HAW HAMBURG

Bachelorarbeit

Andreas Jäckel

Konzeption und Umsetzung einer Portalseite und Integrationsumgebung für die HAWAI Microservices

Fakultät Technik und Informatik
Department Informatik

Faculty of Computer Science and Engineering
Department Computer Science

Andreas Jäckel

Konzeption und Umsetzung einer Portalseite und Integrationsumgebung für die HAWAI Microservices

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.-Ing. Stefan Sarstedt
Zweitgutachter: Prof. Dr. Ulrike Steffens

Eingereicht am: 02. Oktober 2018

Andreas Jäckel

Thema der Arbeit

Konzeption und Umsetzung einer Portalseite und Integrationsumgebung für die HAWAI Microservices

Stichworte

Microservices, Webservices, OAuth2, OpenId Connect, Dashboard

Kurzzusammenfassung

In der vorliegenden Arbeit wird die Herleitung und Implementierung einer Portalseite und Integrationsumgebung für Microservices im Zusammenhang mit dem HAWAI-Projekt aufgezeigt. Hierfür werden die aktuellen Standards der Kommunikation und Authentifikation von Microservices betrachtet. Auf diesen Grundlagen aufbauend, wird auf die Anforderungsspezifikation und den Entwurf für eine Portalseite eingegangen. Abschließend wird eine mögliche Implementierung aufgezeigt und getestet.

Andreas Jäckel

Title of Thesis

Conception and implementation of a portal page and integration environment for HAWAI Microservices

Keywords

Microservices, Webservices, OAuth2, OpenId Connect, Dashboard

Abstract

This thesis shows the derivation and implementation of a portal page and intergration environment for microservices in connection with the HAWAI-Project. For this purpose, the current standards of communication and authentication of microservices are considered. Based on these basics, the requirement specification and the design for a portal page are discussed. Finally, a possible implementation is shown and tested

Inhaltsverzeichnis

Abbildungsverzeichnis **vi**

1 Einleitung **1**
1.1 Problemstellung . . . 1
1.2 Zielsetzung . . . 1
1.3 Gliederung der Arbeit . . . 1

2 Grundlagen **3**
2.1 Microservices . . . 3
2.1.1 Vorteile von Microservices . . . 4
2.1.2 Nachteile von Microservices . . . 4
2.1.3 Kommunikation . . . 5
2.2 Webservices . . . 6
2.3 Verteilte Systeme . . . 7
2.4 Hypertext Transfer Protocol . . . 8
2.4.1 Request . . . 10
2.4.2 Response . . . 11
2.4.3 HTTP-Authentifizierung . . . 12
2.5 Docker-Container . . . 12
2.5.1 Architektur von Docker . . . 13
2.5.2 Dockerfile . . . 14
2.5.3 Docker Compose . . . 15

3 Anforderungsspezifikation **16**
3.1 Das HAWAI Projekt . . . 16
3.2 Anforderungen an die Integrationsumgebung . . . 17
3.2.1 Funktionale Anforderungen . . . 17
3.2.2 Nicht-funktionale Anforderungen . . . 24

3.3 Realisierbarkeitsanalyse . . . 27
3.3.1 Hardware-Analyse . . . 27
3.3.2 Software-Analyse . . . 27
3.4 Fazit . . . 28

4 Systementwurf **29**
4.1 Gliederung des Systems . . . 29
4.1.1 Aufteilung der Webservices . . . 29
4.1.2 Übersicht über das Gesamtbild . . . 31
4.2 Auth-Server . . . 33
4.2.1 Das OAuth2-Protokoll . . . 33
4.2.2 OpenId Connect . . . 37
4.2.3 Identity-Server . . . 38
4.2.3.1 Aufbau . . . 38
4.2.3.2 Entwicklung der Endpunkte . . . 40
4.2.3.3 Identity-Middleware . . . 43
4.3 Auth-Persistence . . . 44
4.4 Dashboard . . . 45
4.4.1 Unterschied SPA und MPA . . . 46
4.4.2 Angular . . . 48
4.4.3 Umsetzung der Anforderungen . . . 50
4.5 File-Manager . . . 51
4.5.1 Representational State Transfer (REST) . . . 52
4.5.2 Simple Object Access Protocol (SOAP) . . . 53
4.5.3 Unterschied REST und SOAP . . . 53
4.5.4 Bestimmung des API-Ansatzes und der Entwicklungsumgebung . . . 54
4.5.4.1 Entwicklungsumgebung . . . 54
4.5.5 Entwurf der API . . . 55
4.5.5.1 Request . . . 55
4.5.5.2 Response . . . 57
4.5.5.3 Authentifizierung . . . 57
4.5.5.4 Cross Origin Resource Sharing . . . 57
4.6 File-Persistence . . . 58
4.7 Interprozesskommunikation . . . 59
4.8 Netflix Open Source Software (OSS) . . . 62
4.9 Dockerize . . . 62

5 Implementierung des Systems **64**
5.1 Auth-Server . . . 64
5.1.1 Startup.cs . . . 65
5.1.2 Config.cs . . . 66
5.1.3 Weitere Klassen . . . 67
5.2 Auth-Persistence . . . 67
5.3 Dashboard . . . 68
5.3.1 auth.config.ts . . . 68
5.3.2 app.component.ts . . . 69
5.3.3 Services . . . 70
5.3.4 Home-Komponente . . . 71
5.4 File-Manager . . . 71
5.4.1 Config . . . 72
5.4.2 Controllers . . . 73
5.4.3 Services . . . 74
5.4.4 Models . . . 74
5.4.5 Middleware . . . 75
5.5 File-Persistence . . . 75
5.6 Docker . . . 76

6 Evaluierung **78**
6.1 Komponententest . . . 78
6.2 Integrationstest . . . 81
6.3 Systemtest . . . 82
6.3.1 Funktionale Anforderungen . . . 83
6.3.2 Nicht-funktionale Anforderungen . . . 84
6.4 Veröffentlichung . . . 85
6.5 Erkenntnisse der Evaluierung . . . 86

