die Hälfte der Befragten [3.8].
- Zugriffsrechte auf die Online-Dienste möchten die meisten Befragten nicht freigeben. Der hohe Anteil an „Weiß nicht“-Antworten lässt vermuten, dass eine Aufklärung bezüglich der Freigabe von Zugriffsrechten seitens der Dienstanbieter sinnvoll ist [3.9].
- Die meisten Online-Dienste werden mit dem PC genutzt, mobile Geräte wie *Smartphones* und *Tablets* kommen auch häufig zum Einsatz [3.10].
- Die führende Plattform für die Verwendung der Online-Dienste ist der *Browser* [3.10].

3.1.2. Kriterienkatalog

In diesem Abschnitt wird ein Kriterienkatalog festgelegt, der Online-Dienste dahingehend untersucht, inwieweit sie sich für eine Dienstkomposition eignen. Mit Hilfe des Kriterienkatalogs können Online-Dienste nach einem einheitlichen Schema bewertet und miteinander verglichen werden. Als Orientierung für den Kriterienkatalog dient die Arbeit „Ein Kriterienkatalog zur Bewertung von Anforderungsspezifikationen“ [20]. Übernommen wurde die Methodik für den Aufbau einzelner Kriterien:

- Einzelne Kriterien sind der Übersicht halber in Kategorien eingeteilt.
- Jedes Kriterium wird mit dem gleichen Schema beschrieben.
- Das Schema eines Kriteriums hat folgende Eigenschaften:
  - Eindeutige ***ID***

- Eine **Beschreibung**, um das Kriterium zu erläutern und die Abhängigkeiten zu anderen Kriterien darzustellen.
- Eine **Motivation** für die Eignung des Kriteriums.
- Die **Ausführung** der Bewertung des Kriteriums. Hier wird beschrieben welche Schritte nötig sind um die Erfüllung des Kriteriums zu prüfen.
- Die **Bewertung**, in wie weit das Kriterium erfüllt ist. Dabei werden die Skalenelemente *voll erfüllt*, *überwiegend erfüllt*, *teilweise erfüllt*, und *nicht erfüllt* verwendet.

In Abbildung 3.11 ist eine Übersicht des Kriterienkatalogs dargestellt.

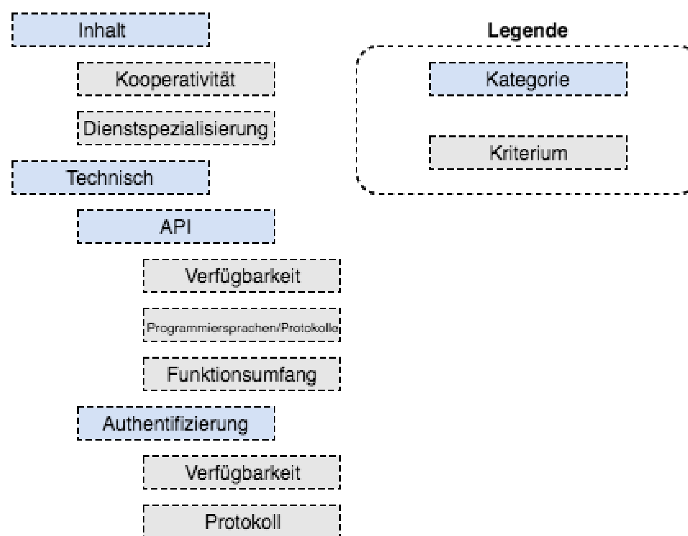


Abbildung 3.11.: Übersicht der Gliederung des Kriterienkatalogs

Die oberste Kategorieebene bilden Inhalt und Technik. Der Inhalt des zu komponierenden Dienstes muss mit anderen Diensten in Verbindung zu bringen sein, um einen Mehrwert schaffen zu können. Des Weiteren ist es von Vorteil, wenn ein Online-Dienst spezialisiert auf einen Dienst ist und nicht selbst bereits eine Komposition von Diensten anbietet. Dies erleichtert die Eingliederung in eine Komposition, weil der Online-Dienst einen klar definierten Anwendungsbereich hat.
Auf der technischen Seite ist ein Application Programmable Interface vorauszusetzen, mit der auf die Daten des Dienstes zugegriffen werden kann. Wie viele Funktionen die API anbietet und mit welchen Programmiersprachen darauf zugegriffen werden kann, ist auch Teil des

Kriterienkatalogs. Ebenfalls im Kriterienkatalog enthalten ist die Authentisierung an der Application Programmable Interface (API), die hinsichtlich der Sicherheit bewertet wird.

3.1.3. Dienstauswahl

In diesem Abschnitt werden die Online-Dienste ausgewählt, die innerhalb der *Middleware* implementiert werden und die Dienstkomposition bilden. Sowohl die Schlussfolgerung aus der Zielgruppenanalyse, als auch die Ergebnisse des Kriterienkatalogs werden für die Auswahl der Online-Dienste genutzt. Für eine Vorauswahl wurde zunächst in Betracht gezogen, die Online-Dienste anhand einer Gewichtung von Nutzerzahlen, Zufriedenheit und Häufigkeit der Verwendung einzuordnen. Weil viele der Online-Dienste von nicht mal 10% der Grundgesamtheit angegeben wurden, würde die Gewichtung für Zufriedenheit und Häufigkeit überproportional stark ausfallen. Aus diesem Grund werden die Nutzerzahlen priorisiert und als einziges Kriterium für die Vorauswahl verwendet. Alle Dienste, die von mehr als 10% der Probanden angegeben wurden, werden in die Vorauswahl aufgenommen. Die Vorauswahl wiederum wird sowohl mithilfe des Kriterienkatalogs (3.11), als auch anhand der Schlussfolgerungen der Zielgruppenanalyse untersucht. Das Ergebnis der Prüfung durch den Kriterienkatalog ist in Abbildung 3.12 zu sehen.

Kriterium	EMIL	Owncloud	Whatsapp	Dropbox	Github	Google Docs	Bitbucket	Slack	Doodle	Google Drive
Inhalt										
Kooperativität										
Dienstspezialisierung										
Technik										
API										
Verfügbarkeit										
Protokolle										
Funktionsumfang										
Authentifizierung										
Verfügbarkeit										
Protkoll										

Legende	keine Angabe
voll erfüllt	teilweise erfüllt
überwiegend erfüllt	nicht erfüllt

Abbildung 3.12.: Bewertung der Online-Dienste anhand des Kriterienkatalogs

Die Bewertung der Online-Dienste anhand des Kriterienkatalogs und der Schlussfolgerungen der Zielgruppenanalyse führen zu folgenden Entscheidungen:

- *EMIL* bietet als *Team-Collaboration*-Dienst viele verschiedene Funktionen an, wodurch eine **Dienstspezialisierung** nicht erfüllt ist. Außerdem wird keine API angeboten.
- *WhatsApp* eignet sich inhaltlich sehr für eine Dienstkomposition. Da *WhatsApp* keine API bereitstellt und somit keine Möglichkeit für einen Zugriff auf die Daten besteht, kann *WhatsApp* nicht Teil der Komposition sein.
- *Google Docs* bietet eine Spezialisierung auf *Document-Collaboration*-Funktionen und auch eine API. Weil die Implementierung der Basisfunktionen von dieser Art von Dienst zu viel Zeit beansprucht, wird *Google Docs* nicht Teil der *Middleware* sein.
- *Doodle* bietet als Dienst zur Terminfindung eine Dienstspezialisierung an, die sich gut mit anderen Diensten in einer Komposition verbinden lassen würde. Wie die Zielgruppenanalyse allerdings ergeben hat, wird der Dienst fast ausschließlich seltener als wöchentlich genutzt (3.5) und weist somit eine hohe Diskrepanz im Vergleich zu den Nutzungszeiten der anderen Online-Dienste auf. Aus diesem Grund wird *Doodle* nicht für die Komposition berücksichtigt.
- Ein Dienst, der keine API anbietet, aber dennoch in die *Middleware* integriert wird, ist *Owncloud*. Dieser Dienst ist essentiell, weil er als *File-Storage*-Dienst von der HAW Hamburg im Department Informatik vorgegeben wird und von vielen Studenten genutzt wird (3.3). Eine Vorabanalyse des Webauftritts von *Owncloud* hat ergeben, dass dieser REST-Konform aufgebaut ist und dadurch ein geregelter Zugriff auf die Ressourcen des Online-Dienstes möglich ist. Obwohl *Owncloud* den technischen Teil des Kriterienkatalogs nicht erfüllt, wird er aufgrund der Umfrageergebnisse und der beschriebenen Aspekte Teil der Komposition.

Die finale Auswahl an Online-Diensten für die prototypische Realisierung ist:

- *File-Storage*: **Owncloud**, **Dropbox**, **Google Drive**
- *Versionsverwaltung*: **Github**, **Bitbucket**
- *Instant-Messaging*: **Slack**

3.1.4. Funktionale Anforderungen

Dieser Abschnitt legt die Funktionalen Anforderungen an die *Middleware* fest. Diese sind unterteilt in Anforderungen für die *Middleware* und Anforderungen an die Funktionen der in Abschnitt 3.1.3 ausgewählten Online-Dienste. Diese Funktionen beschränken sich auf die

Kernaufgaben der einzelnen Dienste, weil eine Implementation aller Funktionen der Online-Dienste den Umfang der vorliegenden Ausarbeitung übersteigt.

- *File-Storage*-Dienste: **OwnCloud, Dropbox, Google Drive**
  - **FA1** Datei in angegebenes Verzeichnis hochladen
  - **FA2** Verzeichnis- und Dateistruktur laden
  - **FA3** Datei aus angegebenem Verzeichnis herunterladen
- *Versionsverwaltungs*-Dienste: **Github, Bitbucket**
  - **FA4** *Repositories* als Liste laden
  - **FA5** Verzeichnis- und Dateistruktur innerhalb eines *Repositories* laden **GiA2**
  - **FA6** anderem Nutzer Zugriff auf *Repository* geben
- **Slack**
  - **FA7** *Channels* als Liste laden
  - **FA8** Nachrichten aus *Channel* lesen
  - **FA9** Nachricht an *Channel* senden
- ***Middleware***
  - **FA10** Zugriff auf die Funktionen der zu implementierenden Online-Dienste (3.1.3)
  - **FA11** Anbieten einer Benutzerverwaltung mit den Funktionen:
    * Registrieren eines neuen Nutzers
    * Login für einen bestehenden Nutzer
    * Speicherung von Authentikationsdaten für externe Dienste bei bestehendem Nutzer
  - **FA12** Anbieten einer Teamverwaltung mit den Funktionen:
    * ein neues Team erstellen
    * Beitritt in ein bestehendes Team
  - **FA13** Authentikationsdienst für die zu implementierenden Dienste (3.1.3)
  - **FA14** abstrakter *File-Storage*-Dienst, der folgende Funktionen unabhängig vom genutzten *File-Storage*-Dienst für Teams bereitstellt:
    * Datei in angegebenes Verzeichnis hochladen
    * Verzeichnis- und Dateistruktur laden
    * Datei aus angegebenem Verzeichnis herunterladen

3.1.5. Nichtfunktionale Anforderungen

Dieser Abschnitt beschreibt die Nichtfunktionalen Anforderungen an die *Middleware*.

- **NA1** plattformunabhängige Lösung
- **NA2** *single point of access*
- **NA3** Vereinheitlichung der Fehlersemantik
- **NA4** Vereinheitlichung von Diensten der gleichen Kategorie

3.2. Architektur

Folgender Abschnitt behandelt die Architektur für die Implementierung der *Middleware.* Dazu werden *Microservices* und monolithische Architekturen erklärt und gegenübergestellt, um ein passendes Architekturmuster zu wählen. Abschließend werden Infrage kommende Technologien vorgestellt, um darauf aufbauend den Technologiestack festzulegen.

3.2.1. Monolithische Architektur

Monolith entstammt dem griechischen Begriff *monólithos* und bedeutet „Stein aus einem Stück". Übersetzt auf die Softwaretechnik beschreibt eine monolithische Architektur ein Softwaresystem, welches seine Funktionen in einem Block vereint. Beispielsweise befinden sich das *User Interface*, der Datenbank-Anschluss und die Kommunikation in einem System. Monolithen bieten einige positive Aspekte. Zum einen können Optimierungen leichter umgesetzt werden, weil sie meistens in einer Programmiersprache entwicklet werden. Zum anderen ist die Zeit für die Beschaffung von Daten gering, weil lediglich auf Programmiersprachen-Ebene Funktionsaufrufe getätigt werden müssen. [21]

3.2.2. Microservice Architektur

Bei einer *Microservice* Architektur setzt sich das Softwaresystem aus mehreren *Microservices* zusammen. Dabei haben die *Microservices* jeweils eine eigene Aufgabe. Jeder dieser *Services* läuft in seinem eigenen Prozess oder auch verteilt auf verschiedenen Systemen. *Microservices* sollen so entkoppelt wie möglich sein. Sie sind über eine Schnittstelle zu erreichen, um Anfragen entgegenzunehmen, diese zu bearbeiten und eine Antwort zu erstellen.
Um eine hohe Entkoppelung zu erreichen, bieten sich für die Schnittstellen leichtgewichtige, zustandslose Protokolle an. Dafür eignen sich HTTP oder auch *Messaging*-Dienste wie *RabbitMQ* [22] und *ZeroMQ* [23]. *Microservice* Architekturen eignen sich vor allem für Projekte mit verschiedenen Teams. Die *Microservices* werden dann fachlich und nicht technisch nach Geschäftsprozessen gekapselt. Bei einer technischen Kapselung müssten Teams beispielsweise ständig in Rücksprache mit dem Datenbank-Team stehen, wenn sie neue Daten in die Daten-

bank einpflegen möchten. Durch die fachliche Einteilung hingegen verwaltet jeder *Microservice* seinen eigenen Kontext und somit auch die Datenbank. [24]

3.2.3. Auswahl

Um den grundlegenden Aufbau der beiden beschriebenen Architekturmuster gegenüberzustellen, sind in Abbildung 3.13 die jeweiligen Strukturen dargestellt.

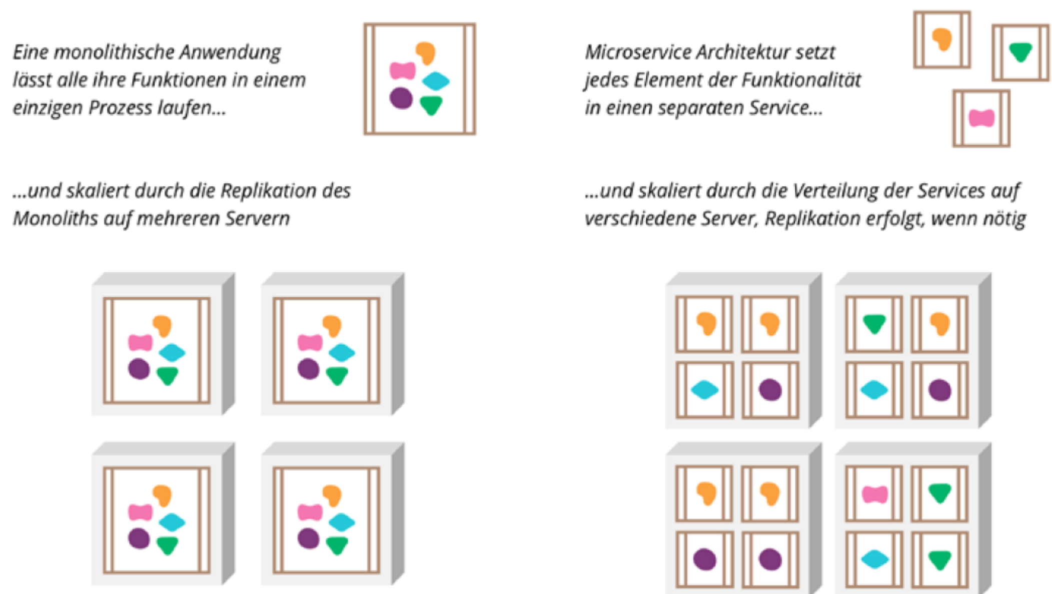


Abbildung 3.13.: Aufbau eines monolithischen Systems im Vergleich zur *Microservice* Architektur [24]

Die Funktionen des Monolithen befinden sich in einem Prozess. Für eine Skalierung wird der gesamte Prozess auf mehreren *Servern* repliziert. Der *Microservice*-Ansatz hingegen spaltet die Funktionen in eigenständige Dienste und Prozesse auf. Die Skalierung erfolgt in dem Fall durch die Verteilung der einzelnen *Services* auf mehrere *Server*. Für die Entwicklung der *Middleware* wird das ***Microservice*** Architekturmuster verwendet. Im folgenden sind sowohl Vorteile als auch Nachteile hinsichtlich des gewählten Architekturmusters und im Vergleich zu einem *Software-Monolithen* aufgelistet.

3.2.3.1. Vorteile

- *Deployment*: Weil jeder *Microservice* in seinem eigenen Prozess läuft, können sie unabhängig voneinander ausgeliefert und in Betrieb genommen werden [24, 2]. Die gewählten zu implementierenden Dienste (3.1.3) erfordern die Einbindung externer APIs, welche sich häufig verändern und die Funktion des Dienstes stören können. Bei diesem Szenario ist ein unabhängiges *Deployment* von Nutzen. Bei einer monolithischen Architektur müsste das komplette System neu in Betrieb genommen werden.
- Plattformunabhängigkeit: Dienste innerhalb der *Microservice* Architektur müssen lediglich über kompatible Schnittstellen verfügen. Die Programmiersprache und die Entwicklungsumgebung für die Implementierung des Dienstes kann jeweils frei gewählt werden [24, 1]. Durch diese Freiheit kann sehr flexibel auf verschiedene externe Online-Dienste reagiert werden, deren APIs in spezifischen Programmiersprachen angeboten werden. Diese Bibliotheken bieten oft einen einfacheren und schnelleren Umgang mit der API an.
- Skalierung: Wie in Abbildung 3.13 gezeigt, kann bei einer *Microservice* Architektur jeder einzelne *Service* unabhängig skaliert werden. Dadurch kann adaptiv auf Belastungen reagiert werden, ohne das gesamte System replizieren zu müssen [25, 60].
- Verantwortlichkeiten: Durch die Kommunikation über klar definierte Schnittstellen sind die Verantwortlichkeiten und Funktionen der jeweiligen Dienste gut ersichtlich. Bei monolithischen Systemen sind die Schnittstellen auf der Klassen-Ebene zu finden. Die meisten Programmiersprachen bieten allerdings keine Möglichkeiten für *Published Interfaces* [26], wodurch nur die Dokumentation und Disziplin des Entwicklers zusätzliche Abhängigkeiten verhindern. [24]

3.2.3.2. Nachteile

- Performance: *Microservices* rufen sich nicht wie monolithische Systeme *in-memory* auf, sondern über die Grenzen eines Prozesses hinaus. Dadurch werden Daten langsamer und mit schwankenden Latenzzeiten ausgetauscht [25, 61].
- Schnittstellen-Abhängigkeiten: Einzelne *Services* können innerhalb einer *Microservice* Architektur zwar leicht ausgetauscht werden. Sobald eine Schnittstelle verändert wird, müssen aber alle *Services* bearbeitet und neu *deployed* werden, die diese Schnittstelle implementieren [25, 80]. Aus diesem Grund ist es besonders wichtig, die Schnittstellen klar und eindeutig zu definieren, um Änderungen an der Schnittstelle gering zu halten.