7 Schluss **87**
7.1 Zusammenfassung . . . 87
7.2 Ausblick . . . 88

Selbstständigkeitserklärung **92**

Abbildungsverzeichnis

2.1 Übersicht der Modularisierung durch Microservices 3
2.2 Integration von Microservices auf unterschiedlicher Ebene [Wolff, 2016, S. 149] 5
2.3 Kommunikation per HTTP 8
2.4 Aufbau und Unterschied eines Docker Containers und einer VM [Docker, 2018] 12
2.5 Die wichtigsten Docker-Komponenten [Mouat, 2016, S. 35] 14

4.1 Übersicht der Komponenten und ihrer Schnittstellen 31
4.2 Vereinfachter allgemeiner Ablauf einer Authorization-Anfrage 34
4.3 Vereinfachter allgemeiner Ablauf einer Authorization-Anfrage 35
4.4 Übersicht des Zusammenspiels von Identity-Server und eine ASP.NET Core-Anwendung aus [Brock und Baier, 2016, The Big Picture]. 39
4.5 Übersicht der wichtigsten Blöcke einer Angular Anwendung [Angular Architecure, 2018] 50
4.6 Entwurf der Dashboard-Portalseite 51
4.7 Übersicht der Datenbank Architektur 58
4.8 Sequenzdiagramm der Interprozesskommunikation 60

5.1 Ausschnitt des Aufbaus des Dashboards 69
5.2 Übersicht über den Aufbau des File-Managers 71

6.1 Response-Nachricht auf eine nicht definierte URI 79
6.2 Response-Nachricht auf einen unautorisierten Request 80
6.3 Liste der Dateien vor der Aktion des Hochladens 80
6.4 Liste der Dateien nach der Aktion des Hochladens 81

Listings

2.1 Beispielhafter POST-Request 10
2.2 Beispielhafter Darstellung einer Response-Nachricht 11

4.1 Beispiel für einen Request und Response Verlauf für einen Endpunkt . . . 42
4.2 Ansicht einer "Hallo Welt"Komponente 48

5.1 Auszug aus der Startup.cs Klasse 65
5.2 API-Konfiguration in der Config.cs 66
5.3 fileServer.services.ts: Request für eine Liste der verfügbaren Dateien 70
5.4 Auszug der URIs-Deklaration 73
5.5 Many-To-Many-Beziehung in der »Clients.php«-Datei 75
5.6 Implementierung der »files«-Datenbank 76
5.7 Auszug aus der Docker-Compose 77

1 Einleitung

1.1 Problemstellung

In der immer stärker ausgeprägten digitalisierten Welt wachsen Softwarelösungen von Unternehmen immer mehr an. Um diesem entgegenzuwirken und eine beherrschbare Übersicht und Wartbarkeit komplexer Softwarelösungen zu erlangen, wird immer mehr auf Services gesetzt. Dies trifft auch auf das HAWAI-Projekt [Hawai, 2018] zu, welches durch die HAW Hamburg Fakultät TI gefördert wird. Das Ziel des Projektes ist es, für Unternehmen typische IT-Anwendungslandschaften zu analysieren und in einer virtuellen Laborumgebung nachzubauen. Durch die vielen entstandenen und entstehenden Services ist die Notwendigkeit einer Portalseite und Integrationsumgebung gewachsen, die als Kommunikationsplattform zwischen den einzelnen Services dienen soll.

1.2 Zielsetzung

Ziel dieser Arbeit ist es, eine Portalseite und Integrationsumgebung für Services des HAWAI-Projektes zu erstellen. Die Aufgabe der Portalseite und Integrationsumgebung, soll die Unterstützung der Kommunikation zwischen den einzelnen Services sein, eine Authentifikation für den Benutzer und den einzelnen Services bieten und außerdem noch über eine UI zur Übersicht aller verfügbaren Services und Dateien verfügen.

1.3 Gliederung der Arbeit

Die Gliederung der Arbeit, wird sich an dem empfohlenen Aufbau von [Krypczyk und Bochkor, 2018] halten. Das erste Kapitel umfasst die Einleitung, in der die Problemstellung und die Zielsetzung näher erläutert werden. Das zweite Kapitel gibt eine allgemeine

Einführung in die Grundlagen, der wichtigsten Verfahren moderner Webanwendungen. Im dritten Kapitel werden die Anforderungen für die Portalseite und Integrationsumgebung spezifiziert. Im anschließenden Kapitel wird der genaue Systementwurf dargestellt und eine Erläuterung der einzelnen Komponenten, die aus den Anforderungen des vorangegangenen kapitels entstanden sind, gegeben. Im fünften Kapitel wird beschrieben, wie die Umsetzung der verschiedenen Systemkomponente realisiert wurde. Im sechsten Kapitel wird eine Evaluierung der implementierten Portalseite und Integrationsumgebung durchgeführt.

Das letzte Kapitel enthält eine Zusammenfassung über das Erreichte in dieser Arbeit. Des Weiteren wird ein Ausblick auf Erweiterungen der Portalseite und Integrationsumgebung gegeben.

2 Grundlagen

Um die Grundlagen für diese Arbeit zu legen, wird in diesem Kapitel zunächst auf die allgemeine Beschreibung der wichtigsten Begriffe und Verfahren der modernen Software-Entwicklung im Hinblick auf die Webentwicklung eingegangen.