Das Hauptargument für die Auswahl der *Microservice* Architektur stellt die Plattformunabhängigkeit dar. Durch die Komposition verschiedener Online-Dienste und deren externe APIs bietet das flexible Auswählen von Technologien innerhalb der *Microservice* Architektur eine gute Basis für die Eingliederung jeglicher Online-Dienste.

3.2.4. Architekturaufbau

Um den grundlegenden Aufbau des Software-Systems aufzuzeigen, ist in Abbildung 3.14 eine Übersicht der Architektur zu sehen.

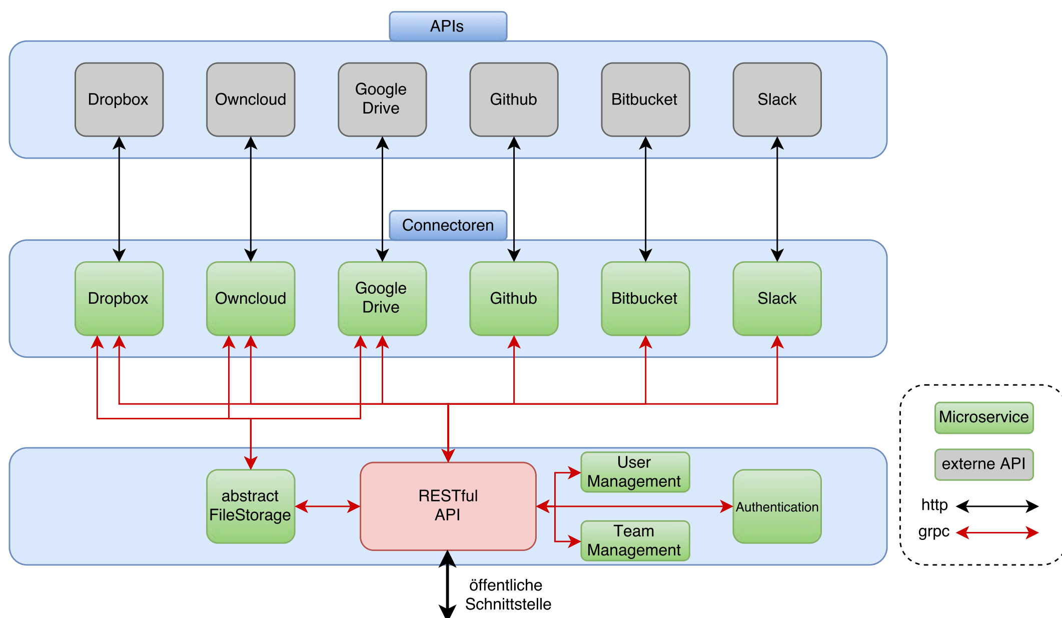


Abbildung 3.14.: Architektonischer Aufbau der *Middleware*

Die Architektur ist in drei Ebenen aufgeteilt. Die unterste Ebene bietet eine *RESTful*-API an, welche den Zugriff auf die *Middleware* bereitstellt. Außerdem beinhaltet sind sowohl *Microservices* für *User* und *Team Management* als auch ein Dienst für die Authentisierung bei externen Diensten und dem abstrakten *File-Storage*-Dienst. Die mittlere Ebene bietet *Connectoren* für jeden zu implementierenden Online-Dienst (3.1.3), die die jeweiligen Anforderungen aus Abschnitt 3.1.4 anbieten. Diese *Microservices* beziehen ihre Daten aus der obersten Ebene, bestehend aus externen APIs der jeweiligen Online-Dienste. Das Protokoll für die Kommunikation innerhalb der unteren Ebene und zu den *Microservices* ist Google Remote Procedure Call (GRPC) [27]. Die Zugriffe der *Microservices* auf die externen APIs erfolgt über Hypertext

Transfer Protocol (HTTP). Jegliche Kommunikation erfolgt nach dem *Request Response* Prinzip. Wie im Abschnitt 2.3 beschrieben, können Kompositionen sowohl reaktiv als auch proaktiv aufgebaut sein. In der Abbildung 3.14 ist zu sehen, dass die komponierten Dienste dem System bereits zur Laufzeit bekannt sind. Die Komposition ist also proaktiv aufgebaut. Grund dafür sind die Charakteristiken einer *Middleware*, die einen häufigen Zugriff erwartet und zum Beispiel bei Datentransfer lange Laufzeiten benötigt. Diese Aspekte weisen auf die Notwendigkeit einer proaktiven Komposition hin [8, 5].

3.3. Technologie

In diesem Kapitel wird der verwendete Technologie-Stack für die Implementierung der *Middleware* vorgestellt.

3.3.1. Entwicklungsplattform

3.3.1.1. Node.js

Node.js ist eine serverseitige JavaScript-Plattform. Sie basiert auf der Laufzeitumgebung "V8" von *Google*, welche in C++ geschrieben ist [28]. Die *Engine* hat die Aufgabe, den Javascript Code zu kompilieren und innerhalb einer Virtual Machine (VM) auszuführen. Das besondere an *Node.js* ist das asynchrone und Event-getriebene Design. *Node.js* fungiert mit einer *Event-Loop*, die von einem einzelnen *Thread* verwaltet wird. Diese *Event-Loop* gibt zeitintensive I/O Operationen an sogenannte *Worker Threads* ab. *Node.js* abstrahiert also die Nebenläufigkeit. Wenn eine Operation fertig bearbeitet wurde, kann der *Callback*, der an das *Event* gekoppelt wurde ausgeführt werden. In der Abbildung 3.15 wurde die Funktionsweise der *Event-Loop* nochmal dargestellt [29].

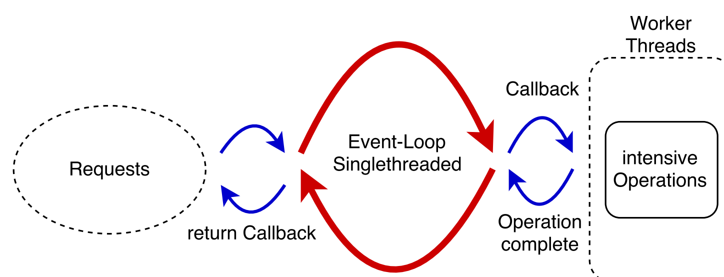


Abbildung 3.15.: Funktionsweise der *Node.js Event-Loop*

Wichtig im Zusammenhang mit *Node.js* ist der Node Package Manager (NPM) [30]. Dieser bietet eine Plattform, über die Entwickler Bibliotheken öffentlich bereitstellen können. Diese Bibliotheken können mit ihren Abhängigkeiten über den NPM installiert werden und somit in jeder *Node.js* Anwendung wiederverwendet werden [31].

Folgende Kriterien sind die Hauptargumente für *Node.js* als Entwicklungsplattform:

- Der Node Package Manager bietet eine Vielzahl an Bibliotheken, die für die Entwicklung von Web-Anwendungen genutzt werden können. Zum Beispiel für das Bearbeiten von HTTP *Requests*, für das *Logging* oder auch diverse Datenbank-Treiber.
- Eine *Middleware*, die ihre Funktionen über eine *RESTful*-API populiert, kann viele nebenläufige Anfragen erwarten. *Node.js* ist durch das asynchrone Design prädestiniert dafür. In der Arbeit „*Performance comparison and evaluation of web development technologies in php, python, and node.js*" [32] von den Autoren *Kai Lei, Yining Ma, Zhi Tan*, wurden *PHP*, *Python* und *Node.js* hinsichtlich ihrer Performance bei der Web-Entwicklung untersucht. Als Ergebnis wurde herausgestellt, dass sich *Node.js* am besten für I/O-intensive Anwendungen eignet, bei der viele nebenläufige Anfragen auftreten [32, 667].
- Das *Deployment* ist eine wichtige Anforderung in einer *Microservice* Architektur (3.2.3). Jede *Node.js* Anwendung besitzt eine *package.json* Datei, die unter anderem alle nötigen Abhängigkeiten beschreibt oder auch den Startpunkt der Anwendung festlegt. Dies macht den *Build*-Prozess reproduzierbar und erleichtert das *Deployment* [33].

Die Entwicklung mit *Node.js* eröffnet die Möglichkeit, hilfreiche *Packages*, die über den Node Package Manager veröffentlicht werden, zu nutzen. Im Folgenden sind einige *Packages* aufgelistet, die während der Entwicklung verwendet wurden:

- *Mongoose* [34] ist ein *Object Modeling Tool*, welches im Zusammenhang mit *MongoDB* verwendet wird. Es ermöglicht die Verwendung von Datenbank-Schemata und den Zugriff auf *Documents* über *Javascript*-Objekte.
- *Request* [35] ist ein vereinfachter HTTP *request client*. Durch das Parsen des eingetroffenen HTTP-Requests wird sowohl die Bearbeitung als auch die Beantwortung der Anfrage vereinfacht.
- *Nconf* [36] verwaltet hierarchische Konfigurationsdateien und ermöglicht das Laden von Umgebungsvariablen und Kommandozeilenparametern.

- *Winston* [37] ist eine universelle *logging* Bibliothek. Es können verschiedene *Log-Level* festgelegt werden und die Ausgabe kann flexibel auf Dateien umgelegt werden.
- *PM2* [38] ist ein Prozess Manager für *Node.js*-Applikationen. Gerade eine *Microservice*-Architektur erfordert ein Tool zum Starten von Applikationen, damit nicht jeder *Microservice* manuell gestartet werden muss. *PM2* bietet außerdem die Möglichkeit, Prozesse automatisch neu zu starten, wenn ein Fehler aufgetreten ist.

3.3.2. Datenbank

Im folgenden Abschnitt wird *MongoDB* als Datenbanksystem vorgestellt, die für einzelne *Microservices* innerhalb der *Middleware* verwendet werden.

3.3.2.1. MongoDB

MongoDB ist eine dokumentenorientierte *NoSQL*-Datenbank. *NoSQL* steht hierbei für ein nicht-relationales Datenbanksystem. Anders als bei relationalen Datenbanken werden bei der *MongoDB* die Daten in *Collections* zusammengefasst, nicht in Tabellen. Innerhalb der *Collections* befinden sich *Documents*, die jeweils einen Eintrag in der Datenbank darstellen. Das Format der *Documents* ist Binary JSON (BSON), eine binäre Darstellung von JSON. BSON zeichnet sich durch eine effiziente Codierung und Decodierung aus. Besondere Eigenschaften der *Documents* ist die schemalose Darstellung. Dies führt dazu, dass sich *Documents* mit unterschiedlichen Eigenschaften in einer *Collection* befinden können, was eine dynamische Erweiterung der Eigenschaften von *Documents* ermöglicht [39].

Im folgenden sind die Kernaspekte aufgelistet, warum sich *MongoDB* als dokumentenorientierte Datenbank besser für die *Middleware* eignet, als zum Beispiel eine relationale Datenbank wie *MySQL*:

- In der Arbeit „*A comparative study: MongoDB vs. MySQL*“ [40] von den Autoren *Cornelia Györödi, Robert Györödi, George Pecherle, Andrada Olah*, wurde die *Performance* hinsichtlich der vier Kernfunktionen *Insert, Select(query), Update, Delete* der Datenbanksysteme *MongoDB* und *MySQL* verglichen. Die Daten in dem vorgestellten Testszenario wurden strukturell aufgebaut. *MongoDB* hatte in allen Funktionen eine geringere Ausführungszeit und bot somit eine bessere *Performance*.
- Ein weiteres Argument ist die native Unterstützung von JSON-Dateien durch *Node.js*. Weil jedes *Document* innerhalb der *MongoDB* eine JSON-Datei darstellt, wird der Umgang mit den einzelnen *Documents* durch *Node.js* erleichtert.

3.3.3. Kommunikation

In diesem Abschnitt werden die Kommunikationsprotokolle und Datenformate für die *Microservices* und die *RESTful*-API vorgestellt.

3.3.3.1. GRPC

GRPC ist ein *Cross Plattform*, *Open Source*, Remote Procedure Call (RPC) *Framework* [27]. Aktuell werden zehn Programmiersprachen unterstützt, wie zum Beispiel C++, Java, Python oder Node.js. *GRPC* nutzt für die Datenübertragung Hypertext Transfer Protocol Version 2 (HTTP/2) [41]. Einige wichtige *Features* von GRPC sind *Load Balancing*, *Authentication* und Abwärtskompatibilität. Um einen *Remote Procedure Call* mit GRPC aufrufen zu können, wird zunächst ein *Service* definiert, der die verschiedenen Funktionen mit dessen *request* und *response* Objekten beschreibt. Die *Services* und Objekte werden mithilfe von *Protobuf* [42] beschrieben, ein Sprach- und Plattform-neutraler Mechanismus zum Serialisieren von Daten. Im Gegensatz zu JSON und XML nutzt *Protobuf* ein Binär-Format. Die Datenstrukturen werden in .proto Dateien geschrieben und anschließend kompiliert. Das Ergebnis der Kompilierung sind Klassen, auf denen Schreib- und Lesevorgänge getätigt werden können [43, 2]. In 3.1 ist ein beispielhafter GRPC *Service* zu sehen.

```
1  service Person {
2    rpc GetAgeOfPerson (AgeRequest) returns (AgeResponse) {}
3  }
4  message AgeRequest {
5    string name = 1;
6  }
7  message GetFileRequest {
8    string name = 1;
9    int32 age = 2;
10 }
```

Listing 3.1: Beispieldatei eines GRPC Services

Dieser *Service* stellt die Funktion **GetAgeOfPerson** bereit, die das Alter einer Person anhand des Namens zurückgibt. Als Parameter werden die in Zeile 6 und 9 beschriebenen *Request* und *Response* Objekte verwendet. Die Werte neben den Variablen dienen dem Serialisieren, es entstehen *Key-Value* Paare zwischen dem eigentlichem Wert der Variable und dem Wert aus der .proto-Datei.

GRPC wird für die Kommunikation zwischen den *Microservices* genutzt. In Betracht gezogen

wurden außerdem *RabbitMQ* oder auch HTTP. Eine wichtige Anforderung an das Kommunikationsprotokoll ist das *Interfacing*. GRPC bietet für zwei Kommunikationspartner klar definierte Funktionen und Parameter mit vorgegeben Datentypen. Dadurch ist die angegebene Schnittstelle klar formuliert und Kommunikationspartner wissen, wie und auf welche Funktionen zugegriffen werden kann. Über *RabbitMQ* oder HTTP hingegen kann nur die Übertragung realisiert werden, ein *Interfacing* müsste über eine *Interface Definition Language* bereitgestellt werden.

3.3.3.2. REST

REST bedeutet *Representational State Transfer* und ist ein Architekturparadigma, das beschreibt, wie auf Ressourcen im Web zugegriffen werden soll. Es gilt als Standard für das *Service* Design im Web [44, 2]. REST beschreibt den Zugriff auf Datenstrukturen und die Übertragung von deren Zuständen. Dabei ist jeder Zugriff zustandslos, liefert also alle Parameter mit, die der Server braucht, um die Anfrage vollständig zu bearbeiten. In 3.1 ist der Aufbau einer URL beschrieben, die eine Ressource bei REST beschreibt.

```
Protokoll://Host/ApplikationsPfad/RessourcenTyp/RessourcenID
```

Listing 3.2: Identifikation von Ressourcen in RESTful Architekturen

Auf diese Ressource kann mit verschiedenen Operationen zugegriffen werden. Diese Operationen werden vereinfacht als CRUD (*Create, Read, Update, Delete)* bezeichnet. In Verbindung mit HTTP als Protokoll werden die Funktionen *POST, GET, PUT, DELETE* auf die CRUD Operationen gemapped. Ein *Service*, der REST als Architekturparadigma implementiert, wird *RESTful Service* genannt. Innerhalb der *Middleware* wird Representational State Transfer (REST) für den öffentlichen Zugriff genutzt. Die Funktionen der *Middleware* werden über eine *RESTful*-API angeboten.

3.3.3.3. JSON und Protobuf

JSON oder ausgeschrieben *JavaScript Object Notation* ist ein leichtgewichtiges, textbasiertes und sprachenunabhängiges Format zum Austausch von Daten [45]. Die Designvorgaben an JSON sind minimal, *portable*, textbasiert und Kompatibilität zu *Javascript* [45]. JSON wird also nativ von Javascript unterstützt, sodass Entwickler keine zusätzlichen Bibliotheken einbinden müssen. In 3.3 ist eine beispielhafte JSON-Struktur zu sehen.

```
{
"Name":"Max",
"Alter":20,
```

```
4 "istStudent":true
5 }
```

Listing 3.3: Beispiel JSON Struktur

Anders als zum Beispiel XML [46] kann JSON verschiedene Objekttypen repräsentieren. Und zwar *String*, *Number*, *Boolean* und *Null* [47].

Protobuf ist der Mechanismus, wie in 3.3.3.1 beschrieben, der bei GRPC zur Serialisierung verwendet wird. Im Rahmen der *Middleware* wird *Protobuf* für die interne *Microservice*-Kommunikation verwendet, und JavaScript Object Notation (JSON) für die öffentliche Schnittstelle. Der Hauptgrund für die Auswahl ist die *Performance*. Eine API sollte Anfragen möglichst schnell bearbeiten, um lange Wartezeiten beim Zugriff zu vermeiden. Eine *Microservice* Architektur benötigt für das Abarbeiten einer Anfrage möglicherweise mehrere verteilte Aufrufe, deswegen sollte die Kommunikation einzelner verteilter Aufrufe möglichst performant sein. *Protobuf* nutzt ein binäres *Encoding* und ist somit hinsichtlich der Datenmenge kleiner als JSON oder XML. In der Arbeit „*Impacts of data interchange formats on energy consumption and performance in smartphones*“ [48] von den Autoren *Bruno Gil, Paulo Trezentos*, wurden *Protobuf*, JSON und XML hinsichtlich ihrer *Performance* untersucht. In dem vorgestellten Szenario ist zu sehen, dass die Datenmenge von *Protobuf* 38% geringer als die Datenmenge von JSON und 48% geringer als die Datenmenge von XML ist [48, 2]. *Protobuf* wird nicht für die öffentliche Schnittstelle genutzt, weil diese für möglichst viele Nutzer leicht zugänglich sein soll, was durch ein allgemein verbreiteteres Format wie JSON oder XML besser erreicht wird. JSON wird XML vorgezogen, weil zum einen die Datenmenge geringer ist, und zum anderen Javascript, die Programmiersprache von *Node.js*, eine native Unterstützung für JSON bietet.

4. Besondere Herausforderungen bei der Implementation

In diesem Abschnitt werden besondere Herausforderungen beschrieben, welche bei der Entwicklung der *Middleware* aufgetreten sind.

4.1. Asynchrone Programmierung

Wie in Abschnitt 3.3.1.1 beschrieben, nutzt *Node.js* ein asynchrones und Event-getriebenes Design. Dies stellt verschiedene Anforderungen an die Entwickler. Bei klassischen Programmiersprachen, wie zum Beispiel *JAVA*, gehört das sequenzielle Ausführen vom Programmcode zur Norm. Bei der Entwicklung mit *Node.js* werden allerdings rechenintensive I/O-Operationen asynchron ausgeführt, wodurch Sprünge bei der Ausführung des Programmcodes auftreten. Um dieses Verhalten aufzuzeigen, ist in 4.1 und 4.2 ein Vergleich zwischen *Java* und *Node.js* anhand eines Beispiels dargestellt.

```
User user = null;
user = database.getUser("Mustermann");
System.out.println(user.name); //Max Mustermann
```

Listing 4.1: Pseudocode Java

```
var user;
database.getUser('Mustermann',function (user) {
        user = user;
});
console.log(user.name); //undefined
```

Listing 4.2: Pseudocode Node.js

Beide Code-Beispiele haben die Aufgabe, den Benutzer mit dem Namen „Mustermann“ aus der Datenbank zu lesen und auszugeben. Der *Java*-Pseudocode läuft sequentiell ab. Zu erst wird eine User-Variable deklariert (Zeile 1), dann ein blockierter Aufruf an die Datenbank gestellt (Zeile 2), um den gesuchten Benutzer zurückzugeben, und zum Schluss wird erfolgreich

der Name des Nutzers ausgegeben (Zeile 3). Der Ablauf beim *Node.js*-Pseudocode ist ähnlich. Zuerst wird die User-Variable deklariert (Zeile 1), dann der Datenbankaufruf getätigt, um den gesuchten Benutzer auf die Variable zu schreiben (Zeile 2-4), und in Zeile 5 wiederum der Name des Nutzers ausgegeben. Wie in den Kommentaren schon zu sehen ist, wird allerdings anstatt des richtigen Namens *undefined* ausgegeben. Dies liegt daran, dass zu dem Zeitpunkt, bei dem die Programmzeile 5 ausgeführt wird, die Variable „*user*" noch nicht mit einem Wert belegt ist. Die Ursache dafür ist das Design von *Node.js*, dass I/O-Operationen asynchron ausführt. Dies führt dazu, dass in Zeile 2 nicht auf das Ergebnis der Datenbankanfrage gewartet wird, sondern die I/O-Operation an die Event-Queue abgegeben und zu einem späteren Zeitpunkt berechnet wird. Um den Namen erfolgreich ausgeben zu können, müsste die Ausgabe des Namens innerhalb der *Callback*-Funktion zwischen Zeile 2 und 4 aufgerufen werden.

Eine weitere Herausforderung bei der asynchronen Programmierung mit *Node.js* ist das serielle Abarbeiten von I/O-Operationen. Wie im vorigen Beispiel gezeigt, wird nicht auf das Ergebnis einer asynchronen Operation gewartet. Um nun mehrere asynchrone Funktionen nacheinander auszuführen, können die jeweiligen *Callbacks* miteinander verschachtelt werden. Der erste *Callback* startet die zweite I/O-Operation, der zweite *Callback* die dritte Operation. Dieser Ablauf wird abhängig von der Anzahl der durchzuführenden Operationen wiederholt. In 4.3 ist ein Code-Beispiel zu sehen, bei dem mehrere Aufrufe nacheinander abgearbeitet werden.

```
1 server.on('request', function(req, res) {
2         db.getSession(req, function(session) {
3                 db.getUserData(session, function(userData) {
4                         page = pageRender(req, session, userData);
5                         res.write(page);
6                 });
7         });
8 });
```

Listing 4.3: Pseudocode für das serielle Ausführen mehrerer I/O-Operationen in *Node.js*. (inspiriert von [49, 43])

Bei diesem fiktiven Beispiel wird eine Anfrage an einen *Webserver* gestellt. Um diese schlussendlich beantworten zu können, werden mehrere Datenbankanfragen benötigt. Zum einen für die zugehörige *Session* und zum anderen für die Benutzerdaten.

Das Verschachteln mehrerer *Callbacks* führt zwar zu dem gewünschten Ergebnis, mehrere I/O-Operationen seriell ausführen zu können, verringert allerdings die Lesbarkeit des *Codes*, und führt dadurch zu einer schlechteren Wartbarkeit. Dieses Problem wird auch als *Pyramid of*

Doom bezeichnet [29, 47], weil eine zur Seite gerichtete *Code*-Pyramide entsteht. Um eine solche *Code*-Pyramide etwas lesbarer zu gestalten, können zum Beispiel die *Callback*-Funktionen benannt werden (4.4).

```
server.on('request', function getSessionFromDb(req, res) {
        db.getSession(req, function getUserDataFromDb(session) {
                db.getUserData(session, function render(userData) {
                        page = pageRender(req, session, userData);
                        res.write(page);
                });
        });
});
```

Listing 4.4: Serielle Ausführung von I/O-Operationen mit benannten Callback-Funktionen. (inspiriert von [49, 44])

Mit diesem Vorgehen werden die einzelnen Aufgaben der Funktionen besser beschrieben und sorgen für eine bessere Verständlichkeit. Um kontinuierlich nach rechts wachsende Einrückungen zu verhindern und die *Code*-Pyramide zu umgehen, können sogenannte *Promises* verwendet werden. Ein *Promise* stellt ein Objekt dar, welches als Platzhalter für das Ergebnis einer Funktion dient, die zu einem späteren Zeitpunkt ausgeführt wird [50]. Ein *Promise* kann sich in einem von drei Zuständen befinden [51]:

- *pending*: Das *Promise* befindet sich im initialen Zustand. Es wurde weder erfolgreich abgeschlossen, noch ist es gescheitert.
- *fulfilled*: Die Operation wurde erfolgreich abgeschlossen.
- *rejected*: Die Operation ist gescheitert.

Im Code-Beispiel 4.5 ist das vorige Szenario mithilfe eines *Promise* gelöst.

```
server.on('request', function getSessionFromDb(req,res) {
        var sessionInfo;
        db.getSession(req)
                .then(session => {
                        sessionInfo = session;
                        return db.getUserData(session);
                })
                .then(userData => {
                        page = pageRender(req,sessionInfo,userData);
                        res.write(page);
```

```
11                    })
12                    .catch(error => {
13                            handleError(error);
14                    });
15 });
```

Listing 4.5: Serielle Ausführung von I/O-Operationen mit Hilfe eines *Promise*

Es ist eine deutlich besser geordnete Struktur zu erkennen, wodurch eine mit der Anzahl der seriell auszuführenden Funktionen wachsende *Code*-Pyramide verhindert wird. Um diese Art von *Code*-Struktur aufbauen zu können, müssen die Funktionen *getSession()* und *getUserData()* ein *Promise* zurückgeben. Wenn die Funktionen ihr *Promise* erfüllen (*fulfilled*-Zustand), die Operation also ohne Fehler erfolgreich abgeschlossen wird, kann ihr Ergebnis an den nächsten *then*-Zweig weitergegeben werden. Wird das *Promise* jedoch nicht erfüllt und die Operation ist fehlgeschlagen (*rejected*-Zustand), wird der *Catch*-Zweig aufgerufen und Fehler können, ohne den *Code* duplizieren zu müssen, an einer Stelle behandelt werden.

4.2. Implementation der *File-Storage*-Dienste

4.2.1. Vereinheitlichung

Owncloud, Dropbox und *Google Drive* wurden als *File-Storage*-Dienste für die Dienstkomposition ausgewählt (3.1.3). Die drei Dienste sollen das Laden der Verzeichnis- und Dateistruktur ermöglichen und das Hoch- und Herunterladen von Dateien bereitstellen (3.1.4). Um eine einfache Anwendung der *Middleware* zu ermöglichen und die Dienste untereinander austauschbar zu gestalten, sollen die Zugriffe auf die einzelnen Funktionen der *File-Storage*-Dienste vereinheitlicht werden. Die Eingabe- und Ausgabeparameter sollen für alle drei Dienste identisch sein.

Im Abschnitt 3.14 wurde die Architektur der *Middleware* beschrieben. Die Anforderung der Vereinheitlichung hat mehrere Auswirkungen auf das Software-System. Der Zugriff auf die *Middleware* für den Anwender über die *RESTful*-API unterscheidet sich für die drei *File-Storage*-Dienste nur im Uniform Resource Locator. Sowohl die Parameter des HTTP-*Requests* als auch das Ergebnis der Anfrage sollen die gleichen Signaturen aufweisen. Auch die Schnittstellen der *File-Storage-Microservices*, die mit den APIs der Online-Dienste kommunizieren, sollen identisch sein und sich nur anhand der Zugriffsadresse unterscheiden.

Obwohl *Owncloud, Dropbox* und *Google Drive* als *File-Storage*-Dienste identische Anwendungsszenarien anbieten, sind einige Unterschiede in der internen Funktionsweise zu berücksichtigen.

Ein erster Unterschied ist im Auslesen der Verzeichnis- und Dateistruktur aus den APIs der Online-Dienste zu erkennen. Während *Owncloud* und *Dropbox* die Dateien und Ordner innerhalb eines Verzeichnisses anhand des Namens identifizieren, ordnet *Google Drive* jeder Datei und jedem Ordner einen *Identifier* zu. Für die Benutzung der angebotenen APIs der drei Dienste bedeutet dies, dass Aktionen auf die Ressourcen von *Google Drive* einen *Identifier* benötigen, wohingegen *Owncloud* und *Dropbox* ein Tupel aus Ressourcenpfad und Ressourcenname für den Zugriff nutzen. Weil zwei der drei *File-Storage*-Dienste den gleichen Ansatz verfolgen, wurden die Schnittstellen der *Microservies* und der *RESTful*-API dahingehend ausgerichtet, eine Ressource durch den Pfad und den Namen zu identifizieren. Dies bedeutet für den *Google Drive Connector* (3.2.4), dass ein *Workaround* nötig ist, um die Identifikation der Pfade und Dateien an die anderen beiden *File-Storage*-Dienste anzupassen. In 4.1 ist zu sehen, welche *Requests* an die API von *Google Drive* nötig sind, um Dateien anhand des Pfads und des Namens herunterzuladen.

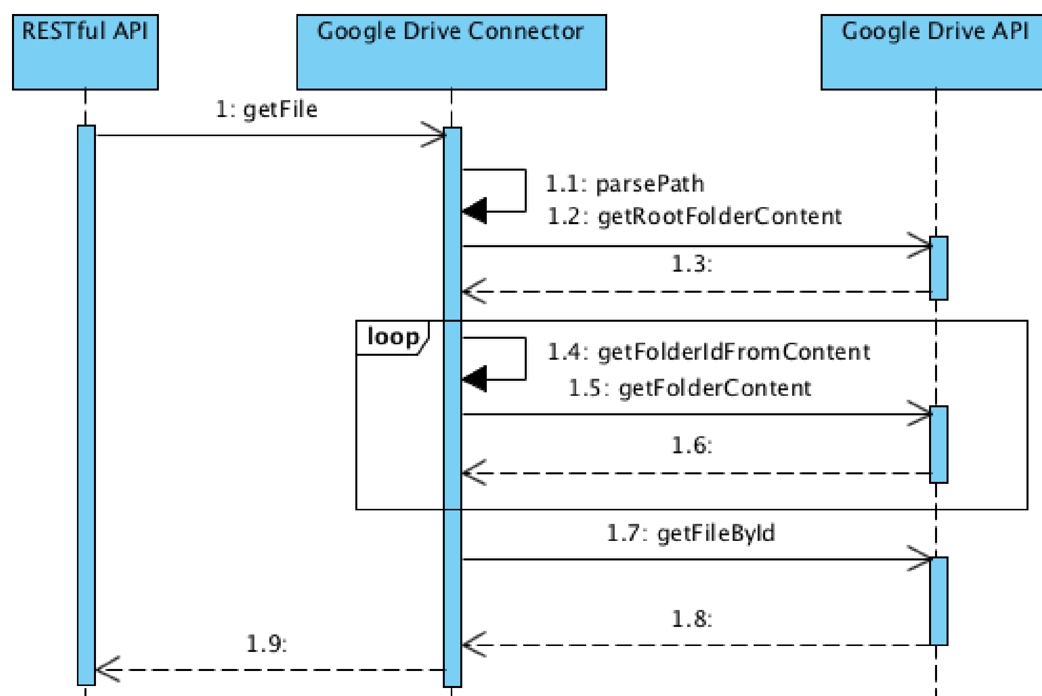


Abbildung 4.1.: Sequenzdiagramm eines Dateidownloads beginnend bei der *RESTful*-API, über den *Google Drive Connector*, bis hin zur externen API von *Google Drive*

Vorab ist zu erkennen, dass alle Schritte außer 1.1 und 1.4 asynchron ablaufen. Darin spiegelt sich *Node.js* als Entwicklungsplattform wider. Im Diagramm wurden exemplarisch drei Aktoren deklariert. Die eigentliche Anfrage wird an die *RESTful*-API gestellt, die diese wiederum an den *Google Drive Connector* weitergibt. Dort wird zunächst in Schritt 1.1 der Pfad der herunterzuladenden Datei geparsed, um die einzelnen Ordner herauszulesen. Draufhin werden in Schritt 1.2 die Metadaten des *Root*-Ordners gelesen. Der *Root*-Ordner ist der einzige Ordner innerhalb von *Google Drive*, der anhand seines Namens „Root“ ausgelesen werden kann [52] und keinen *Identifier* benötigt. Nun wird für jeden im Pfad angegeben Unterordner der entsprechende Ordner-*Identifier* aus den Metadaten des *Root*-Ordners beziehungsweise des aktuellen Unterordners gelesen (Schritt 1.4), um über den Ordner-*Identifier* erneut die Metadaten des nächsten Ordners zu lesen (Schritt 1.5). Sobald der Zielordner erreicht ist, wird der *Identifier* der zu ladenden Datei aus den Metadaten des Zielordners gelesen und schlussendlich heruntergeladen (Schritt 1.7).

Ein weiterer Unterschied in der internen Funktionsweise von den *File-Storage*-Diensten ist beim Hochladen von Dateien zu erkennen. *Owncloud* und *Google Drive* antworten mit einem Fehler, wenn eine Datei in einen noch nicht erstellten Ordner geladen werden soll. *Dropbox* allerdings erstellt die fehlenden Ordner und lädt die Datei in das Zielverzeichnis. Neue Ordner müssen nur dann erstellt werden, wenn eine Datei in einen nicht vorhandenen Ordner hochgeladen werden soll. Die Aktion ist also an die *Upload*-Funktion gebunden. Aus diesem Grund bietet die *RESTful*-API der *Middleware* keine zusätzliche Funktion für das Erstellen von Verzeichnissen an. Stattdessen werden die *Microservices* für *Owncloud* und *Google Drive* dahingehend angepasst, dass innerhalb der Dateiupload-Funktion mögliche fehlende Ordner erstellt werden. Dazu kontrolliert der jeweilige *Connector* des *File-Storage*-Dienstes bei jeder Dateiupload-Anfrage, ob alle benötigten Ordner vorhanden sind und erstellt mögliche fehlende Ordner. Dies wird mit einer Anfrage über die API des Online-Dienstes umgesetzt.

4.2.2. Datei-Austausch

Eine weitere Herausforderung besteht im Austausch von Dateien in Verbindung mit den *File-Storage*-Diensten. Durch die *Microservice*-Architektur müssen die Dateien über mehrere Prozesse transferiert werden. Wie in Abschnitt 3.3.3.1 beschrieben, wird GRPC als Kommunikationsprotokoll unter den einzelnen *Microservices* verwendet. Allerdings besitzt GRPC eine maximale Größe für einzelne Nachrichten von 4 Megabyte [53]. Aus diesem Grund können Dateien nicht im Ganzen transferiert werden. Um das Problem zu lösen, müssen die Dateien in mehrere *Chunks* aufgeteilt und einzeln verschickt werden. GRPC bietet für dieses Anwen-

dungsszenario das *Streamen* von Nachrichten an. Dabei werden die einzelnen *Chunks* in den *Stream* geschrieben und auch beim Empfänger als einzelne *Chunks* gelesen. Durch das Beenden des *Streams* wird signalisiert, dass alle Daten gesendet wurden. Das *Streamen* von Dateien hat mehrere positive Auswirkungen auf das gesamte Software-System. Zum einen müssen keine größeren Datenmengen im Arbeitsspeicher gehalten werden. Zum anderen steigt die Geschwindigkeit beim Hoch- und Herunterladen, weil nicht auf das Laden der gesamten Datei gewartet werden muss, sondern bereits empfangene Teile der Datei weitergereicht werden können. Die Implementierung der *File-Storage-Connectoren* nutzt das *Request-Package* (3.3.1.1), ein HTTP-Client, der sowohl das Senden als auch das Empfangen von HTTP-Anfragen als *Stream* ermöglicht [35]. Dadurch können die einzelnen *Chunks* direkt an den GRPC-*Stream* weitergegeben werden, wodurch die angegebenen Vorteile, geringere Speichernutzung und bessere *Performance*, ermöglicht werden. In Abbildung 4.2 ist das *Streamen* der Datei noch einmal vereinfacht dargestellt. Im Gegensatz zur Abbildung 4.1, wird nur der direkte Dateiaustausch berücksichtigt.

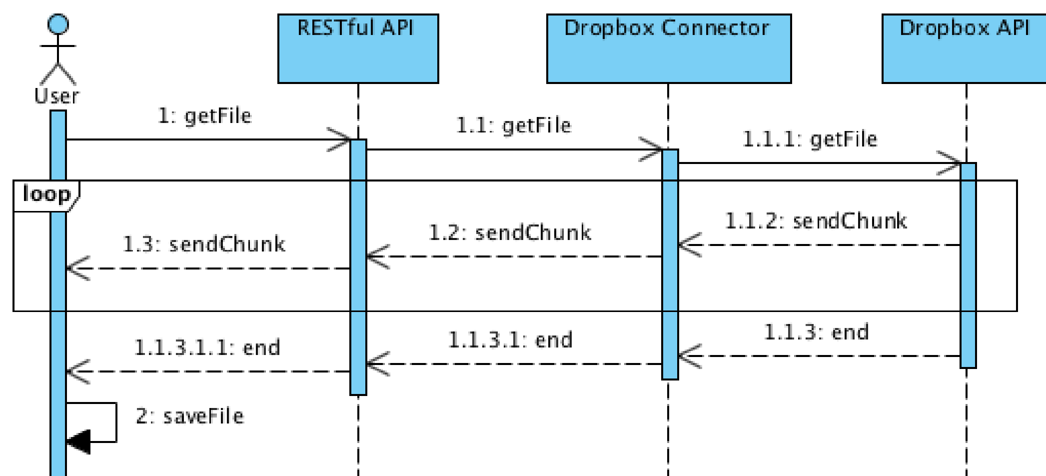


Abbildung 4.2.: Sequenzdiagramm über das Streamen einer Datei innerhalb der *Middleware*

4.3. Abstrakter *File-Storage*-Dienst

Es wurde als Anforderung formuliert, einen abstrakten *File-Storage*-Dienst in die *Middleware* zu integrieren. Dieser soll die bereits implementierten *File-Storage*-Dienste nutzen, um daraus einen neuen Dienst zu komponieren. Ziel des neuen Dienstes ist es, einen Dateispeicher für

Teams anzubieten, der sich aus den bereits vorhandenen *File-Storage*-Diensten zusammensetzt. In 4.3 ist die Idee des abstrakten *File-Storage*-Dienstes skizziert.