2.1 Microservices

Bei großen Software-Projekten ist es oft schwierig die Übersicht über die einzelnen Funktionen zu behalten. Ebenso leidet die Wartbarkeit bzw. die Erneuerung von beliebigen Software-Funktionen. Eine Lösung ist die Modularisierung von Funktionen. Das heißt, es werden große System in kleinere Module zerlegt, um die Wartbarkeit zu verbessern. Microservices greift diesen Ansatz auf und geht noch einen Schritt weiter. Die kleineren Module werden in eigenständige lauffähige Prozesse gekapselt. So sind zum Beispiel bei einem E-Shop der Vorgang einer Bestellung und die Produkte in jeweils einen Prozess unterteilt und können unabhängig voneinander gewartet oder verbessert werden. Siehe Abbildung (2.1)

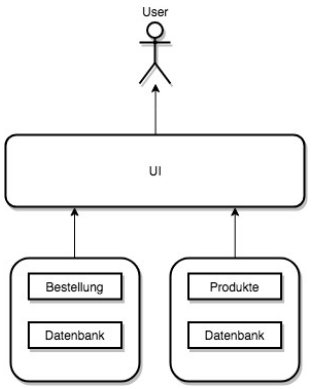

Abbildung 2.1: Übersicht der Modularisierung durch Microservices

2.1.1 Vorteile von Microservices

In der Studie von Davide Taibi, Valentina Lenarduzzi, Claus Pahl und Andrea Janes [Taibi u. a., 2017] konnten folgende Vorteile von Microservices identifiziert werden.

leichte Ersetzbarkeit Da Microservices unabhängig voneinander laufen, können die einzelnen Microservice leicht ersetzt bzw. ausgetauscht werden. Der neue Service muss nur die gleichen Schnittstellen wie der alte Service anbieten.

parallele Entwicklung Da Micrsoservices unabhängig voneinander in Produktion gebracht werden können, können mehrere Teams ebenso unabhängig voneinander an verschiedenen Teilen des Projekts arbeiten.

nachhaltige Software-Entwicklung Durch ihre im Verhältnis geringe Größe, bleiben Microservices auch zu einem späteren Zeitpunkt wartbar.

Robustheit Microservices-Systeme können gegen den Ausfall einzelner Microservices abgesichert werden.

Technologiefreiheit Weil Microservices gekapselt laufen, können die einzelnen Microservices in unterschiedlichen Technologien implementiert werden.

2.1.2 Nachteile von Microservices

Eberhard Wolff spricht in seinem Buch [Wolff, 2016] auch Nachteile der Microservice-Architektur an:

versteckte Beziehungen Die Hauptarchitektur eines Systems besteht aus den Beziehungen der einzelnen Microservices. Da diese Beziehungen über eine API abgewickelt werden, ist auf den ersten Blick meist nicht klar, welcher Microservice welchen anderen aufruft.

Latenz Die Kommunikation der einzelnen Microservices findet über das Netzwerk statt. In einem Netzwerk kann es zu Latenzen kommen und diese beeinträchtigen die Reaktionszeit eines einzelnen Microservice.

Monitoring Das Monitoring der Microservices wird deutlich komplexer, da eine Vielzahl von Microservices in einem Projekt beteiligt sind.

Refactoring Sollen Funktionalitäten zwischen Microservices verschoben werden, ist die Umsetzung nur sehr schwer durchführbar.

2.1.3 Kommunikation

Die Kommunikation von Microservices kann auf unterschiedlichen Ebenen stattfinden (siehe Abb. 2.2)

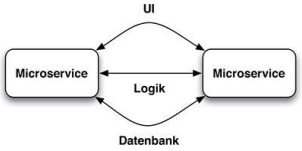

Abbildung 2.2: Integration von Microservices auf unterschiedlicher Ebene [Wolff, 2016, S. 149]

Auf der UI-Ebene gibt es verschiedene Ansätze die Kommunikation von Microservices umzusetzen. In der Regel sollten Microservices ihre eigene UI bereitstellen. Die UI wird im einfachsten Fall als Single-Page-Application (SPA) umgesetzt. Durch einen einfachen Hyperlink können dann andere Microservices aufgerufen werden. Ein anderer Ansatz wäre nur eine SPA für alle Microservices, wobei die einzelnen Microservices weiterhin strikt getrennt bleiben. Ebenso könnte die UI im einfachen HTML umgesetzt werden.

Die Kommunikation auf der Logikebene kann durch Representational State Transfer (REST), Service Oriented Architecture (SOAP) und Messaging umgesetzt werden. Beim Messaging werden Nachrichten zwischen den einzelnen Microservices verschickt. Ein Vorteil ist bei Messaging, dass selbst bei einem Ausfall einzelner Microservices die Nachrichten nicht verloren gehen und später erneut versendet werden können. Da dieser Ansatz in der vorliegenden Arbeit nicht umgesetzt wurde, wird hier auch nicht näher drauf eingegangen. Für weiterführende Informationen kann das Buch von Eberhard Wolff [Wolff, 2016] herangezogen werden. Ein Überblick über REST ist unter Punkt (4.5.1) und ein Überblick über SOA(P) ist unter Punkt (4.5.2) zu finden.

Auf der Datenbankebene sind die Microservices weiterhin voneinander getrennt, können allerdings gemeinsam eine Datenbank nutzen und so auf die Datenbanksätze zugreifen. Diese Art der Kommunikation ist allerdings nicht zu empfehlen, da es bei Änderungen der Datensätze zu Fehlern bei anderen Microservices kommen kann. Es sollte daher für jeden Microservice eine eigene Datenverwaltung zur Verfügung stehen. Im besten Fall

wird der Polyglot-Presistence-Ansatz verfolgt. Die Polyglot-Persistence verfolgt das Ziel, verschiedene Datenbanktechnologien für unterschiedliche Anforderungen der Datenbanken zu verwenden.

2.2 Webservices

Webservices sind für Rechnernetze konzipiert worden, um Maschine-zu-Maschine Kommunikation zu ermöglichen. Die Basis der Kommunikation ist das HTTP-Protokoll. Durch Webservices ist es möglich vorhandene oder selbst entwickelte Schnittstellen über das Internet anzusprechen und Informationen oder Daten auszutauschen. Microservices können mit Webservices implementiert werden und sind somit für den Informationsaustausch über das Internet gut ansprechbar.

Wenn man einen bestehenden Webservice, wie zum Beispiel Google-Maps, ansprechen möchte, muss man der Schnittstelle des Google-Maps-Webservices mitteilen, welche Information man zur Verfügung gestellt bekommen möchte. Dies geschieht über das HTTP-Protokoll, welches ein bestimmtes Format an den Google-Maps-Webservice sendet. Dieses Format hängt davon ab, wie der Google-Maps-Webservice seine Schnittstelle implementiert hat. Hier wären zu nennen REST, SOAP, Messaging. Nähere Informationen zu diesen Verfahren können im Kapitel (4.5) gefunden werden.