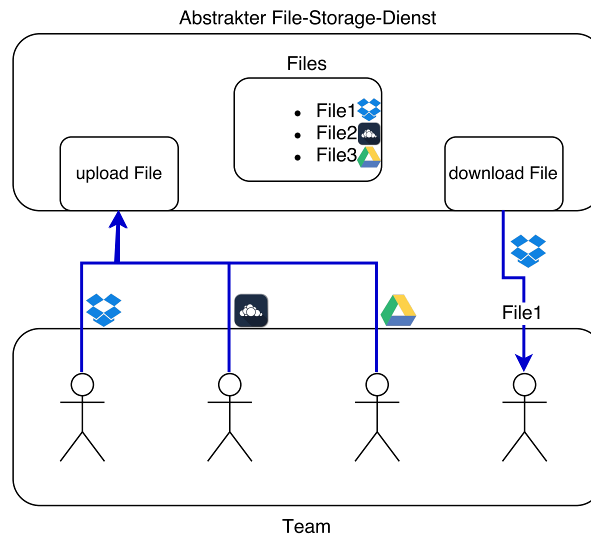


Abbildung 4.3.: Übersicht der Idee des abstrakten *File-Storage*-Dienstes

Dateien können über die *Dropbox*, *Owncloud* oder *Google Drive* hochgeladen werden. Der *Download* erfolgt wiederum über den *File-Storage*-Dienst, über den die Datei hochgeladen wurde.

Für die Entwicklung dieses Dienstes wurden zwei verschiedene Umsetzungsszenarien evaluiert. Im ersten Szenario wird für jedes Teammitglied innerhalb seines gewählten *File-Storage*-Dienstes, also *Owncloud, Dropbox* oder *Google Drive*, ein Teamordner angelegt. Wird eine Datei von einem Teammitglied hochgeladen, wird diese mit allen Mitgliedern im Team geteilt, indem die Datei in alle Teamordner geladen wird. Der Download der Datei ist trivial. Es müsste lediglich ein *Request* an den entsprechenden *File-Storage*-Dienst gestellt werden, bei dem der Pfad auf die Datei innerhalb des Teamordners zeigt.

Im zweiten Szenario wird ebenfalls für jedes Teammitglied ein Teamordner erstellt. Wird eine Datei über einen der *File-Storage*-Dienste hochgeladen, werden die relevanten Parameter der Anfrage gespeichert. Dies betrifft sowohl Name und Pfad der Datei als auch der Name des Benutzers, der die Anfrage autorisiert. Wenn nun eine Datei heruntergeladen werden soll, werden die gespeicherten Parameter genutzt, um den *Download* zu starten.

Um eines der beiden Szenarios auszuwählen, wurden zwei wichtige Kriterien genauer betrachtet: Zugriffskontrolle und Serverlast. Im ersten Szenario befinden sich die Dateien in dem gewählten Dateispeicher aller Teammitglieder. Jeder Nutzer hat demnach Kontrolle über die Dateien, wohingegen im zweiten Szenario die Dateien im Speicher des Nutzers abgelegt sind, der die Datei hochgeladen hat. Das Löschen einer Datei würde den Zugriff auf ebendiese für alle Mitglieder des Teams verhindern. Die Serverlast hingegen ist im ersten Szenario deutlich höher. Die Anzahl der *Uploads* steigt mit der Anzahl der Teammitglieder. Demgegenüber muss im zweiten Szenario lediglich ein *Upload* ausgeführt werden. Ein weiteres Problem des ersten Szenarios tritt auf, wenn ein neues Teammitglied beitritt. In dem Fall müssen alle Dateien, die sich im Teamordner befinden, in einen *File-Storage*-Dienst des neuen Mitglieds hochgeladen werden. In Abwägung der genannten Vor- und Nachteile beider Szenarien wird für die Implementierung das zweite Szenario ausgewählt. Für diese Entscheidung sind insbesondere die geringere Serverlast sowie die Gewährleistung einer datenseitigen Effizienz im Umgang mit wechselnden Teammitgliedern ausschlaggebend.

Für die Implementierung des abstrakten *File-Storage*-Dienstes wurde ein weiterer *Microservice* entwickelt. Dieser bietet die Funktionen Datei-*Upload*, Datei-*Download* und das Laden von Verzeichnisstrukturen an. Um die Parameter eines *Uploads* zu speichern und diese zu einem späteren Zeitpunkt für einen *Download* zu nutzen, wird eine *MongoDB Collection* genutzt. Wie in Abschnitt 3.3.1.1 beschrieben, wird das *Node-Package* „*Mongoose*“ genutzt, um Dokumente mit einem Schema in die *MongoDB* zu schreiben. In 4.6 sind die genutzten Schemata zu sehen.

```
var FileStorageSchema = new mongoose.Schema({
    virtualFilePath: {type: String, required: true},
    serviceFilePath: {type: String, required: true},
    fileName: {type: String, required: true},
    userName: {type: String, required: true},
    serviceName: {type: String, required: true}
});
var TeamStorageSchema = new mongoose.Schema({
    teamName: {type: String, required: true},
    files:[FileStorageSchema]
});
```

Listing 4.6: Schemata für den abstrakten *File-Storage*-Dienst

Weil jedes Team einen Teamordner besitzt und Dateien diesem zugeordnet werden, wurden zwei Schemata erstellt, um die Dateien logisch mit den Teams zu verbinden. Im *TeamStorage-*

Schema, innerhalb des *files*-Arrays, liegen alle Einträge für Dateien des jeweiligen Teams. Die einzelnen Felder des *File-Storage*-Schemas haben folgende Bedeutung:

- virtualFilePath: Der Pfad der Datei innerhalb des abstrakten *File-Storage*-Dienstes. Fragt ein Nutzer die Verzeichnisstruktur an, wird das Ergebnis aus diesen Einträgen zusammengesetzt.
- serviceFilePath: Beschreibt den Pfad zu dem *File-Storage*-Dienst, in dem die Datei hochgeladen wurde. Wird für die Anfrage eines *Downloads* wiederverwendet.
- fileName: Name der Datei.
- userName: Name des Benutzers, der die Datei hochgeladen hat. Bei einem *Download* der Datei wird der Benutzername dafür genutzt, den Vorgang zu autorisieren.
- serviceName: Name des *File-Storage*-Dienstes, in dem die Datei abgelegt ist.

4.4. Authentifizierung und Autorisierung

Die *Middleware* implementiert insgesamt sechs Online-Dienste. Jeder dieser Dienste bietet eine API für Anwendungsentwickler an, um auf die Daten von verschiedenen Benutzern zugreifen zu können. Die Dienste *Google Drive, Dropbox, Github, Bitbucket* und *Slack* nutzen dafür das Open Authentication 2 (OAUTH2) Protokoll. *Owncloud* hingegen bietet eine sogenannte *Basic Authentication* an. Im Folgenden werden zunächst die beiden Authentifizierungs-Methoden beschrieben und danach der Dienst für die Authentifizierung innerhalb der *Middleware* vorgestellt.

4.4.1. OAUTH2

Viele Online-Dienste bieten APIs an, um Entwicklern eine Möglichkeit zu bieten, Applikationen für den Dienst zu schreiben. Problematisch ist hierbei zu bewerten, dass Benutzernamen und Passwörter direkt an den Entwickler weitergegeben werden, um Zugriff auf die Daten der Benutzer zu ermöglichen. Eine Authentifizierung über Benutzername und Passwort führt unter anderem zu folgenden Problemen [54, 3]:

- Mögliche Benutzer der Applikation könnten davon abgeschreckt sein, ihr Passwort weiterzugeben und benutzen den Dienst somit nicht.

- Der Zugriff findet uneingeschränkt statt. Die Applikation erhält nicht nur Zugriff auf Informationen, die benötigt werden, sondern zu allen Daten des Benutzers. Damit steigt auch die Verantwortung des Entwicklers, die Passwörter der Nutzer sicher zu speichern.
- Probleme mit der Zuverlässigkeit entstehen. Wenn ein Benutzer sein Passwort ändert, verliert die Applikation den Zugriff auf die Daten.
- Der Widerruf des Zugriffs wird nur durch das Ändern des Passworts ermöglicht. Infolgedessen wird aber auch der Zugriff aller anderen Applikationen widerrufen, die mit der API verbunden sind.

Um solche Probleme zu lösen, ist das OAUTH2-Protokoll entstanden. OAUTH2 bietet eine Autorisierung basierend auf Tokens an, mit der Zugriffe auf einzelne Ressourcen ermöglicht werden, ohne direkten Zugriff auf die Login-Daten eines Benutzers zu benötigen. In dem Protokoll werden vier verschiedene Rollen spezifiziert [55, 5]:

- **Resource Owner**: Beschreibt eine Entität, die Zugriff auf eine geschützte Ressource gewähren kann. Oftmals wird hiermit auch der *End-User* bezeichnet, der einem Dienst Zugriff auf seine Daten gewährt.
- **Resource Server**: Ein Server, der die geschützten Ressourcen bereitstellt, und Zugriff auf diese gewähren kann, wenn eine gültige Anfrage mithilfe von *access tokens* gestellt wurde.
- **Client**: Eine Applikation, die Anfragen an geschützte Ressourcen stellt, stellvertretend für den *Resource Owner*. Im Rahmen der vorliegenden Ausarbeitung ist die entwickelte *Middleware* ein Beispiel für einen solchen *client*.
- **Authorization Server**: Der Server, der *Access Tokens* an den *Client* ausliefert, nachdem der *Resource Owner* authentifiziert und eine gültige Autorisierung erhalten wurde. *Authorization Server* und *Resource Server* können identisch sein. Dies ist nicht genauer spezifiziert.

In der Abbildung 4.4 ist der abstrakte Ablauf des OAUTH2-Protokolls zu sehen.

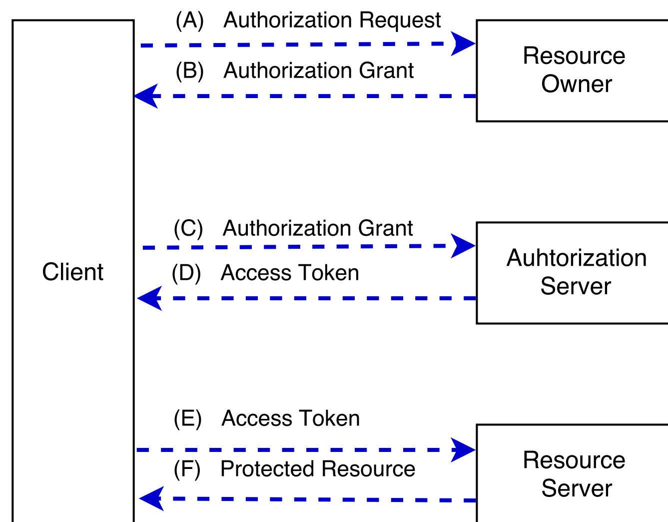


Abbildung 4.4.: Abstrakter Ablauf des *OAUTH2*-Protokolls [55, 10]

In der Abbildung finden sich die zuvor beschriebenen vier Rollen mitsamt ihrer Wechselwirkungen untereinander wieder. Der Vollständigkeit halber muss noch hervorgehoben werden, dass die Anfrage A nicht direkt an den *Resource Owner*, sondern idealerweise auch indirekt über den *Authorization Server* als Verbindungsstelle gestellt werden kann. In Schritt D wird dem *Client* ein *Access Token* übermittelt. Dieses Token ermächtigt den *Client*, vom *Resource Owner* autorisierte Anfragen an den *Resource Server* zu stellen. Neben dem *Access Token* spezifiziert das *OAUTH2* -Protokoll das *Refresh Token*. Mit diesem können abgelaufene *Access Tokens* erneuert werden, ohne den gesamten Protokoll-Ablauf wiederholen zu müssen [55, 10].

4.4.2. Basic Authentication

Im Gegensatz zu *OAUTH2* dient die *Basic Authentication* lediglich dazu, Benutzer zu authentifizieren, ohne ergänzende individuelle Autorisierungen zu ermöglichen. Einem Server kann also mitgeteilt werden, welcher Benutzer gerade mit ihm kommuniziert. Die *Basic Authentication* wird mit dem Benutzernamen und dem Passwort durchgeführt und wurde für das HTTP-Protokoll spezifiziert [56]. Soll eine HTTP-Anfrage authentifiziert werden, wird das *Header*-Feld „ Authorization“ genutzt. In diesem Feld wird ein String gesetzt, der sich aus dem Benutzernamen, gefolgt von „:“ und dem Passwort zusammensetzt. Dieser String wird anschließend Base-64 verschlüsselt. Eine *Basic Authentication* mit dem Benutzernamen „test“

und dem Passwort „123456“ würde folgendermaßen aussehen:

„Authorization: Basic dGVzdDoxMjM0NTY=“.

Basic Authentication sollte ohne zusätzliche Sicherheitsmechanismen nicht für vertrauliche Daten genutzt werden, weil eine Übertragung in Klartext vorliegt [56, 19].

4.4.3. Microservice für die Authentifizierung

Für die *Middleware* wurde ein separater Dienst für die Authentisierung und Autorisierung an den externen Diensten implementiert. Dieser Dienst ist in Form eines *Microservices* vorhanden, um konform mit der grundlegenden *Microservice*-Architektur zu sein. Weil fünf Dienste das OAUTH2-Protokoll nutzen (*Dropbox, Google Drive, Slack, Github, Bitbucket*) und der Ablauf der *Basic Authentication* für *Owncloud* trivial ist, wickelt der implementierte Dienst nur die OAUTH2-Anfragen ab. Die Authentifizierung für *Owncloud* wird direkt über die öffentliche Schnittstelle vorgenommen. Um ein Maß an Sicherheit in diesem Fall zu gewährleisten, sollte sowohl ein Frontend, welches die API der *Middleware* nutzt, als auch die öffentliche Schnittstelle über Hypertext Transfer Protocol Secure (HTTPS) gehostet werden [57]. Der Service für die Authentifizierung besitzt folgende Aufgaben:

- Alle fünf Dienste nutzen ihren *Authorization Server* für den in Abbildung 4.4 zu sehenden *Authorization Request*. Der *Microservice* stellt für jeden Dienst eine URL zur Verfügung, die zu dem jeweiligen *Authorization Server* zeigt. Der URL angehängt ist ein *state Query*-Parameter, bestehend aus dem Namen des Nutzers, der die Anfrage stellt. Dieser wird zu einem späteren Zeitpunkt benötigt [55, 24].

- Nachdem ein Nutzer die URL für die Autorisierung erhalten hat und zu diesem navigiert, muss er sich mit den Anmeldedaten für den entsprechenden Online-Dienst anmelden und die Autorisierung bestätigen. Durch die Bestätigung wird der Schritt B in Abbildung 4.4 eingeleitet. Das empfangen des *Authorization Grant* ist die nächste Aufgabe des *Microservices*. Dazu wird für jeden Dienst eine entsprechende *Callback-URL* angelegt. Über diese URL wird der *Grant* in Form eines Codes empfangen. Ebenfalls Teil der Anfrage ist der *state*-Parameter, der im Schritt zuvor mitgegeben wurde.

- Der Empfangene Code kann nun verwendet werden, um das *Access Token* und einen möglichen *Refresh Token* zu erhalten. Dafür wird der Code zusammen mit einer Client ID und einem *Client Secret* an den *Authroization server* des jeweiligen Dienstes geschickt [55, 15]. *Client ID* und *Client Secret* werden von dem *Authorization Server* bereitgestellt, wenn ein *Client*, in dem Fall die *Middleware*, sich für den jeweiligen Dienst registriert.

Sind alle Daten korrekt, antwortet der *Authorization Server* mit dem *Access Token* und einem möglichen *Refresh Token*. Diese können für den Zugriff auf die Daten des Benutzers innerhalb des Online-Dienstes verwendet werden.

- Die letzte Aufgabe des *Microservices* ist das Persistieren der Daten. Dafür werden die Tokens zusammen mit dem Benutzernamen, der im *state*-Parameter zu finden ist, an den *User Management Microservice* über GRPC gesendet und dort in einer MongoDB *Collection* gespeichert.

Ob eine API mit OAUTH2 *Refresh Tokens* zurückgibt, hängt von der internen Implementation ab. Die Spezifikation gibt vor, dass Anfragen mit dem *Grant Type* „Authorization Code Grant" einen **Refresh Token** zurückgeben sollen, aber nicht müssen [55, 23]. Von den fünf Diensten mit OAUTH2-Unterstützung geben *Google Drive* und *Bitbucket* ein *Refresh Token* zurück. Als weiterer Parameter wird in diesem Fall ein *Timestamp* in Sekunden zurückgeben, der den Ablaufzeitpunkt des *access tokens* signalisiert. Die anderen vier Online-Dienste nutzen lediglich *Access Tokens*. Die Verwendung von *Refresh Tokens* kann die Sicherheit eines Dienstes erhöhen. Erhält ein Unbefugter Zugriff auf ein *Access Token* mit einem unbegrenzten Ablaufdatum, können für eine unbestimmte Zeit Daten eines Nutzers entwendet werden. Werden hingegen *Refresh Tokens* genutzt, haben die *Access Tokens* oftmals eine geringe Lebensdauer, weil es möglich ist diese zu erneuern und Angreifern nur ein kleines Zeitfenster für die Manipulation der Daten geboten wird.

Um das Zusammenspiel der verschiedenen *Microservices* und der externen Dienste im Zusammenhang mit OAUTH2 aufzuzeigen, ist in Abbildung 4.5 ein Sequenzdiagramm zu sehen. Beispielhaft dargestellt ist der Prozess für die Autorisierung von *Google Drive*. Die Aktoren RESTful-API, *Authentication Service* und *UserManagement Service* sind Teil der *Middleware*, wohingegen der *Google Authorization Server* ein von *Google* bereitgestellter Dienst für das OAUTH2-Protokoll ist. Nachdem die Autorisierung wie im Diagramm dargestellt vollzogen wurde, kann der Nutzer Anfragen über die *Middleware* an den *Resource Server* von *Google Drive* stellen. Läuft das *Access Token* ab, wird es mithilfe des *Refresh Tokens* erneuert. Dies läuft folgendermaßen ab: Bevor die *RESTful*-API eine Anfrage an den jeweiligen *Connector* (3.2.4) weitergibt, wird das passende Token für die Autorisierung beim *Usermanagement-Microservice* abgefragt. Stellt dieser anhand des Zeitstempels fest, dass das Token abgelaufen ist, wird es über den *Authentication Service* erneuert.

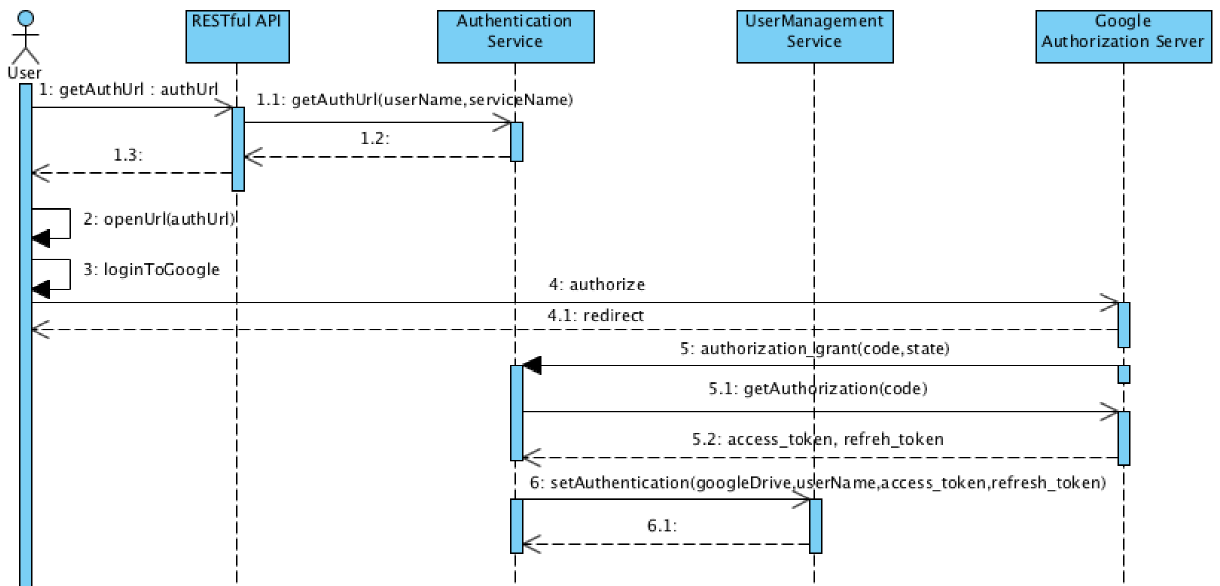


Abbildung 4.5.: Beispiel für die Autorisierung anhand von *Google Drive*

4.5. Erweiterbarkeit und Wartbarkeit der RESTful-API

Die *RESTful*-API dient in der vorliegenden *Microservice*-Architektur als *Gateway*, um Anfragen an die jeweiligen *Connectoren* weiterzugeben. Dabei durchläuft jede Anfrage folgende Instanzen:

- Die Authentifizierung an der *Middleware* über den *User Management Service*. Dafür wurde die bereits erwähnte *Basic Authentication* genutzt. Das Anbieten eines eigenen OAUTH2-Servers, welcher eine sichere Alternative bieten würde, übersteigt den Rahmen der vorliegenden Ausarbeitung.
- Das Initialisieren des *GRPC*-Clients für die Netzwerkkommunikation.
- Die Kontrolle des HTTP-Requests hinsichtlich der erforderlichen *Query*-und *Header*-Parameter.
- Die Abfrage des Tokens für die Autorisierung des jeweiligen Online-Dienstes beim *User Management*-Service.

- Das Aufrufen der passenden GRPC-Funktion, die die Anfrage an den jeweiligen *Connector* weitergibt. Dabei findet außerdem ein mapping der HTTP-Request-Parameter auf die GRPC-Funktionsparameter statt.

- Die Antworten auf den HTTP-Request. Dafür wird das Ergebnis der GRPC-Funktion nach JSON umgewandelt und dem *Response Body* des HTTP-Requests zugeordnet.

Um diese Aufgaben nicht für jeden Dienst und dessen Routen separat programmieren zu müssen, wurde ein Format in JSON entwickelt, mit dem ein Dienst innerhalb der *Middleware* für die *RESTful*-API beschrieben werden kann (4.7, 4.8).

```
1 {
2 "service_name": "string",
3 "grpc_ip": "string",
4 "grpc_port": "integer",
5 "grpc_service_name": "string",
6 "grpc_package_name": "string",
7 "authentication_type": "string",
8 "requests": [Request]
9 }//Config
```

Listing 4.7: JSON-Format für die Konfiguration eines Dienstes innerhalb der RESTful-API

```
1 {
2 "http-method": "string",
3 "route": "string",
4 "query_parameter": ["string"],
5 "header_parameter": ["string"],
6 "grpc_function": "string",
7 "response_parameter": ["string"]
8 }
9 //Request
```

Listing 4.8: JSON Format für einen Request

Für jeden Online-Dienst, der Teil der *Middleware* ist, wurde eine Konfigurationsdatei in diesem Format erstellt. Die Verbindungen zu den jeweiligen *Connectoren* werden anhand der Konfiguration hergestellt. Außerdem werden die Routen, die die *RESTful*-API anbietet, anhand des *Request*-Arrays (Zeile 8 in 4.7) für jeden Dienst erstellt. So müssen Änderungen nur in einer Datei angepasst werden, wodurch die Wartbarkeit erhöht wird. Außerdem können neue Dienste schneller implementiert werden, was wiederum die Erweiterbarkeit steigert.

Im Abschnitt 3.3.3.2 wurde bereits REST als Architekturparadigma vorgestellt. Wichtig in diesem Fall sind die verschiedenen Stufen einer *RESTful*-API, mit denen ausgedrückt werden kann, wie genau das Architekturparadigma umgesetzt wurde. [58]

- Level 0: Die erste Stufe einer *RESTful*-API wird sehr rudimentär beschrieben. Das für die Implementation genutzte Protokoll soll als Transportprotokoll dienen. REST als Architekturparadigma gibt zwar kein spezielles Protokoll vor, dennoch werden die

meisten von ihnen über HTTP angeboten. Des Weiteren sollen Anfragen anhand von Uniform Resource Identifiers (URIs) gestellt werden.

- Level 1: Die nächste Stufe führt die Ressourcen ein. Anstatt Anfragen explizit an ein System oder Dienst zu stellen, werden die Ressourcen direkt angesprochen. Als Beispiel könnte die Ressource */hotels* angesprochen werden, die eine Liste von verschiedenen Hotels zurückgibt. Aus dieser Liste kann ein Hotel ausgewählt werden, um Daten für das explizite Hotel zu erhalten (*/hotels/beispielhotel*).

- Level 2: Diese Stufe führt die Benutzung der HTTP-Verben ein. Während die vorigen Stufen ebenfalls HTTP-Verben nutzen, beschreibt diese Stufe die korrekte Benutzung anhand der Spezifikation von HTTP [57]. Zum Beispiel soll der lesende Zugriff auf Ressourcen über *GET* erfolgen. *GET* wird auch als eine *safe operation* bezeichnet, diese besagt, dass das Ausführen der Methode keine signifikante Auswirkung auf die Ressource hat. Für das Ändern oder Hinzufügen von Ressourcen sollen *PUT* und *POST* genutzt werden. Das Löschen soll über das Verb *DELETE* stattfinden. Eine weitere Anforderung an diese Stufe ist das Benutzen von HTTP-Statuscodes, um Anwendern eine hilfreiche Rückmeldung zu liefern. Dies erleichtert den Umgang mit der API.

- Level 3: Diese Stufe ist auch unter dem Namen Hypertext As The Engine Of Application State (HATEOAS) bekannt. Antworten der API sollen Verlinkungen, also URIs für andere Ressourcen beinhalten, um dem Anwender mitzuteilen, wie er nach der Anfrage weiter verfahren kann. Dies ermöglicht dem Entwickler der API, bestehende URIs von Ressourcen ohne größere Auswirkungen umbenennen zu können. Diese Stufe kann außerdem als Dokumentation der API gesehen werden.

Die *RESTful*-API der *Middleware* erfüllt die Anforderungen bis zur zweiten Stufe. Die dritte Stufe stellt den höchsten Aufwand dar und hat keinen Einfluss auf die Funktionalität, wodurch sie für die prototypische Implementation nicht berücksichtigt wurde. Durch die Berücksichtigung von Konventionen wird die Wartbarkeit und Erweiterbarkeit deutlich vereinfacht.

5. Anwendung der *Middleware*

In diesem Kapitel wird die Anwendung der implementierten *Middleware* behandelt. Zunächst wird die Auswahl des *Frameworks* für die Entwicklung diskutiert. Anschließend wird die ausgewählte Plattform und das Ergebnis des *Frontends* vorgestellt. Abschließend werden die Anforderungen anhand von Anwendungsfällen validiert.

5.1. *Frontend*-Entwicklung

5.1.1. Anforderungen

Die Anforderungen an das beispielhafte *Frontend* und das für die Entwicklung benötigte *Framework* werden anhand der Zielgruppenanalyse und der Eigenschaften der *Middleware* bestimmt. Im Abschnitt 3.1.1.3 wurden die Ergebnisse der Zielgruppenanalyse zusammengefasst. Dabei wurde festgestellt, dass die Befragten hauptsächlich einen Internet-Browser nutzen, um auf Online-Dienste zuzugreifen. Dieser Zugriff erfolgt zum größten Teil mit dem PC, wobei ein nennenswerter Anteil der Nutzerschaft auch mobile Geräte wie Smartphones und Tablets nutzt. Daraus leitet sich die erste Anforderung an das *Frontend* ab. Eine *Cross-Plattform*-Lösung in Form einer Web-Anwendung mit einem responsivem Webdesign. Dadurch können die meisten Nutzer angesprochen und das ursprüngliche Nutzungsverhalten aufgegriffen werden. Eine weitere Anforderung entsteht durch das Verhalten einer API, wie sie von der *Middleware* angeboten wird. Das *Frontend* erhält die darzustellenden Daten mithilfe von HTTP-Anfragen an die API. Dabei werden die Elemente der Webseite ständig verändert. Traditionelle Ansätze der Web-Entwicklung laden bei jeder Änderung die komplette Seite neu, wodurch höhere Latenzzeiten entstehen [59]. Stattdessen können sogenannte Single Page Applications (SPAs) genutzt werden. SPAs können im *Internet-Browser* ausgeführt werden und benötigen kein erneutes Laden bei der Benutzung. SPAs machen sich Asynchronous JavaScript And XML (AJAX) [60] zunutze, wodurch eine Webseite asynchron aktualisiert werden kann, ohne den eigentlichen Webserver ansprechen zu müssen, der die Webseite *hostet*. Das *Framework* für die Entwicklung der Webseite soll also die Erstellung von SPAs ermöglichen.

5.1.2. Auswahl des *Frameworks*

Für die Entwicklung des *Frontends* als *Single Page Application* wurden insbesondere *Angular2* und *React* als für diese Ausarbeitung infrage kommende *Frameworks* identifiziert. Beide dieser *Frameworks* werden aktiv weiterentwickelt und von einer großen *Community* begleitet [61] [62]. *React* ist eine von *Facebook* genutzte und entwickelte *Javascript*-Bibliothek zum Erstellen von Benutzeroberflächen [63]. Die Bibliothek nutzt ein Komponenten-Modell. Jede Komponente verwaltet seinen eigenen Zustand. Die Komponenten können miteinander komponiert werden, um komplexe Benutzeroberflächen bereitzustellen. Die Kommunikation mit dem Document Object Model (DOM) wird über das sogenannten *virtual-DOM* abstrahiert [64, 6]. Jede Komponente besitzt eine *render*-Funktion, über die das *virtual-DOM* angesprochen werden kann.

Angular2 wird von *Google* entwickelt und bietet ein *Framework* für die Entwicklung von Web-Applikationen [65]. *Angular2*-Anwendungen können mit *Javascript* entwickelt werden, *Google* empfiehlt jedoch *Typescript* zu nutzen [66]. *Typescript* ist eine Obermenge von *Javascript* und wird vor dem *Deployment* zu *Javascript* kompiliert [67]. *Javascript*-Bibliotheken können in *Typescript* eingebunden werden. Die wichtigsten Vorteile von *Typescript* sind: Typisierung und *Interfaces*. Ähnlich zu *React* nutzt auch *Angular2* ein Komponenten-basiertes Modell. Allerdings werden die Komponenten in Modulen gesammelt. Eine Web-Applikation kann aus einem oder mehreren Modulen mit jeweils einem oder mehreren Komponenten bestehen. *Angular2* nutzt *Templates* für die Darstellung im *Browser*. Jeder Komponente wird ein *Template* zugeordnet. Sollen Daten aus der Komponente an das DOM weitergegeben werden, wird das *Two-Way-Data-Binding* genutzt. Änderungen werden dadurch sowohl im DOM sichtbar als auch in der eigentlichen Komponente.

Sowohl *React* als auch *Angular2* verfolgen das Ziel, vollständige Anwendungen an den *Browser* zu liefern. SPAs können also mit beiden *Frameworks* beziehungsweise Bibliotheken erstellt werden. Für die Entwicklung des beispielhaften *Frontends* wurde schlussendlich *Angular2* gewählt. Im Gegensatz zu *React* bietet *Angular2* einen unterstützenden Technologie-Stack an, um Web-Anwendungen zu entwickeln. Dazu zählen:

- *Angular CLI*: *Google* bietet ein Kommandozeilen-Programm für die Erstellung von Web-Anwendungen mit *Angular2*. Es ermöglicht das Generieren von lauffähigen Projekten und von einzelnen Komponenten. Der generierte *Code* folgt den empfohlenen *Best Practices*. Darüber hinaus bietet *Angular CLI* die Möglichkeit, das Projekt sowohl im *Development*-

als auch im *Production*-Mode auszuführen. *Angular CLI* kann also begleitend für das gesamte *Angular2*-Projekt genutzt werden. [68]

- *Protractor*: Wichtig für das Testen der Anforderungen an die *Middleware* mithilfe des *Frontends* ist ein *Framework*, welches den Zugriff auf den *Browser* automatisieren kann. Zu diesem Zweck hat *Google Protractor* entwickelt. Ursprünglich diente *Protractor* zum Testen von *Angular1*-Anwendungen, dem Vorgänger von *Angular2*. Seit Version 2.5.0 wird allerdings auch *Angular2* unterstützt [69]. Tests mit *Protractor* können direkt über die *Angular CLI* gestartet werden. [70]
- Angular Material: Das Material Design ist eine von *Google* entwickelte Designsprache. Es wird das sogenannte *Flat Design* verwendet, welches auf realistische, zurückgenommene Darstellungen abzielt. Angular Material bietet vorgefertigte Komponenten für *Angular2* an. [71]

Typescript ist ein weiteres Argument für die Benutzung von *Angular2*. Durch die Typisierung und die Möglichkeit Interfaces zu benutzen, können wartbare und erweiterbare Web-Anwendungen entwickelt werden. Auch die Verwendung von verschiedenen Komponenten wie *Services, Modules, Components* und *Directives* (5.1.3.1), unterstützen die Entwicklung einer robusten Web-Anwendung.

5.1.3. Angular2

In diesem Abschnitt wird *Angular2* mit den einzelnen Bestandteilen des *Frameworks* erklärt.

5.1.3.1. Architektur

Die Architektur von *Angular2* setzt sich aus mehreren Elementen zusammen. Im Folgenden werden die wichtigsten von ihnen erklärt:

- *Module*: *Angular Modules* ermöglichen das Organisieren einer Applikation in kohäsive und funktionale Blöcke. Mithilfe des *Decorators* „@NgModule“ wird eine Klasse als *Module* identifiziert. Das *Module* deklariert *Components, Directives, Pipes* und *Services*, aus denen das *Module* zusammengesetzt ist. Jede *Angular2*-Applikation besitzt ein sogenanntes *Root-Module*, welches als Einstiegspunkt fungiert. Bibliotheken für *Angular2* werden als *Modules* bereitgestellt, um die Nutzung in anderen Projekten zu ermöglichen. [72]
- *Component*: Ein *Component* kontrolliert einen Teil der *View*, also der Web-Anwendung. Ein Navigationsmenü, eine Tabelle oder ein Login-Formular können beispielsweise mittels eines *Components* bereitgestellt werden. Durch *Angular* werden die *Components* zur

Laufzeit erstellt, geändert und zerstört. Entwickler können mit sogenannten *Lifecycle Hooks* [73] auf diese Ereignisse reagieren. Jedem *Component* wird ein *Template* zugeordnet, wodurch *Angular* mitgeteilt wird, wie das *Component* dargestellt werden soll. Das *Template* ist wie eine HTML-Datei mit einigen *Angular*-spezifischen Funktionen, den sogenannten *Directives*, aufgebaut. [74]

- *Directive*: Die für die Darstellung verwendeten *Components* in *Angular2* sind dynamisch. *Directives* werden von *Angular2* in Elemente für das DOM gewandelt. Mit dem *Decorator* „@Directive“ wird eine Klasse als *Directive* deklariert. Ein *Component* gilt als *Directive* mit einem *Template*. Weitere *Directives* sind *structural* und *attribute Directives*. Das Layout der Anwendung und somit die Elemente des DOM können mit den *strucutural Directives* verändert werden. In 5.1 ist eine solche *Directive zu sehen.*

```
<li *ngFor="let number of numbers">
  <p *ngIf="number%2 === 0">{{number}}</p>
</li>
```

Listing 5.1: Beispiel von *structural Directives* mit *ngIf und *ngFor

Für jede Zahl im *numbers*-Array wird ein <li> Element im DOM erstellt. Ist die Zahl gerade, wird die Zahl in der Anwendung als Absatz dargestellt. *Attribute Directives* hingegen verändern das Verhalten oder die Darstellung eines bereits bestehenden Elements. Es können zum Beispiel CSS-Klassen dynamisch hinzugefügt oder verändert werden.

- *Service*: Ein *Service* in *Angular2* ist nicht genau spezifiziert. Am häufigsten besteht ein *Service* aus einer Klasse, die nur eine bestimmte Funktion hat. Das Loggen von Ereignissen oder das Umrechnen von Währungen könnten als *Service* implementiert werden. Die Kommunikation zu einem *Backend* sollte immer durch *Services* implementiert werden. *Components* sollten einfach gehalten werden und Aufgaben an *Services* abgeben. [74]

Wie die einzelnen Komponenten der *Angular2*-Architektur miteinander interagieren ist in der Abbildung 5.1 dargestellt. Zu sehen sind die Bestandteile eines beispielhaften *Modules*. Wichtig für das Verständnis von *Angular2* ist die *Dependency Injection* [75]. Mithilfe der *Dependency Injection* können neue Instanzen von *Services* mit allen benötigten Abhängigkeiten einer *Component* beigefügt werden. Dazu erstellt ein sogenannter *Injector* alle *Services*, die einem Konstruktor beigefügt wurden, und übergibt sie nach der Instanzierung dem *Component*. Das Bereitstellen von Daten aus dem *Component* für das *Template* wird *Property Binding* oder auch *Data Binding* genannt [76]. Der Zugriff auf *Events*, zum Beispiel das Klicken eines *Buttons*,

wird mit dem *Event Binding* realisiert. Dadurch kann über das *Component* programmatisch auf *Events* reagiert werden [76].