Durch die Implementierung der Schnittstellen durch REST, SOAP oder Messaging, bleiben die separaten Webservices frei in der Wahl ihrer Programmiersprache. So kann der eigene Webservice in Java geschrieben sein und der Google-Maps-Webservice in PHP. Obwohl Java und PHP sich nicht gegenseitig aufrufen können, können sie über diese Schnittstellen kommunizieren.

Damit der eigene Webservice weiß, welche Schnittstellen angesprochen werden können, kann in den meisten Fällen eine XML Datei oder eine JSON-Nachricht von einem anderen Webservice angefordert werden. In dieser Nachricht oder Datei stehen dann die Informationen über die Schnittstellen eines Webservices, die angesprochen werden können.

2.3 Verteilte Systeme

Verteilte System bilden die Grundlage von Microservices. Eine klare Definition von verteilten System gibt es nicht. Einen Ansatz für eine lockere Definition gibt Andrew S. Tanenbaum in seinem Buch Distributed Systems: Principles and Paradigms [Tanenbaum und van Steen, 2002].

> *A distributed system is a collection of independent computer that appears to its users as a single coherent system.*
>
> *(Andrew S. Tanenbaum [Tanenbaum und van Steen, 2002, S. 2])*

Aus dieser Definition kann man zwei Aspekte erkennen. Zu einem sind die Computer (maschinen-) unabhängig voneinander und die Benutzer bekommen von der Kommunikation zwischen Ihnen nichts mit. Zum anderen denken die Benutzer, sie verwenden nur ein System. Beide Aspekete sind essentiell für verteilte Systeme. Ebenfalls sollte es für verteilte System relativ einfach sein ihre Funktionalitäten zu erweitern.

Das Hauptziel der verteilten Systeme ist es, den Benutzern Zugriff auf Ressourcen zu ermöglichen. Die Ressourcen befinden sich in der Regel, an anderer Stelle im Netzwerk und können mit anderen Benutzern in einen kontrollierten Rahmen geteilt werden. Dabei ist es wichtig, dass der Benutzer nicht mitbekommt, dass die Ressourcen auf mehrere Komponenten im Netzwerk aufgeteilt sind. Dies wird durch die Transparenz beschrieben. Tanebaum unterteilt die Transparenz wie in Tabelle (2.1) zu sehen.

Verteilte System finden überall dort Anwendung, wo zum Beispiel viel Rechenleistung für Berechnungen benötigt wird, bei Zugriffen auf Datenbanksystemen oder in Autos für die Abgleichung von Daten usw.

Transparenz	Beschreibung
Zugriff	Verbirgt Unterschiede in der Datendarstellung und wie auf Ressourcen zugegriffen wird.
Standort	Verbirgt, wo sich eine Ressource. befindet
Migration	Verbirgt, dass eine Ressource eventuell an einen anderen Ort, verschoben worden ist.
Umzug	Verbirgt, dass eine Ressource während der Nutzung, verschoben werden kann.
Replikation	Verbirgt, dass die Ressource ein Replikat ist.
Nebenläufigkeit	Verbirgt, dass eine Ressource eventuell von vielen Benutzern verwendet wird.
Fehler	Verbirgt, die Fehler und das Wiederherstellen von einer Ressource.
Persistence	Verbirgt wie die Ressource gespeichert ist.

Tabelle 2.1: Verschiedene Formen der Transparenz in verteilten Systemen (ISO,1995)

2.4 Hypertext Transfer Protocol

Das Hypertext Transfer Protocol (HTTP) ist das Kommunikationsprotokoll des World Wide Webs. Zu den wichtigsten Funktionen gehört Dateien von Webservern anzufordern und in den Browser zu laden.

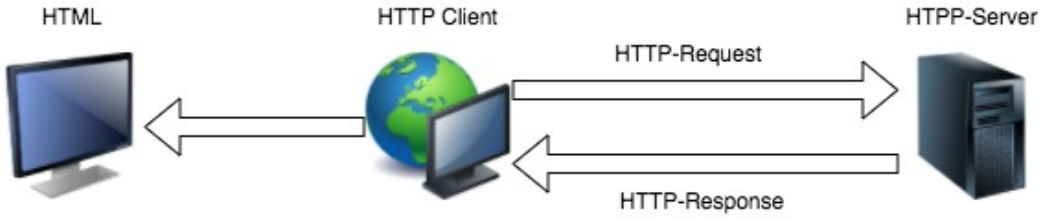

Abbildung 2.3: Kommunikation per HTTP

Bei HTTP wird das Client-Server-Prinzip zur Kommunikation (siehe Abbildung 2.3) angewendet. Ein HTTP-Client (Client) sendet eine Anfrage (Request) an einen HTTP-Server (Server, z.B. ein Webservice). Dieser bearbeitet die Anfrage und schickt eine Ant-

wort (Response) an den Client zurück. Der Client kann nun die Antwort nutzen, um Daten beispielsweise in einem HTML-Format anzuzeigen.

Die in der Kommunikation verwendeten Request- und Response-Nachrichten sind in einen Kopf- (Header) und einen Datenteil (Body) gegliedert. im Header befinden sich Steuerelemente und im Body können sich Dateien oder andere Daten befinden.

HTTP-Methoden

Jeder Request wird durch die Angabe einer Methode eingeleitet. Die Methoden teilt dem Server mit, wie er mit dem Request umgehen soll. Am häufigsten werden die Methoden GET und POST verwendet. Im Folgenden ein Überblick über alle HTTP-Methoden.

Get Die GET-Methode dient der Anforderungen von Daten. Über GET lassen sich auch Formular-Daten übertragen. Diese werden an die URL angehängt.

POST Die POST-Methode ist der GET-Methode ähnlich. Sie überträgt eine große Anzahl an Daten, die sich im Body befinden, an einen Server.

PUT Die PUT-Methode dient dazu Daten auf einen Server hochzuladen. Sie muss nicht wie POST durch ein Skript verarbeitet werden.

DELETE Die DELETE-Methode löscht die angegeben Daten auf einem Server.

HEAD Die HEAD-Methode zielt darauf ab, den Server anzuweisen, den gleichen Header wie bei GET zu senden, aber keinen Inhalt im Body zu übertragen.

PATCH Die PATCH-Methode ändert ein bestehendes Dokument auf einem Server. Das Dokument wird nicht, wie bei der POST-Methode vollständig ersetzt.

TRACE Die TRACE-Methode liefert eine Anfrage so zurück, wie ein Server sie empfangen hat. So wird nachvollzogen, ob die Anfragge auf dem Weg zum Server verändert wurde.

OPTIONS Die OPTIONS-Methode beauftragt den Server eine Liste von seinen Methoden und Merkmalen zu senden.

2.4.1 Request

Ein Request des HTTP besteht aus der URL, den Methoden und dem Request-Header. Am Anfang eines Request steht die Methode, dahinter kommen die URL und die verwendete HTTP-Version.

- Methode URL HTTP-Version
- Genereller Header
- Request Header
- Entität Header (optional)
- Leerzeile
- Request Entität

Die Leerzeile wird bei der POST-Methode für die Datenübertragung genutzt. Listing (2.1) zeigt einen beispielhaften POST Request.

Listing 2.1: Beispielhafter POST-Request

```
1  POST /upload.php HTTP/1.1
2  Host: fileserver.com
3  User-Agent: Mozilla/5.0
4  Accept: image/gif, image/jpeg, */*
5  Content-type: application/x-www-form-urlencoded
6  Content-length: 51
7  Connection: close
8
9
10 Name=Wer%C3%9Fen+R%C3%B6ssl&location=BeispielHuasen&PLZ=12345
```

2.4.2 Response

Die Response-Nachricht repräsentiert die Antwort des Servers auf ein Request des Clients. Der Repsonse besteht aus der verwendeten HTTP-Version, einem Status-Code des Responses und einer Klartext-Meldung des Status-Codes. Danach folgen die Header und die durch eine Leerzeichen getrennten Daten. Es gibt ein Vielzahl an verschiedenen Status-Codes, die auf der Seite des Internet Engineering Task Forces [Fielding und Reschke, 2014] eingesehen werden können.

- HTTP-Version und Status-Code
- Klartext-Meldung des Status-Codes
- Genereller Header
- Response Header
- Entität Header (optional)
- Leerzeile
- Daten, falls vorhanden

Listing (2.2) zeigt ein Beispiel für ein Response des Servers auf ein Request.

Listing 2.2: Beispielhafter Darstellung einer Response-Nachricht

```
1 HTTP/1.x 200 OK
2 Date: Tue, 08 Sep 2009 15:47:06 GMT
3 Server: Apache/1.3.34 Ben-SSL/1.55
4 Keep-Alive: timeout=2, max=200
5 Connection: Keep-Alive
6 Transfer-Encoding: chunked
7 Content-Type: text/html
```

2.4.3 HTTP-Authentifizierung

Sollen Daten und Informationen vor Dritten geschützt werden, kann das Basic-HTTP-Authentication-Schema verwendet werden. Hierfür wird der Request-Header um den Zusatz *Authorization* erweitert. Danach folgt die Art der Authentifizierung und eine Kodierung des Benutzernames und des Passwortes. Als sicher gilt dies Art der Authentifizierung nicht, da der Benutzername und das Passwort nicht verschlüsselt sind. Um die Sicherheit zu erhöhen, muss auf Verfahren wie SSL/TLS Verschlüsselung von HTTPS zurückgegriffen werden.

2.5 Docker-Container

> *Container sind eine schlanke und portable Möglichkeit, beliebige Anwendungen und Ihre Abhängigkeiten zu verpacken und transportabel zu machen*
> *(Adrian Mouat [Mouat, 2016, S. ix Vorwort])*

Somit ist sichergestellt, dass bei einer Portierung, die Anwendung lauffähig bleibt. Die Ähnlichkeit zu einer virtuellen Maschine (VM) ist naheliegend, da sich in einem Docker-Container eine Instanz eines Betriebssystems befindet, auf dem die Anwendungen laufen. Allerdings ist der Aufbau des Containers im Vergleich zu einer VM einfacher und damit effizienter. Siehe hierzu Abbildung (2.4)

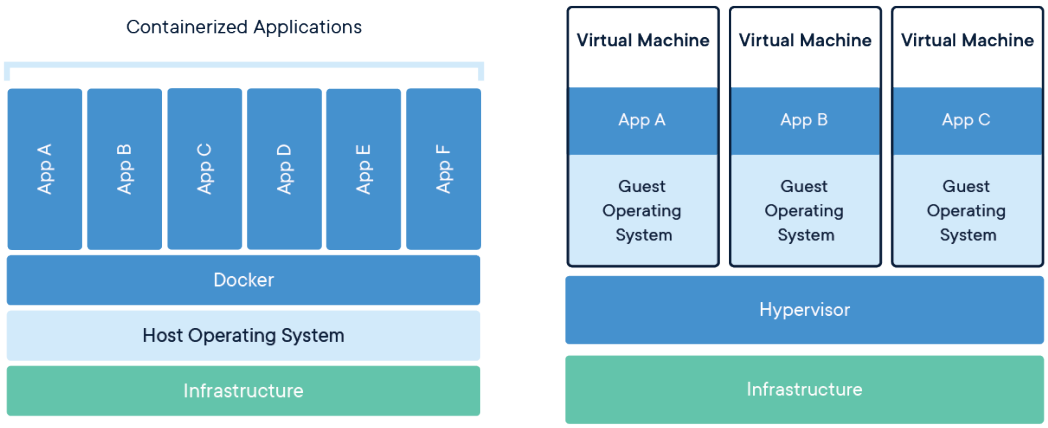

Abbildung 2.4: Aufbau und Unterschied eines Docker Containers und einer VM [Docker, 2018]

Auf der rechten Seite sind drei Anwendungen zu sehen, die auf einem Host in unterschiedlichen VMs laufen. Der Hypervissior wird dazu benötigt, VMs zu erstellen und laufen zu lassen. Des Weiteren steuert er den Zugriff auf das Gast-Betriebssystem. Jede virtuelle Maschine benötigt eine komplette Kopie des Betriebssystems für sich. Im Gegensatz dazu, verwenden die Anwendungen in einem containerisierten System gemeinsam den Kernel des Host-Betriebssystems. Jede Anwendung ist in einem separaten Container isoliert. Der Hauptunterschied ist, dass eine VM eine fremde Umgebung emuliert und ein Container, Anwendungen portabel und in sich abgeschlossen macht.