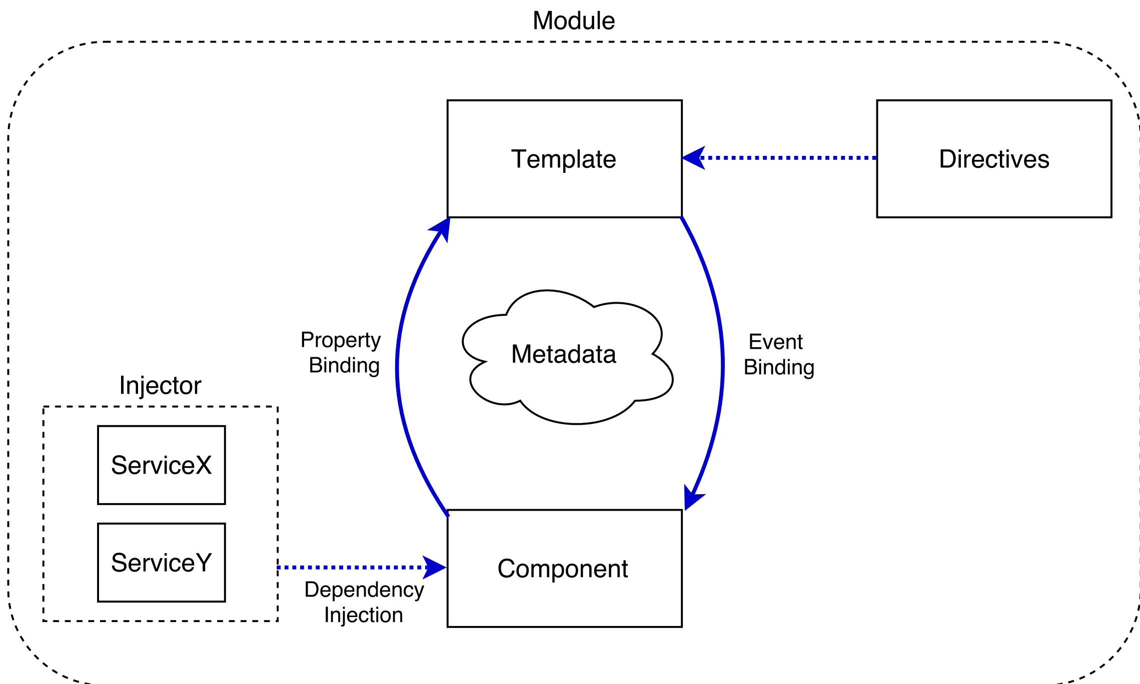


Abbildung 5.1.: *Angular2*-Architektur abstrakt dargestellt (angelehnt an [74])

5.1.3.2. Just in Time Compilation

Um eine mit *Angular2* entwickelte Anwendung mit dem *Browser* darstellen zu können, müssen die *Components* und *Templates* von dem *Angular Compiler* in ausführbaren *Javascript*-Code übersetzt werden. Dieses Verfahren ist der Standard für den Entwicklungsprozess und wird auch *Just in Time Compilation* genannt. Der Nachteil dieser Methode ist zum einen die erhöhte Ladezeit, weil der *Browser* die Kompilierung vornimmt. Zum anderen weist die Anwendung eine höhere Datenmenge auf, weil der Compiler mitgeliefert werden muss. [77]

5.1.3.3. Ahead of Time Compilation

Im Gegensatz zur *Just in Time Compilation* wird bei der *Ahead of Time Compilation* die Anwendung vor der Bereitstellung kompiliert. Es wird also nur einmal der Programmcode kompiliert

und dann über einen Server bereitgestellt, anstatt eine Version der Anwendung bereitzustellen, die von jedem *Browser* vor der Benutzung kompiliert werden muss. Durch diese Form des *Deployments* kann die Anwendung schneller dargestellt werden, weil keine Kompilierung nötig ist. Außerdem wird die Datenmenge der Anwendung deutlich reduziert, da der *Angular Compiler* nicht mitgeliefert werden muss. Diese Art von Kompilierung erfordert allerdings einige Änderungen am Quellcode. Es müssen beispielsweise die Konfigurationsdateien der Anwendungen angepasst und auch die Ordnerstruktur muss umgestellt werden. Öffentlich bereitgestellte Bibliotheken unterstützen oft keine *Ahead of Time Compilation*, wodurch Projekte die von diesen abhängig sind, nicht problemlos auf diese Form des *Deployments* umstellen können. Mit dem Kommandozeilen-Programm *Angular Cli* kann das Projekt *Ahead of Time* kompiliert werden. Für die Entwicklung sollte allerdings darauf verzichtet werden, weil der Vorgang einige Minuten in Anspruch nimmt. Zum Testen der Web-Anwendung sollte also die *Just in Time Compilation* verwendet werden. [77]

5.1.3.4. Lazy Loading

Angular2 erlaubt das *Lazy Loading* [78] von *Modules*. Ein auf diese Weise implementiertes *Module* wird erst dann geladen, wenn es auf der Webseite angezeigt werden soll. Dadurch können sich Ladezeiten deutlich reduzieren. Besonders für Anwendungen mit mehreren Unterseiten bietet sich das *Lazy Loading* an. Um eine Anwendung auf das *Lazy Loading* umzustellen, müssen einige Änderungen vorgenommen werden. Die Struktur der Web-Anwendung muss dahingehend umgestellt werden, dass alle *Components* die Teil einer Unterseite sind, in einem *Module* zusammengefasst werden. Diese *Modules* müssen beim *Angular Router* [79], der für das clientseitige Routing verantwortlich ist, bei den entsprechenden Routen hinterlegt werden. Bei einer Anwendung, die auf das *Lazy Loading* verzichtet, würden *Components* anstatt *Modules* an den Routen anliegen.

5.1.4. Ergebnis

Das *Frontend* wurde als Webseite mit dem bereits vorgestellten Technologie-Stack implementiert. *Angular CLI* wurde für die Verwaltung des Projektes verwendet. Damit konnten Bibliotheken eingebunden und der Code für die Entwicklung und Produktion sowohl angepasst als auch ausgeliefert werden. Für ein einheitliches Design wurde *Angular Material* als UI-*Framework* verwendet. Die Webseite besteht aus drei verschiedenen Routen: *login, register* und *home*. Trotz dem SPA-Aufbau kann durch clientseitiges *Routing* auf der Webseite navigiert werden. Ermöglicht wird dies durch den *Angular Router* [79]. Das Erstellen eines neuen Nutzers für die *Middleware* wird unter der Route „register“ angeboten. Das Einloggen eines bestehenden

Nutzers hingegen unter der Route „login“. Die Kerninhalte der *Middleware* werden allesamt auf der Hauptseite angeboten. Die Webseite wurde auf drei Unterseiten aufgeteilt, weil das Erstellen und Authentifizieren eines Nutzers Voraussetzung ist, um auf die Funktionen zugreifen zu können und sich somit eine strukturelle Trennung anbietet. Die Kernfunktionalität der *Middleware* wird über die Hauptseite des *Frontends* angeboten. In der Abbildung 5.2 ist ein *Mockup* zu sehen, welches das Endergebnis der Hauptseite darstellt.

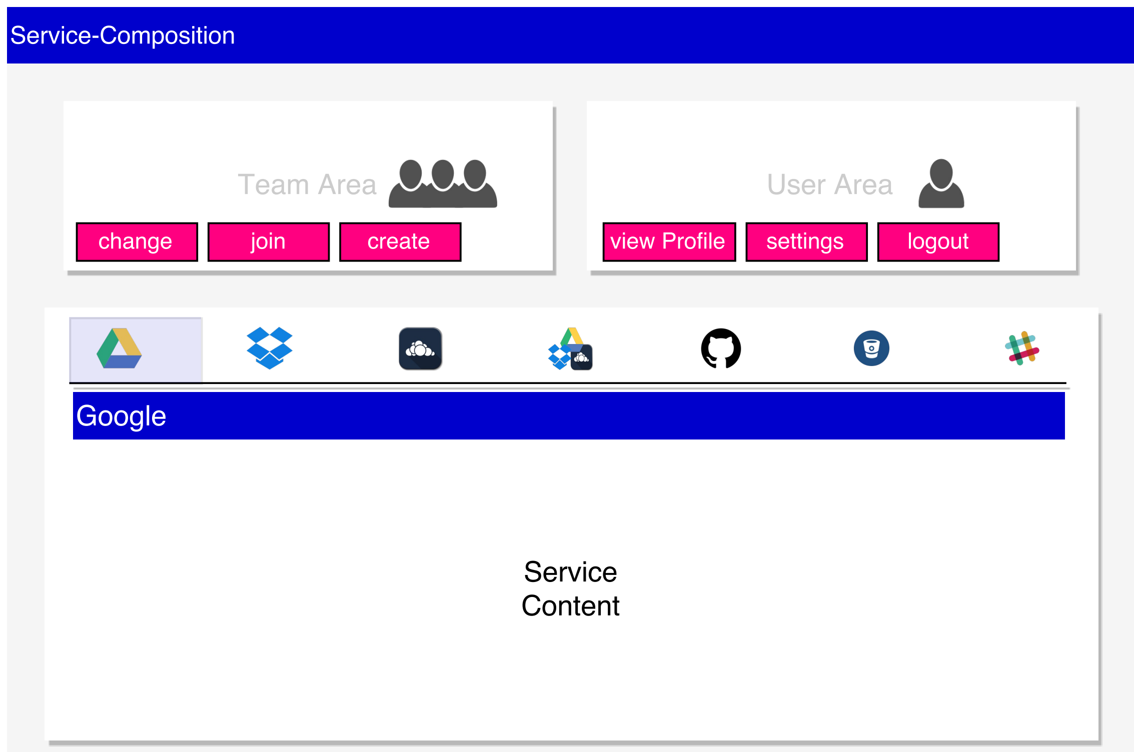


Abbildung 5.2.: Vereinfachtes Mockup der Webseite

Das für das Material Design typische Kartendesign ist gut zu erkennen. Dabei werden Komponenten innerhalb einer Karte dargestellt, die mit dezentem Schatten vom Hintergrund hervorgehoben werden. Die Seite ist in drei Elemente aufgeteilt: *Team Area, User Area* und ein Bereich für die implementierten Online-Dienste. Die Funktionalität der Hauptseite richtet sich an den im Abschnitt 3.1.4 festgelegten Anforderungen. Über die *Team Area* können Teams erstellt, beigetreten und geändert werden. Die wichtigste Funktionalität der *User Area* ist die Autorisierung der Online-Dienste. Über den *Button* „settings“ wird ein Dialog geöffnet, in dem

die einzelnen Dienste autorisiert werden können. Das große, untere Element fungiert als Container für den Inhalt der Online-Dienste, die ihre Daten von der *Middleware* beziehen. Dabei wurde das *Tab*-Element aus dem *Angular Material* UI-Framework genutzt [80], um zwischen den Diensten hin- und herzuschalten. Um die generelle *User Experience* beim Benutzen der Web-Anwendung zu verbessern, wurden Animationen für das Warten auf Zustandsübergänge verwendet. HTTP-Anfragen an die *Middleware*, die eine unbestimmte Zeit in Anspruch nehmen, werden mit einem *Progress Spinner* [81] dargestellt. Für die Aktualisierung von Inhalten der Online-Dienste wurde eine Animation gewählt, mit der die neuen HTML-Elemente horizontal in die Darstellung gleiten. Entwickelt wurde diese Animation über ein natives *Feature* von *Angular2*, mit dem Animationen in *Typescript* geschrieben werden können, ohne CSS zu verwenden [82]. Ein Screenshot des *Frontends* und ein Link zu der Webseite sind als Anlage beigefügt [B.1]. Um die Ladezeiten beim initialen Start der Webseite und beim Navigieren zu verringern, wurden die beiden bereits angesprochenen Techniken *Ahead of Time Compilation* (5.1.3.3) und *Lazy Loading* (5.1.3.4) umgesetzt. Ein Problem entstand bei der *Ahead of Time Compilation*. Für die Darstellung von Notifikationen in der Web-Anwendung wurde eine externe Bibliothek verwendet [83], die einen *Angular Service* dafür anbietet. Dieser ist allerdings nicht kompatibel für die *Ahead of Time Compilation*. Für die Lösung wurde ein *Angular Service* geschrieben, der für die Darstellung der Notifikationen die *Snackbar*-Dialoge von *Angular Material* nutzt. Um nicht alle *Components* umschreiben zu müssen, die den inkompatiblen *Service* verwendet haben, wurde das gleiche öffentliche Interface mit folgenden Methoden verwendet: onError und onSuccess. Für das *Lazy Loading* wurde die Web-Anwendung in vier *Modules* aufgeteilt. Jeweils ein *Module* für die Routen: *login, register* und *home*. Das letzte *Module* umfasst die Eigenschaften, die alle anderen *Modules* benötigen. Durch diese Strukturierung muss das geteilte *Module* mit den Eigenschaften nur einmal geladen werden. Außerdem wird die Wartbarkeit und Erweiterbarkeit der Anwendung verbessert. Um die Auswirkungen der Änderungen auf die Ladezeiten zu testen, wurde die Anwendung *Just in Time* und *Ahead of Time* kompiliert. Die Ergebnisse sind in der Abbildung 5.1 dargestellt.

Kompilierung	**Bundle Größe**	**initiale Ladezeit**	**Ladezeit Login > Home**
Just in Time	7,2 MB	2,55 s	2,95 s
Ahead of Time	3,2 MB	2,01 s	2,55 s

Tabelle 5.1.: Vergleich zwischen *Just in Time* und *Ahead of time* Kompilierung

Der Test wurde lokal ausgeführt, um Schwankungen der Übertragungsrate zu minimieren. Die Gesamtgröße der kompilierten Daten (*Bundle*) wurde durch die *Ahead of Time Compilation*

um 56 Prozent verringert. Die initiale Ladezeit und die Ladezeit der Navigation von Login zur Hauptseite hat sich um 0,44 Sekunden respektive 0,4 Sekunden verringert. Die Differenz in den Ladezeiten entsteht durch die Kompilierung, die bei der *Ahead of Time Compilation* nicht mehr nötig ist. Wird die Seite über das Internet geladen, entstehen noch größere Unterschiede. Besonders für Nutzer, die über das mobile Netz auf die Webseite zugreifen, macht sich eine deutlich geringere *Download*-Größe positiv bemerkbar.

Eine besondere Herausforderung entsteht beim Umgang mit dem HTTP-Client von *Angular2* [84]. Die *File-Storage*-Dienste ermöglichen das Hoch- und Herunterladen von Dateien. Um dem Nutzer eine Rückmeldung über den Status einer Transaktion geben zu können, muss der Fortschritt des HTTP-Requests bekannt sein. Die zu diesem Zeitpunkt aktuelle *Angular2* Version 2.4.6 bietet derzeit keine Möglichkeit dafür. Der *Angular2* HTTP-Client bildet einen Wrapper um den *XMLHttpRequest* [85], der für das einfache Laden von Daten einer URL genutzt wird. Der Wrapper bietet zwar die wichtigsten Funktionen des *XMLHttpRequests* an, allerdings nicht den Zugriff auf das *Progress-Event*, mit dem die bereits empfangenen Daten gelesen werden können und somit ein Fortschritt der Transaktion abgelesen werden kann. Um dieses Problem zu lösen, wurde das *XHRBackend* überschrieben, über das die HTTP-Anfragen normalerweise bearbeitet werden. In der Dokumentation des *XHRBackends* empfiehlt *Google* diese Klasse zu überschreiben, wenn die Funktionalität erweitert werden soll [86]. In der ursprünglichen Implementierung wird ein *Observable* zurückgegeben, welches sobald der komplette HTTP-*Request* bearbeitet wurde die *Response* zurückgibt. *Observables* wurden von *ReactiveX* für die asynchrone Programmierung entwickelt und werden innerhalb von *Angular2* verwendet [87]. Sie fungieren nach den Prinzipien des *Observer*- oder auch *Reactor-Patterns*. Klassen können sich bei einem *Observable* anmelden und Daten von diesem erhalten. Ein *Observable* implementiert drei Methoden:

- *onNext*: Diese Methode wird vom *Observable* aufgerufen, um neue Daten an die *Observer* weiterzugeben. Die Daten werden auch als *Items* bezeichnet.
- *onError*: Ist ein Fehler bei der Generierung der erwarteten Daten aufgetreten, wird diese Methode aufgerufen.
- *onCompleted*: Werden keine weiteren Daten mehr übermittelt, wird diese Methode aufgerufen. Bei einem Fehler wird nur die *onError*-Methode aufgerufen. [87]

Das überschriebene *XHRBackend* gibt nicht mehr nur das Endergebnis des *Requests* zurück, sondern auch die *Progress-Events*. Als Folge müssen Klassen, die das neue *XHRBackend* für

HTTP-Anfragen nutzen, zwischen den Objekten, die über das *Observable* empfangen wurden, unterscheiden.

5.2. Validierung der Anwendungsfälle

Um die Funktionsweise des *Frontends* testen zu können, wurden Anwendungsfälle festgelegt, die die Anforderungen an die *Middleware* abdecken. Diese Anwendungsfälle wurden mithilfe von *End-to-End*-Tests modelliert. Dazu wird *Protractor* genutzt, ein Test-*Framework* für *Angular*-Applikationen. Mithilfe von *Protractor* kann der Zugriff auf Webseiten automatisiert werden, indem Benutzeraktionen wie zum Beispiel das Klicken auf ein DOM-Element, ausgelöst werden. Der technische Aufbau von *Protractor* ist in Abbildung 5.3 dargestellt.

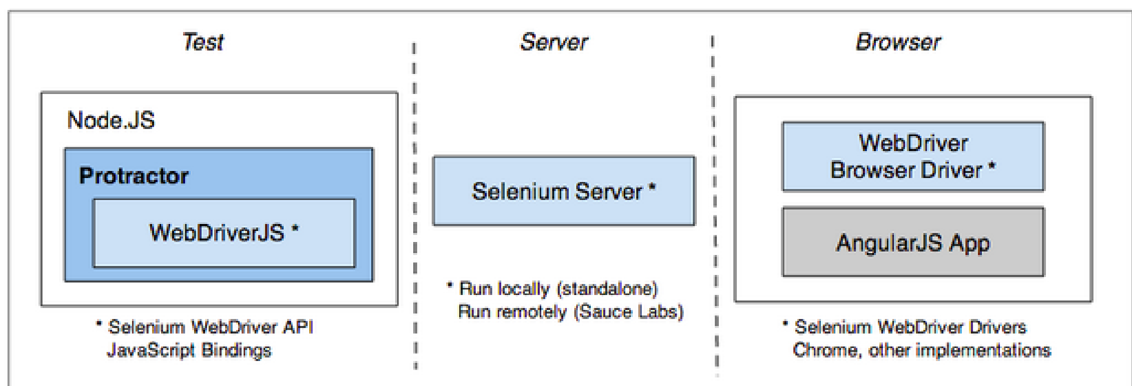


Abbildung 5.3.: Technischer Aufbau von *Protractor* [88]

Protractor wurde als *Node.js* Programm entwickelt. Der Zugriff auf den *Browser* wird über *Selenium* realisiert, ein *Browser Automation Framework* [89]. *Protractor* fungiert als *Wrapper* für die *Webdriver* API von *Selenium*. In diesem Fall wird die *Javascript*-Version des *Webdrivers* genutzt, die *Selenium* allerdings auch in verschiedenen anderen Programmiersprachen von anbietet. Die Kommunikation mit dem *Browser* und der Applikation findet über den *Selenium Server* statt. Dieser kommuniziert über ein JSON-Protokoll, dem *Webdriver Wire Protocol* mit dem *Browser Driver*, welcher für alle gängigen *Browser* vorhanden ist. Die Tests für *Protractor* können mit den Test-*Frameworks Jasmine* oder *Mocha* geschrieben werden. Für die entwickelten Tests wurde *Jasmine* genutzt, weil es der Standard für *Protractor* ist. Die Tests wurden nach den offiziellen Gestaltungsrichtlinien von *Protractor* aufgebaut [90]. Dabei sollen Testdateien

für jede Unterseite der Webseite erstellt werden. Außerdem sollen „*Page Objects*“ deklariert werden, welche die Elemente und Informationen auf der Unterseite beschreiben. Für jede Unterseite soll wiederum eine Testdatei erstellt werden, die auf das *Page Object* zugreift und die Testszenarien programmatisch beschreibt. *Protractor* kann in Verbindung mit *Angular CLI* genutzt werden. Über den Befehl „ng e2e“ können alle Tests, die mit *Protractor* definiert wurden, ausgeführt werden. Getestet wurden die in Abschnitt 3.1.4 festgelegten Anforderungen.

6. Fazit

6.1. *Lessons Learned*

In diesem Abschnitt werden Erkenntnisse beschrieben, die während der Anfertigung der vorliegenden Ausarbeitung gewonnen wurden.

- **Zielgruppenanalyse**: Während der Befragungsdurchführung (3.1.1) sind Unklarheiten bei den Probanden zu einigen Fragen entstanden. Besonders die Frage bezüglich der Zugriffsrechte musste während der Befragung genauer erklärt werden. In diesem Zusammenhang war der Begriff des Drittanbieters nicht allen Probanden klar verständlich. Generell hätten erweiterte Beschreibungstexte zu den einzelnen Fragen für ein klareres Verständnis gesorgt. Ein weiteres Problem entstand bei einigen Fragen durch die Mehrfachauswahl. Bei Mehrfachauswahlen ist keine Gewichtung zwischen den verschiedenen Antworten möglich, wodurch Extremfälle nicht berücksichtigt werden. Wenn beispielsweise ein Befragter fast ausschließlich einen Online-Dienst über den PC nutzt und nur äußerst selten am *Smartphone*, kann diese Differenzierung mit dem vorliegenden Fragebogen nicht abgebildet werden (A.1). An dieser Stelle ist es sinnvoll, eine Skala zu nutzen, die die Antworten gewichtet.

- **Middleware**: Bei der Entwicklung einer *Microservice*-Architektur wird empfohlen, die einzelnen *Microservices* ohne gemeinsame Codequellen aufzubauen [25, 70]. Der Grund dafür ist das *Deployment*. Wird ein Fehler in einer geteilten Codequelle gefunden, müssen alle *Microservices*, die diese verwenden, die Änderung übernehmen und neu ausgeliefert werden. Dies kann das Ziel der *Microservice*-Architektur gefährden, einzelne *Microservices* unabhängig auszuliefern [25, 70]. Während der Implementierung der *Microservices* hat sich gezeigt, dass einige entwickelte *Microservices* große Ähnlichkeiten aufweisen. In einigen Fällen sollte darüber nachgedacht werden, entgegen der Empfehlung unterstützende Codequellen anzufertigen, die zwischen den *Microservices* geteilt werden. Das *Streaming* der Dateien der *File-Storage-Microservices* beispielsweise folgt dem gleichen Schema. Mit dem *Request-Package* (3.3.1.1) wird eine HTTP-Anfrage an die *File-Storage*-API gestellt. Sobald ein *Chunk* empfangen wurde, wird es an den *GRPC*-Client (3.3.3.1)

weitergegeben und an den *GRPC*-Server beziehungsweise die *RESTful*-API der *Middleware* gesendet. Dieser Vorgang könnte ausgelagert werden und einen *Wrapper* um die benutzten Bibliotheken ergeben: *GRPC* und *Request*. Änderungen an den einzelnen Bibliotheken betreffen ohnehin schon die *Microservices*, weil sie elementar für deren Funktionsweise sind. Der *Wrapper* würde lediglich die Funktionen der einzelnen Bibliotheken zusammenfassen.
Eine weitere Erkenntnis, die im Verlauf der Entwicklung gewonnen wurde, betrifft den Entwicklungsaufwand einer *Microservice*-Architektur. Die Programmierung eines Dienstes, der als eigenständige Applikation fungiert, erfordert mehr Aufwand als die Erstellung eines *Features* in einem Software-Monolithen. Auch die Entwicklung von passenden Schnittstellen, die über das Netzwerk angeboten werden, führt zu einer längeren Entwicklungszeit. Für die Anforderungen der *Middleware* in der vorliegenden Ausarbeitung ist die *Microservice*-Architektur sinnvoll, weil viele Nutzer auf die implementierten Online-Dienste zugreifen und eine flexible Skalierung nötig ist. Dennoch ist im Kontext weiterer Software-Projekte stets eine sorgfältige Abwägung der Vor- und Nachteile einer *Microservice*-Architektur gegenüber anderen Lösungen erforderlich.

- **Frontend**: Der genutzte Technologie-Stack für die Entwicklung der Web-Anwendung setzt sich aus *Frameworks* und Bibliotheken von *Angular2* zusammen. *Angular2* wird stetig weiterentwickelt. Zu Beginn der Entwicklung des *Frontends* war die Version 2.2.3 die aktuellste. Zum Zeitpunkt der Fertigstellung war *Angular2* schon in der Version 2.4.6 verfügbar. Durch diese schnellen Entwicklungssprünge werden viele neue *Features* bereitgestellt, womit bestehende Implementierungen verbessert werden können. Die Version 2.3.0 beispielsweise eröffnete die Möglichkeit, neben Klassen auch *Components* zu vererben. Die einzelnen Online-Dienste werden in der Web-Anwendung in Tabs dargestellt. Viele der Eigenschaften in den *Components* der Online-Dienste wiederholen sich. Die Erstellung eines Basis-*Components*, welches vererbt wird, würde die Wartbarkeit und Wiederverwendbarkeit verbessern. Es ist also zu empfehlen, die *Milestones* der zukünftigen Veröffentlichungen von genutzten Technologien zu lesen, um bessere Designentscheidungen treffen zu können.

6.2. Entwicklungsstand

Die *Middleware* wurde vollständig nach den in Abschnitt 3.1.4 und 3.1.5 formulierten Anforderungen entwickelt. Die Kernfunktionen der sechs implementierten Online-Dienste können über die *Middleware* genutzt werden, indem die bereitgestellte *RESTful*-API verwendet wird.

Es wird also ein *single point of access* bereitgestellt. Die Lösung ist durch die Verwendung des HTTP-Protokolls auch plattformunabhängig nutzbar. Einige Abstriche treten bei der Vereinheitlichung der Fehlersemantik auf. Die Vereinheitlichung wird erreicht, indem die Fehlermeldung dem Benutzer mitteilt, warum die Anfrage gescheitert ist und welcher Dienst dafür verantwortlich ist. Allerdings sind noch nicht alle Fehlerfälle abgedeckt. Zusätzlich zu den formulierten Anforderungen wurde eine weitere Funktion für die *File-Storage*-Dienste entwickelt, der Transfer von Dateien aus einem *File-Storage*-Dienst in einen anderen. Dazu wurde ein weiterer *Microservice* implementiert. Dieser greift auf die Schnittstellen der bestehenden *File-Storage-Microservices* zu, um Dateien herunter- und hochzuladen. Dadurch entsteht ein neuer Dienst, der aus bereits bestehenden Diensten komponiert wurde. Ein Transfer-Dienst hätte auch über das *Frontend* implementiert werden können, indem die Schnittstellen der öffentlichen *RESTful*-API der *Middleware* genutzt werden. Dieser Ansatz wurde allerdings nicht gewählt, weil der Vorgang mehr Zeit in Anspruch nimmt. Die interne Implementierung erfordert nicht das Herunter- und Hochladen einer zu transferierenden Datei zu dem Client. Dieses Beispiel zeigt allerdings die Flexibilität, mit der die *Middleware* sowohl vom Benutzer als auch vom Entwickler genutzt werden kann.

Das beispielhafte *Frontend* wurde in Form einer Web-Anwendung realisiert und ermöglicht den Zugriff auf alle Funktionen, die in den Anforderungen beschrieben sind. Der Fokus wurde dabei auf die Bedienbarkeit gerichtet. Die Funktionen der Online-Dienste werden auf einer Oberfläche angeboten, um die gleichzeitige Verwendung mehrerer Dienste zu ermöglichen. Die Oberfläche wurde nach dem *Material Design* gestaltet, wodurch bedienbare Elemente und Bereiche hervorgehoben werden.

6.3. Stärken und Schwächen der Implementierung

6.3.1. Stärken

- Durch die Plattformunabhängigkeit kann die *Middleware* flexibel genutzt werden. Viele Entwickler können auf die *Middleware* zugreifen, ohne ihren Technologie-Stack anpassen zu müssen.
- Die *Middleware* erlaubt durch die *Microservice*-Architektur ein unabhängiges *Deployment*. Dadurch können einzelne *Microservices* aktualisiert und weiterentwickelt werden, ohne das gesamte Software-System neu ausliefern zu müssen. Außerdem führen uner-

wartete Fehler nicht zu einem Absturz des gesamten Systems. Lediglich der betroffene *Microservice* würde ausfallen.

- Das Skalieren von einzelnen *Microservices* ist ein weiterer Vorteil der aktuellen Implementierung der *Middleware*. Dies ermöglicht eine adaptive Anpassung bei steigendem Netzwerk-*Traffic*. GRPC bietet eine native Unterstützung für das *Load Balancing* an. In Kooperation mit bekannten Web-Servern wie zum Beispiel *NGINX* [91], welche das von GRPC genutzte HTTP/2 Protokoll unterstützen, kann die *Middleware* produktiv eingesetzt werden.

- Das *Frontend* bietet eine einheitliche Oberfläche an. Dienste der gleichen Kategorie können über eine identische Oberfläche vom Nutzer verwendet werden. Dadurch wird die *User Experience* gefördert. Des Weiteren kann der Nutzer Zeit sparen, wenn anstelle der einzelnen Anwendungen der Online-Dienste das Frontend verwendet wird. Der Zugriff auf die Dienste erfolgt über eine Web-Anwendung. Das Wechseln zwischen mehreren Geräten und Anwendungen, um verschiedene Dienste zu nutzen, entfällt. Das *Frontend* kann aufgrund der Komponenten-basierten Architektur leicht erweitert werden.

6.3.2. Schwächen

- Als eine Schwäche der Implementierung ist die Dienst-Auswahl zu identifzieren. Neben den drei *File-Storage*-Diensten, wurden auch zwei Versionsverwaltungsdienste und ein *Instant-Messaging*-Dienst implementiert (3.1.3). Die Kernfunktionen der Versionsverwaltungsdienste sind allerdings nicht für den Betrieb im *Internet-Browser* geeignet. Das Ändern eines *Repositories* (*commit*) beispielsweise wird nicht von den APIs der Versionsverwaltungsdienste *Github* und *Bitbucket* angeboten [92], [93]. Stattdessen werden die *commits* und weitere Kernfunktionen über *Git-Clients* [94] initiiert.

- Die Erweiterbarkeit der *Middleware* könnte noch gesteigert werden, vor allem bei der Erstellung von weiteren *Microservices*. Der Technologie-Stack von GRPC als Netzwerkkommunikationsprotokoll und das *Request-Package* von *Node.js* als HTTP-Client haben die meisten *Microservices* der *Middleware* gemein. Ein Code-Generator, der einen minimalen *Microservice* generiert, könnte die Erweiterbarkeit steigern. GRPC erstellt zwar dynamisch die *Stubs*, das *Mapping* von Funktionen auf die Schnittstelle könnte allerdings ein Code-Generator erleichtern.

- Für die *Middleware* wurde ein abstrakter *File-Storage*-Dienst entwickelt, welcher das Speichern von Dateien für Teams ermöglicht, unabhängig von den genutzten *File-Storage*-Diensten der einzelnen Teammitglieder (4.3). Allerdings verwenden Benutzer über die *Middleware* indirekt Tokens für die Autorisierung eines anderen Nutzers. Diese Designentscheidung wurde zugunsten der Dateneffizienz beim Umgang mit dem abstrakten *File-Storage*-Dienst getroffen (4.3). Die Anwendungsszenarien für solch einen Dienst sind vorhanden. Ob die Zielgruppe mit der Zugriffskontrolle einverstanden ist, kann in der vorliegenden Ausarbeitung nicht geklärt werden.

- Jede Anfrage an die *RESTful*-API der *Middleware* erfordert derzeit eine *Basic Authentication* (4.4.2), um die Anfrage dem Benutzer zuordnen zu können. Dadurch muss der Zugriff auf die *RESTful*-API stets über HTTPS erfolgen, wodurch die gesamte Anfrage, einschließlich des Benutzernamens und des Passworts, verschlüsselt wird. Durch das Anbieten einer Authentifizierung über OAUTH2 (4.4.1) wäre eine HTTPS-Verbindung nicht mehr zwingend notwendig, lediglich die initiale Authentifizierung müsste mit HTTPS verschlüsselt werden. Außerdem könnten Zugriffe auf die *Middleware* flexibler gestaltet werden, indem nur bestimmte Online-Dienste autorisiert werden.

6.4. Ausblick auf zukünftige Entwicklungen

Für die Entwicklungen im Bereich der Dienstkomposition von Online-Diensten sind vor allem die angebotenen APIs entscheidend. Von den sechs Online-Diensten, die in die *Middleware* integriert wurden, bieten fünf Dienste (ausgenommen *Owncloud*) eine offizielle API mit Dokumentation an. All diese APIs sind über HTTP erreichbar. Drei dieser APIs sind *REST*-konform aufgebaut, wohingegen die *Slack*- und *Dropbox*-APIs nach *Remote Procedure Calls* aufgebaut sind und nicht auf Ressourcen basieren. Obwohl mit dem Architekturparadigma von REST ein Standard vorhanden ist, wird dieser noch nicht von allen Anbietern von Online-Diensten genutzt. Noch dazu unterscheiden sich die jeweiligen Implementierungen. Vor allem die dritte Stufe des *Richardson Maturity Model* [58] einer *RESTful*-API, welche die Navigation zu weiterführenden Ressourcen angibt, auch *Hypermedia Controls* genannt, weist bei den jeweiligen Online-Diensten Unterschiede auf. Zum Beispiel gibt die API von *Bitbucket* alle weiterführenden Links in einem Array zurück [93], während die API von *Github* Links ohne übergeordnete Struktur als String zurückgibt. Entwickler müssen für den Zugriff auf jede API, selbst wenn sie ein Architekturparadigma wie REST nutzen, separaten Code schreiben. Interessant für zukünftige Entwicklungen könnte die Erstellung eines *Frameworks* sein, welches eine automatische Komposition von Online-Diensten ermöglicht. Voraussetzung hierfür ist die Vereinheitlichung

der *RESTful*-APIs. Eine zentrale Anforderung an die Entwicklung eines solchen *Frameworks* ist, dass die genutzten APIs vollständig *self discoverable* sind. Die Erstellung eines *Frameworks* eröffnet über diesen Untersuchungsrahmen hinaus vielfältige Anwendungsgebiete für Dienst-Kompositionen und bleibt insofern als ein vielversprechender Anhaltspunkt für zukünftige Untersuchungen festzuhalten.

A. Zielgruppenanalyse

A.1. Fragebogen

Studiengang: Fachsemester:

Web-Dienst	Wie häufig benutzen Sie den Dienst?	Wie zufrieden sind Sie mit dem Dienst?	Antrieb für die Nutzung?	Mit welchem Gerät benutzen Sie den Dienst? (Mehrfachauswahl)	Wie benutzen Sie den Dienst? (Mehrfachauswahl)	In welchem Umfeld benutzen Sie den Dienst? (Mehrfachauswahl)	Welche Zugriffsrechte würden Sie Drittanbietern am Dienst freigeben?
OwnCloud	○ Täglich ○ Wöchentlich ○ Seltener	Zufrieden ○○○○○ Unzufrieden	○ Lehrende ○ Kommilitonen ○ Eigener	○ PC ○ Smartphone/Tablet	○ Applikation ○ Browser ○ App(mobil)	○ Hochschule ○ Beruf ○ Privat	○ Keine ○ Innerhalb des Drittanbieters ○ Vollständig ○ Weiß nicht
EMIL	○ Täglich ○ Wöchentlich ○ Seltener	Zufrieden ○○○○○ Unzufrieden	○ Lehrende ○ Kommilitonen ○ Eigener	○ PC ○ Smartphone/Tablet	○ Applikation ○ Browser ○ App(mobil)	○ Hochschule ○ Beruf ○ Privat	○ Keine ○ Innerhalb des Drittanbieters ○ Vollständig ○ Weiß nicht
	○ Täglich ○ Wöchentlich ○ Seltener	Zufrieden ○○○○○ Unzufrieden	○ Lehrende ○ Kommilitonen ○ Eigener	○ PC ○ Smartphone/Tablet	○ Applikation ○ Browser ○ App(mobil)	○ Hochschule ○ Beruf ○ Privat	○ Keine ○ Innerhalb des Drittanbieters ○ Vollständig ○ Weiß nicht
	○ Täglich ○ Wöchentlich ○ Seltener	Zufrieden ○○○○○ Unzufrieden	○ Lehrende ○ Kommilitonen ○ Eigener	○ PC ○ Smartphone/Tablet	○ Applikation ○ Browser ○ App(mobil)	○ Hochschule ○ Beruf ○ Privat	○ Keine ○ Innerhalb des Drittanbieters ○ Vollständig ○ Weiß nicht
	○ Täglich ○ Wöchentlich ○ Seltener	Zufrieden ○○○○○ Unzufrieden	○ Lehrende ○ Kommilitonen ○ Eigener	○ PC ○ Smartphone/Tablet	○ Applikation ○ Browser ○ App(mobil)	○ Hochschule ○ Beruf ○ Privat	○ Keine ○ Innerhalb des Drittanbieters ○ Vollständig ○ Weiß nicht
	○ Täglich ○ Wöchentlich ○ Seltener	Zufrieden ○○○○○ Unzufrieden	○ Lehrende ○ Kommilitonen ○ Eigener	○ PC ○ Smartphone/Tablet	○ Applikation ○ Browser ○ App(mobil)	○ Hochschule ○ Beruf ○ Privat	○ Keine ○ Innerhalb des Drittanbieters ○ Vollständig ○ Weiß nicht
	○ Täglich ○ Wöchentlich ○ Seltener	Zufrieden ○○○○○ Unzufrieden	○ Lehrende ○ Kommilitonen ○ Eigener	○ PC ○ Smartphone/Tablet	○ Applikation ○ Browser ○ App(mobil)	○ Hochschule ○ Beruf ○ Privat	○ Keine ○ Innerhalb des Drittanbieters ○ Vollständig ○ Weiß nicht
	○ Täglich ○ Wöchentlich ○ Seltener	Zufrieden ○○○○○ Unzufrieden	○ Lehrende ○ Kommilitonen ○ Eigener	○ PC ○ Smartphone/Tablet	○ Applikation ○ Browser ○ App(mobil)	○ Hochschule ○ Beruf ○ Privat	○ Keine ○ Innerhalb des Drittanbieters ○ Vollständig ○ Weiß nicht
	○ Täglich ○ Wöchentlich ○ Seltener	Zufrieden ○○○○○ Unzufrieden	○ Lehrende ○ Kommilitonen ○ Eigener	○ PC ○ Smartphone/Tablet	○ Applikation ○ Browser ○ App(mobil)	○ Hochschule ○ Beruf ○ Privat	○ Keine ○ Innerhalb des Drittanbieters ○ Vollständig ○ Weiß nicht
	○ Täglich ○ Wöchentlich ○ Seltener	Zufrieden ○○○○○ Unzufrieden	○ Lehrende ○ Kommilitonen ○ Eigener	○ PC ○ Smartphone/Tablet	○ Applikation ○ Browser ○ App(mobil)	○ Hochschule ○ Beruf ○ Privat	○ Keine ○ Innerhalb des Drittanbieters ○ Vollständig ○ Weiß nicht

Abbildung A.1.: Fragebogen für die Zielgruppenanalyse

B. Frontend

B.1. Ergebnis

Link für das *Frontend*: https://transport-protocol.github.io/SPF.