Carl Boettiger beschreibt in seiner wissenschaftlichen Abhandlung [Boettiger, 2015] Vorteile von Docker.

- Durch die effiziente Nutzung von Ressourcen können viele Container parallel auf einer Host-Maschine laufen, wodurch eine Simulierung und Nutzung eines verteilten Systems möglich wird.
- Durch die einfache Portierbarkeit eines Containers können Fehler bei der Änderung der Laufzeitumgebung umgangen werden.
- Anwender können einfach einen Container herunterladen, diesen ausführen und müssen sich keine Gedanken über nötige Abhängigkeiten machen.

2.5.1 Architektur von Docker

In Abbildung (2.5) sind die wichtigsten Komponente eines Docker-Systems zu erkennen.

Deamon Der Docker-Deamon ist unter anderem für das Erstellen, Ausführen und Überwachen der Container zuständig.

Client Der Docker-Client wird zur Kommunikation mit dem Docker-Deamon verwendet. Die Kommunikation findet über HTTP statt.

Registry In der Docker-Registry werden erstellte Images abgelegt und zur Verteilung zur Verfügung gestellt. Die Standard-Registry ist der Docker-Hub. Hier sind eine Vielzahl an Images zu finden.

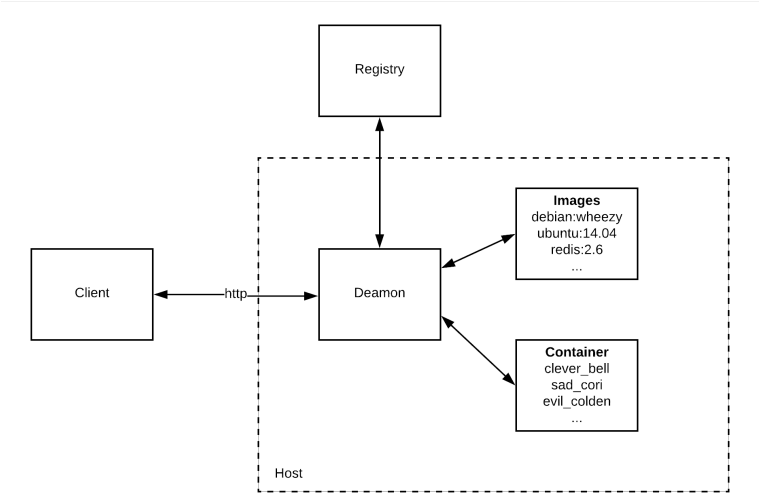

Abbildung 2.5: Die wichtigsten Docker-Komponenten [Mouat, 2016, S. 35]

Images Ein Docker-Image besteht aus vielen einzelnen Schichten, die eine Architekturlandschaft beschreiben.

Container Die Docker-Images werden in einen Docker-Container geladen und dort ausgeführt.

2.5.2 Dockerfile

Das wichtigste Element eines Docker-Containers ist die Dockerfile. Die Dockerfile ist eine Textdatei und beinhaltet Befehle, um ein Docker-Image zu erzeugen. Jede Anweisung erzeugt dabei eine Schicht des Images. Die einzelnen Schichten werden nach einander in einen eigenen Docker-Container geladen und gestartet. Nach jeder Ausführung des so erzeugten individuellen Docker-Containers wird nach der Erstellung der nächsten Schicht der vorige Docker-Container gelöscht. Am Ende entsteht ein Image mit einer Vielzahl von aufeinander liegenden Schichten. Jede dieser Schichten ist ein Read-Only-Dateisystem. Das erhaltene Image wird beim Starten in einen Docker-Container umgewandelt. Dabei fügt die Docker-Engine dem Image ein Read-Write-Dateisystem und verschiedene Einstellungen wie IP-Adresse und der Container-ID hinzu.

2.5.3 Docker Compose

Docker-Compose bietet die Möglichkeit mehrere Docker-Container zu definieren und zu betreiben. Damit dies bewerkstelligt werden kann, wird eine Docker-Compose Datei verwendet. Diese basiert auf der Sprache Yet Another Markup Language (YAML). Diese Datei erzeugt die Option mit einem einzigen Befehl alle Docker-Container mit bestimmten Konfigurationen zu starten.

3 Anforderungsspezifikation

In diesem Kapitel wird eine Anforderungsanalyse für die zu entstehende Portalseite und Integrationsumgebung durchgeführt. Hierzu wird zuerst ein genauerer Blick auf das HAWAI Projekt geworfen, um die Voraussetzungen aufzuzeigen. Danach werden die Anforderungen für diese Arbeit bestimmt und eingeschätzt.

3.1 Das HAWAI Projekt

Das HAWAI-Projekt (HAW Laboratory for Architecture and IT Management) [Hawai, 2018] ist eine Forschungsgruppe des Department Informatik an der Hochschule für Angewandte Wissenschaften in Hamburg. Die Forschungsgruppe wurde im Jahr 2017 ins Leben gerufen, um Themen wie IT Management, Softwarearchitektur und Datenscience unter dem Gesichtspunkt der Digitalisierung und der Globalisierung noch näher zusammen zu bringen und zu verbessern.