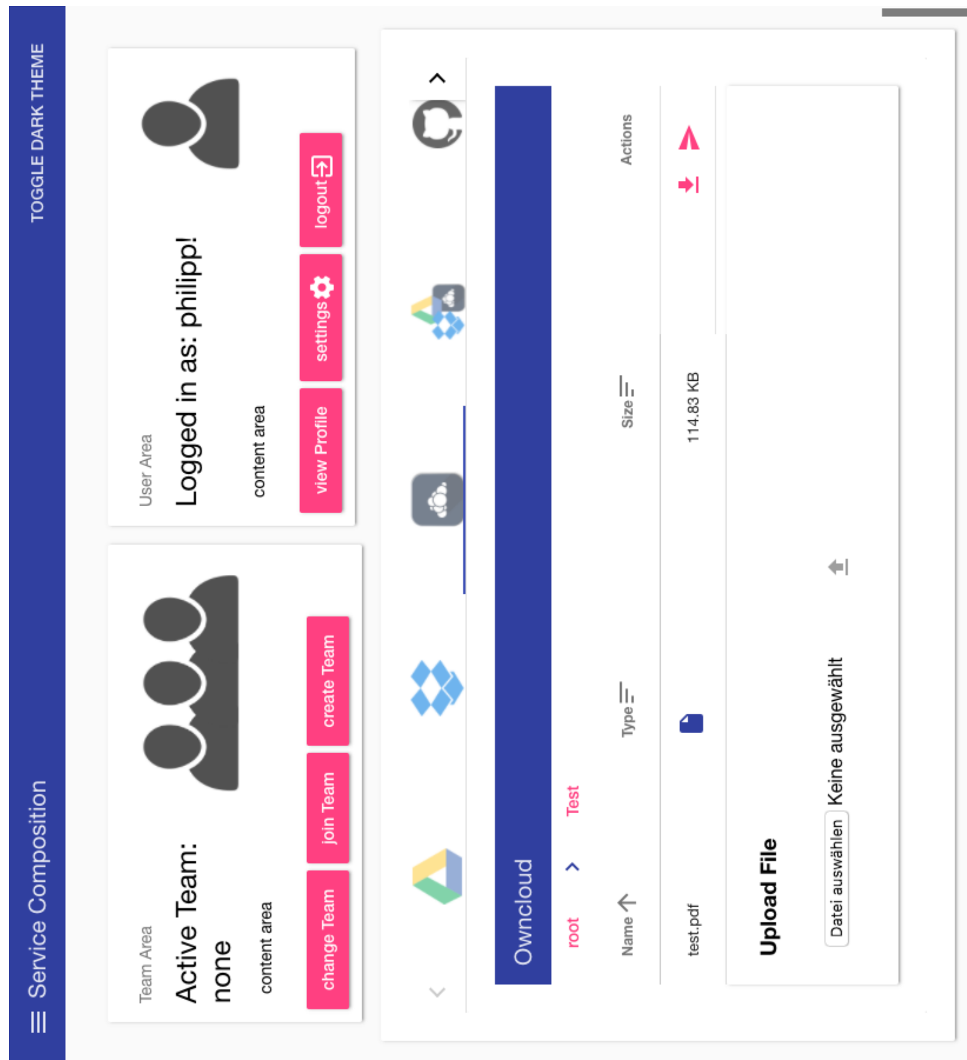


Abbildung B.1.: Screenshot des *Frontends*

Abkürzungsverzeichnis

API Application Programmable Interface

SMS Short Message Service

REST Representational State Transfer

HTTP Hypertext Transfer Protocol

HTTPS Hypertext Transfer Protocol Secure

HTTP/2 Hypertext Transfer Protocol Version 2

GRPC Google Remote Procedure Call

PC Personal Computer

NPM Node Package Manager

RPC Remote Procedure Call

XML Extensible Markup Language

JSON JavaScript Object Notation

VM Virtual Machine

BSON Binary JSON

SaaS Software as a Service

URL Uniform Resource Locator

URI Uniform Resource Identifier

OAUTH2 Open Authentication 2

HATEOAS Hypertext As The Engine Of Application State

SPA Single Page Application

AJAX Asynchronous JavaScript And XML

DOM Document Object Model

HTML Hypertext Markup Language

CSS Cascading Style Sheets

UI User Interface

Literaturverzeichnis

[1] L. Wang, J. Tao, M. Kunze, A. C. Castellanos, D. Kramer, and W. Karl, “Scientific cloud computing: Early definition and experience.” in *HPCC*, vol. 8, 2008, p. 3.

[2] B. A. Nardi, S. Whittaker, and E. Bradner, “Interaction and outeraction: instant messaging in action,” in *Proceedings of the 2000 ACM conference on Computer supported cooperative work*. ACM, 2000, p. 80.

[3] B. Collins-Sussman, B. Fitzpatrick, and M. Pilato, *Version control with subversion.* Ö'Reilly Media, Inc.", 2004.

[4] M.-S. E. Scale, “Cloud computing and collaboration,” *Library Hi Tech News*, vol. 26, no. 9, pp. 10–13, 2009.

[5] P. Bianco, R. Kotermanski, and P. F. Merson, “Evaluating a service-oriented architecture,” 2007.

[6] S. Jones, “Toward an acceptable definition of service [service-oriented architecture],” *IEEE software*, vol. 22, no. 3, pp. 87–93, 2005.

[7] ITWissen. (2016) Definition online. [Online]. Available: http://www.itwissen.info/definition/lexikon/Online-Betrieb-OL-online-processing.html

[8] M. Reichert and D. Stoll, “Komposition, choreograhpie und orchestrierung von web services: Ein überblick,” in *EMISA Forum*, vol. 24, no. 2, 2004, pp. 21–32.

[9] G. Kapitsaki, D. Kateros, I. Foukarakis, G. Prezerakos, D. Kaklamani, and I. Venieris, “Service composition: State of the art and future challenges,” in *2007 16th IST Mobile and Wireless Communications Summit*. IEEE, 2007, pp. 1–5.

[10] Informa. (2016) Ott messaging traffic will be twice volume of p2p sms traffic this year. [Online]. Available: http://informa.com/media/press-releases-news/latest-news/ott-messaging-traffic-will-be-twice-volume-of-p2p-sms-traffic-this-year/

[11] C. Rodríguez-Bustos and J. Aponte, "How distributed version control systems impact open source software projects," in *Proceedings of the 9th IEEE Working Conference on Mining Software Repositories.* IEEE Press, 2012, pp. 36–39.

[12] M. Vallance, P. A. Towndrow, and C. Wiz, "Conditions for successful online document collaboration," *TechTrends*, vol. 54, no. 1, p. 22, 2010.

[13] C. M. u. B. Hilkert, Daniel und Wolf, "Das as-a-service-paradigma: Treiber von veraenderungen in der software-industrie?" in *Software-as-a-Service.* Springer, 2010, pp. 57–74.

[14] D. Bakken, "Middleware," *Encyclopedia of Distributed Computing*, vol. 11, 2001.

[15] P. Fischer and P. Hofer, *Lexikon der Informatik.* Springer-Verlag, 2010.

[16] P. Stein, "Forschungsdesigns für die quantitative sozialforschung," in *Handbuch Methoden der empirischen Sozialforschung.* Springer, 2014.

[17] H. Hamburg. (2016) Studierende der haw hamburg sommersemester 2016. [Online]. Available: https://www.haw-hamburg.de/fileadmin/user_upload/Presse_und_Kommunikation/Downloads/C_1_Web_SoSe_16_Studierende_gesamt_20160617.pdf

[18] B. Rasch, M. Friese, W. Hofmann, and E. Naumann, *Quantitative methoden.* Springer, 2004.

[19] Doodle. (2016) Doodle. [Online]. Available: http://doodle.com/de/

[20] H. Röder, S. Franke, C. Müller, and D. Przybylski, "Ein kriterienkatalog zur bewertung von anforderungsspezifikationen," *Softwaretechnik-Trends*, vol. 29, no. 4, 2009.

[21] M. ten Hompel and T. Schmidt, "Softwareengineering," *Warehouse Management: Organisation und Steuerung von Lager-und Kommissioniersystemen*, pp. 221–254, 2008.

[22] Pivotal. (2016) Rabbitmq. [Online]. Available: https://www.rabbitmq.com/

[23] iMatix Corporation. (2016) Zeromq. [Online]. Available: http://zeromq.org/

[24] M. Fowler and J. Lewis, "Microservices: Nur ein weiteres konzept in der softwarearchitektur oder mehr," *Objektspektrum*, vol. 1, no. 2015, pp. 14–20, 2015.

[25] E. Wolff, "Microservices: Grundlagen flexibler softwarearchitekturen," *Dpunkt. Verlag GmbH. ISBN: 9783864903137*, 2015.

[26] M. Fowler, “Public versus published interfaces,” *IEEE Software*, vol. 19, no. 2, pp. 18–19, 2002.

[27] G. Inc. (2016) A high performance, open-source universal rpc framework. [Online]. Available: http://www.grpc.io

[28] S. Tilkov and S. Vinoski, “Node. js: Using javascript to build high-performance network programs,” *IEEE Internet Computing*, vol. 14, no. 6, p. 80, 2010.

[29] S. Pasquali, *Mastering Node. js.* Packt Publishing Ltd, 2013.

[30] P. Teixeira, *Professional Node. js: Building Javascript based scalable software.* John Wiley & Sons, 2012.

[31] I. npm. (2016) npmjs. [Online]. Available: https://www.npmjs.com/

[32] K. Lei, Y. Ma, and Z. Tan, “Performance comparison and evaluation of web development technologies in php, python, and node. js,” in *Computational Science and Engineering (CSE), 2014 IEEE 17th International Conference on.* IEEE, 2014, pp. 661–668.

[33] I. npm. (2016) Using a package.json. [Online]. Available: https://docs.npmjs.com/getting-started/using-a-package.json

[34] V. Karpov. (2016) Mongoose mongodb odm. [Online]. Available: https://www.npmjs.com/package/mongoose

[35] S. Velichkov. (2016) Simplified http request client. [Online]. Available: https://www.npmjs.com/package/request

[36] J. Cruger. (2016) Hierarchical node.js configuration with files, environment variables, command-line arguments, and atomic object merging. [Online]. Available: https://www.npmjs.com/package/nconf

[37] C. Robbins. (2016) A multi-transport async logging library for node.js. [Online]. Available: https://www.npmjs.com/package/winston

[38] A. Strzelewicz. (2017) Production process manager for node.js applications with a built-in load balancer. [Online]. Available: https://www.npmjs.com/package/pm2

[39] M. Alvermann, “EinfÃ¼hrung in mongodb,” 2011.

[40] C. Győrödi, R. Győrödi, G. Pecherle, and A. Olah, “A comparative study: Mongodb vs. mysql,” in *Engineering of Modern Electric Systems (EMES), 2015 13th International Conference on.* IEEE, 2015, pp. 1–6.

[41] M. Belshe, M. Thomson, and R. Peon, “Hypertext transfer protocol version 2 (http/2),” 2015.

[42] G. Inc. (2016) Protocol buffers are a language-neutral, platform-neutral extensible mechanism for serializing structured data. [Online]. Available: https://developers.google.com/protocol-buffers/

[43] N. Gligorić, I. Dejanović, and S. Krčo, “Performance evaluation of compact binary xml representation for constrained devices,” in *2011 International Conference on Distributed Computing in Sensor Systems and Workshops (DCOSS).* IEEE, 2011, pp. 1–5.

[44] R. Battle and E. Benson, “Bridging the semantic web and web 2.0 with representational state transfer (rest),” *Web Semantics: Science, Services and Agents on the World Wide Web*, vol. 6, no. 1, pp. 61–69, 2008.

[45] T. Bray, “The javascript object notation (json) data interchange format,” 2014.

[46] T. Bray, J. Paoli, C. M. Sperberg-McQueen, E. Maler, and F. Yergeau, “Extensible markup language (xml),” *World Wide Web Consortium Recommendation REC-xml-19980210. http://www. w3. org/TR/1998/REC-xml-19980210*, vol. 16, p. 16, 1998.

[47] D. Crockford, “The application/json media type for javascript object notation (json),” 2006.

[48] B. Gil and P. Trezentos, “Impacts of data interchange formats on energy consumption and performance in smartphones,” in *Proceedings of the 2011 workshop on open source and design of communication.* ACM, 2011, pp. 1–6.

[49] T. Hughes-Croucher and M. Wilson, *Einfuehrung in Node. js.* O’Reilly Germany, 2012.

[50] E. Standard. (2017) Ecmascript 2017 languagae specification. [Online]. Available: https://tc39.github.io/ecma262/#sec-promise-objects

[51] M. D. Network. (2017) Promise. [Online]. Available: https://developer.mozilla.org/en/docs/Web/JavaScript/Reference/Global_Objects/Promise

[52] G. Developers. (2017) Work with folders. [Online]. Available: https://developers.google.com/drive/v3/web/folder

[53] Google. (2017) Google groups. [Online]. Available: https://groups.google.com/forum/#!topic/grpc-io/UiCvZRS1B9I/discussion

[54] R. Boyd, *Getting started with OAuth 2.0.* Ö'Reilly Media, Inc.", 2012.

[55] D. Hardt, "The oauth 2.0 authorization framework," 2012.

[56] J. Franks, P. Hallam-Baker, J. Hostetler, S. Lawrence, P. Leach, A. Luotonen, and L. Stewart, "Http authentication: Basic and digest access authentication," Tech. Rep., 1999.

[57] R. Fielding, J. Gettys, J. Mogul, H. Frystyk, L. Masinter, P. Leach, and T. Berners-Lee, "Hypertext transfer protocol–http/1.1," Tech. Rep., 1999.

[58] M. Fowler, "Richardson maturity model: steps toward the glory of rest," *Online at http://martinfowler. com/articles/richardsonMaturityModel. html*, 2010.

[59] M. S. Mikowski and J. C. Powell, "Single page web applications," *B and W*, 2013.

[60] J. J. Garrett *et al.*, "Ajax: A new approach to web applications," 2005.

[61] Facebook. (2017) facebook/react. [Online]. Available: https://github.com/facebook/react

[62] Google. (2017) angular/angular. [Online]. Available: https://github.com/angular/angular

[63] Facebook. (2017) A javascript library for building user interfaces. [Online]. Available: https://facebook.github.io/react/

[64] P. Speaker-Hunt, P. Speaker-O'Shannessy, D. Speaker-Smith, and T. Speaker-Coatta, "React: Facebook's functional turn on writing javascript," *Queue*, vol. 14, no. 4, p. 40, 2016.

[65] Google. (2017) Angular. [Online]. Available: https://angular.io/

[66] ——. (2017) Angular quickstart typescript. [Online]. Available: https://angular.io/docs/ts/latest/quickstart.html

[67] Microsoft. (2017) Typescript. [Online]. Available: https://www.typescriptlang.org/

[68] Google. (2017) Angular-cli. [Online]. Available: https://cli.angular.io/

[69] ——. (2017) Protractor - changelog. [Online]. Available: https://github.com/angular/protractor/blob/master/CHANGELOG.md

[70] ——. (2017) Protractor - end-to-end testing for angularjs. [Online]. Available: http://www.protractortest.org/#/

[71] ——. (2017) Angular material. [Online]. Available: https://material.angular.io/

[72] ——. (2017) Angular modules. [Online]. Available: https://angular.io/docs/ts/latest/guide/ngmodule.html#

[73] ——. (2017) Lifecycle hooks. [Online]. Available: https://angular.io/docs/ts/latest/guide/lifecycle-hooks.html

[74] ——. (2017) Architecture overview. [Online]. Available: https://angular.io/docs/ts/latest/guide/architecture.html#

[75] ——. (2017) Angular2 dependency injection. [Online]. Available: https://angular.io/docs/ts/latest/guide/architecture.html#!#dependency-injection

[76] ——. (2017) Angular2 data binding. [Online]. Available: https://angular.io/docs/ts/latest/guide/architecture.html#!#data-binding

[77] ——. (2017) Angular2 jit aot. [Online]. Available: https://angular.io/docs/ts/latest/cookbook/aot-compiler.html#!#aot-jit

[78] ——. (2017) Angular lazy loading. [Online]. Available: https://angular.io/docs/ts/latest/guide/deployment.html#!#lazy-loading

[79] ——. (2017) Angular2 router. [Online]. Available: https://angular.io/docs/ts/latest/guide/router.html

[80] ——. (2017) Component - tabs. [Online]. Available: https://material.angular.io/components/component/tabs

[81] ——. (2017) Angular2 material component progress spinner. [Online]. Available: https://material.angular.io/components/component/progress-spinner

[82] ——. (2017) Angular2 animations. [Online]. Available: https://angular.io/docs/ts/latest/guide/animations.html

[83] F. Lauc. (2017) An easy to use notification library for angular 2. [Online]. Available: https://www.npmjs.com/package/angular2-notifications

[84] Google. (2017) Angular2 http client. [Online]. Available: https://angular.io/docs/ts/latest/api/http/index/Http-class.html

[85] M. D. Network. (2017) Xmlhttprequest. [Online]. Available: https://developer.mozilla.org/de/docs/Web/API/XMLHttpRequest

[86] Google. (2017) Xhrbackend. [Online]. Available: https://angular.io/docs/ts/latest/api/http/index/XHRBackend-class.html

[87] ReactiveX. (2017) Observable. [Online]. Available: http://reactivex.io/documentation/observable.html

[88] Google. (2017) Protractor - how it works. [Online]. Available: http://www.protractortest.org/#/infrastructure

[89] ThoughtWorks. (2017) Seleniumhq browser automation. [Online]. Available: http://www.seleniumhq.org/

[90] Google. (2017) Protractor style guide. [Online]. Available: http://www.protractortest.org/#/style-guide

[91] I. Nginx. (2017) nginx. [Online]. Available: https://nginx.org/en/

[92] I. GitHub. (2017) Github api. [Online]. Available: https://developer.github.com/v3/

[93] Atlassian. (2017) Bitbucket api. [Online]. Available: https://developer.atlassian.com/bitbucket/api/2/reference/resource/

[94] L. T. Junio C. Hamano, Shawn O. Pearce. (2017) git –fast-version-control. [Online]. Available: https://git-scm.com/

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

Hamburg, 27. Februar 2017 Philipp Prögel