Das Hauptaugenmerk liegt darauf, in einem Laborversuch, eine virtuelle Umgebung für Anwendungslandschaften von Unternehmen zu erstellen und zu simulieren. Diese Umgebung gibt die Möglichkeit, dass Mitarbeiter und Studierende forschen, komplexe Softwarearchitekturen entwickeln und zu beherrschen lernen und Abschlussarbeiten erstellen zu können.

Durch die Möglichkeit der Erstellung von komplexen Softwarearchitekturen im HAWAI-Projekt, haben sich bereits einige in sich abgeschlossene Softwarelösungen angesammelt. Siehe hierzu zum Beispiel die Bachelor-Thesis Umsetzung komplexer Geschäftsprozesse in Verteilten Systemen mit Docker [Johannsen, 2018] oder das noch nicht ganz abgeschlossene IT-Projekt Änalyse Your Enterprise". Um bei der Vielzahl der Projekte nicht den Überblick zu verlieren, ist die Notwendigkeit einer Portalseite und Integrationsumgebung gestiegen.

3.2 Anforderungen an die Integrationsumgebung

Um die Anforderungen an die Integrationsumgebung zu spezifizieren, wurden in mehreren Gesprächen mit dem betreuenden Professor und den Kommilitonen des HAWAI-Projektes, Ziele und Aufgaben der Integrationsumgebung definiert. Die Anforderungen wurden in funktionale und nicht-funktionale Anforderungen, oftmals auch Qualitätsanforderungen genannt, (vgl. [Toth, 2015, S. 16]) unterteilt.

3.2.1 Funktionale Anforderungen

Die funktionalen Anforderungen beschreiben die Funktionalitäten und das Verhalten des Systems.

Benutzer kann sich registrieren

Das System soll einem Benutzer die Möglichkeit bieten, sich bei der HAWAI Umgebung zu registrieren.

Aktor: Benutzer

Vorbedingung: Der Benutzer ist noch nicht registriert.

Ablauf:

- Der Benutzer ruft die Portalseite und Integrationsumgebung über den Browser auf.
- Die Portalseite und Integrationsumgebung prüft, ob der Benutzer schon bekannt ist.
- Wenn nicht, den Benutzer auffordern, sich bei der Portalseite und Integrationsumgebung zu registrieren.

Auswirkung: Bei Erfolg wird der Benutzer im System registriert. Wenn die Anfrage nicht erfolgreich war, wird eine Fehlermeldung übertragen.

Benutzer kann sich einloggen

Das System soll einem Benutzer die Möglichkeit bieten, sich bei der HAWAI-Umgebung zu einzuloggen.

Aktor: Benutzer

Vorbedingung: Der Benutzer hat sich vorher registriert.

Ablauf:

- Der Benutzer ruft die Portalseite und Integrationsumgebung über den Browser auf.
- Die Portalseite und Integrationsumgebung prüft, ob der Benutzer schon bekannt ist und schon eingeloggt ist.
- Wenn ja, zeige geschützten Bereich an.
- Wenn nicht, den Benutzer auffordern, sich bei der Portalseite und Integrationsumgebung zu einzuloggen.

Auswirkung: Bei Erfolg wird der Benutzer in das System eingeloggt. Wenn die Anfrage nicht erfolgreich war, wird eine Fehlermeldung übertragen.

Benutzer kann sich ausloggen

Das System soll einem Benutzer die Möglichkeit bieten, sich bei der HAWAI-Umgebung zu abzumelden.

Aktor: Benutzer

Vorbedingung: Der Benutzer befindet sich in einem eingeloggten Zustand.

Ablauf:

- Der Benutzer befindet sich auf der Portalseite und Integrationsumgebung.
- Der Benutzer möchte die aktuelle Sitzung beenden und klickt den entsprechenden Button .
- Die Portalseite und Integrationsumgebung beendet die aktuelle Sitzung.

Auswirkung: Bei Erfolg wird der Benutzer vom System abgemeldet. Der Sitzungsstatus geht auf inaktiv.Wenn die Anfrage nicht erfolgreich war, wird eine Fehlermeldung übertragen.

Ansicht der verfügbaren Services

Das System soll einem Benutzer, der sich in dem geschützten Bereich befindet, die verfügbaren Services anzeigen.

Aktor: Benutzer

Vorbedingung: Der Benutzer ist im System eingeloggt.

Ablauf:

- Der Benutzer ruft die Portalseite und Integrationsumgebung über den Browser auf.
- Sind Services verfügbar, werden sie dem Benutzer angezeigt.
- Wenn nicht, werden keine Services angezeigt.

Auswirkung: Bei Erfolg wird eine Liste der verfügbaren Services angezeigt. Wenn die Anfrage nicht erfolgreich war, wird eine Fehlermeldung übertragen.

Ansicht verfügbarer Daten

Das System soll einem Benutzer, der sich in dem geschützten Bereich befindet, die für ihn verfügbaren und erstellten Daten anzeigen.

Aktor: Benutzer

Vorbedingung: Der Benutzer ist im System eingeloggt.

Ablauf:

- Der Benutzer ruft die Portalseite und Integrationsumgebung über den Browser auf.
- Sind Daten für den Benutzer verfügbar, werden diese angezeigt.
- Wenn nicht, werden keine Daten angezeigt.

Auswirkung: Bei Erfolg wird eine Liste der für den Benutzer verfügbaren und erstellten Daten angezeigt. Wenn die Anfrage nicht erfolgreich war, wird eine Fehlermeldung übertragen.

Authentifizierung von Services

Damit die Services, die Berechtigungen haben, Benutzerdaten abzurufen, müssen diese sich im System authentifizieren.

Aktor: Service

Vorbedingung: Die Services sind dem System bereits bekannt.

Ablauf:

- Der Service schickt eine Nachricht an eine Authentifizierungsschnittstelle des Systems.
- Das System prüft, ob der Service berechtigt ist.
- Wenn ja, darf der Service die Daten des Benutzers bearbeiten.
- Wenn nicht, dann erfolgt eine Fehlermeldung.

Auswirkung: Bei Erfolg erhält der Service die Berechtigung zur Verwendung der Benutzerdaten.

Wenn die Anfrage nicht erfolgreich war, wird eine Fehlermeldung übertragen.

Services werden über aktuelle Sitzung informiert

Die Services können den Status einer Sitzung des Benutzers abfragen.

Aktor: Service

Vorbedingung: Die Services sind dem System bereits bekannt.

Ablauf:

- Die Services können bei einer Schnittstelle anfragen, ob die Sitzung des Benutzers aktiv oder inaktiv ist.
- Die Schnittstelle teilt den Services, den Sitzungsstatus des Benutzers mit.
- Bei aktivem Status, unternehmen die Services nichts.
- Bei inaktivem Status, melden sie den Benutzer in ihrem Service ab.

Auswirkung: Bei inaktivem Status des Benutzers, wird in jedem registrierten und aktiven Service, der Benutzer abgemeldet.

Wenn die Anfrage nicht erfolgreich war, wird eine Fehlermeldung übertragen.

Datenspeicherung

Das System soll die Daten, die von einem Service generiert wurden, abspeichern können.

Aktor: Service

Vorbedingung: Der Service ist im System registriert und hat die Genehmigung auf die Datenschnittstelle zuzugreifen. Wenn die Anfrage nicht erfolgreich war, wird eine Fehlermeldung übertragen.

Ablauf:

- Der Service hat generierte Daten, die er im System für den Benutzer hinterlegen möchte.
- Der Service ruft eine Schnittstelle des Systems auf und übergibt die generierten Daten.
- Bei Erfolg werden die Daten im System unter dem Benutzerprofil gespeichert.
- Wenn nicht, dann erfolgt eine Fehlermeldung.

Auswirkung: Bei Erfolg werden generierte Daten unter dem Benutzerprofil im System eingetragen und gespeichert. Wenn die Anfrage nicht erfolgreich war, wird eine Fehlermeldung übertragen.

Daten löschen

Die Services sollen die Befugnis bekommen, Daten die nicht mehr benötigt werden, zu löschen.

Aktor: Service

Vorbedingung: Der Service ist im System registriert und hat die Genehmigung auf die Datenschnittstelle zuzugreifen.

Ablauf:

- Der Service ruft eine Schnittstelle des Systems auf, um den System mitzuteilen, dass Daten gelöscht werden sollen.
- Bei Erfolg werden die angegebenen Daten im System unter dem Benutzerprofil gelöscht.
- Wenn nicht, dann erfolgt eine Fehlermeldung.

Auswirkung: Bei Erfolg werden die angegebenen Daten unter dem Benutzerprofil gelöscht und aus dem System entfernt. Wenn die Anfrage nicht erfolgreich war, wird eine Fehlermeldung übertragen.

Daten erhalten

Das System soll Daten für die Services bereithalten und bei Anfrage an die Services übertragen.

Aktor: Service

Vorbedingung: Der Service ist im System registriert und hat die Genehmigung auf die Datenschnittstelle zuzugreifen.

Ablauf:

- Der Service spricht die Datenschnittstelle an und teilt dem System, mit welche Daten er übertragen bekommen möchte.
- Das System sucht die entsprechenden Daten und überträgt diese an den Service.
- Bei Erfolg werden die Daten übertragen.
- Wenn nicht, dann erfolgt eine Fehlermeldung.

Auswirkung: Bei Erfolg werden gewünschte Daten vom Benutzer an den Service übertragen und dieser kann die Daten verarbeiten. Wenn die Anfrage nicht erfolgreich war, wird eine Fehlermeldung übertragen.

Triggern von Services

Das System soll einem Service mitteilen, dass für diesen Daten zur Abholung bereitliegen.

Aktor: Service

Vorbedingung: Der Service ist im System registriert und es liegen Daten für diesen bereit.

Ablauf:

- Ein Service spricht eine Schnittstelle im System an und teilt dem System mit, dass Daten für einen anderen Service bereitstehen.
- Das System teilt dem zweiten Service mit, dass Daten für ihn vorliegen.
- Der zweite Service ruft eine Schnittstelle des Systems auf und fragt die entsprechenden Daten ab.
- Bei Erfolg werden die Daten an den zweiten Service übertragen.
- Wenn nicht, dann erfolgt eine Fehlermeldung.

Auswirkung: Bei Erfolg werden die entsprechenden Daten an den zweiten Service übertragen und dieser kann die Daten verarbeiten. Wenn die Anfrage nicht erfolgreich war, wird eine Fehlermeldung übertragen.

Verwaltung für Daten

Das System soll eine Verwaltung für Daten haben. Dieser soll die Zugriffsrechte auf die Daten organisieren.

Aktor: Service

Vorbedingung: Ein Service spricht die Datenschnittstelle des Systems an.

Ablauf:

- Ein Service spricht die Datenschnittstelle des Systems an.
- Die Verwaltung für die Dateien entscheidet, ob der Service zum jetzigen Zeitpunkt auf die gewünschten Daten zugreifen darf.
- Bei Erfolg gewährt die Verwaltung den Zugriff auf die Daten.
- Wenn nicht, wird der Service eingereiht und erhält eine Nachricht.

Auswirkung: Bei Erfolg ist es dem Service erlaubt, auf die entsprechenden Daten zuzugreifen. Wenn die Anfrage nicht erfolgreich war, wird der Service eingereiht und eine Nachricht wird an den ihn gesendet.

Monitoring

Das System soll ein Monitoring, der durchgeführten Aktionen bieten.

Aktor: Service

Vorbedingung: Services sind ordnungsgemäß registriert.

Ablauf:

- Verschiedene Services sprechen unterschiedliche Schnittstellen des Systems an und lösen so Aktionen innerhalb des Systems aus.
- Jede einzelne Aktion im System wird dem Monitoring mitgeteilt.
- Bei erfolgreichen Aktionen werden diese Nachrichten durchs Monitoring dargestellt.
- Bei Fehlern in den Aktionen werden diese Nachrichten durchs Monitoring dargestellt

Auswirkung: Im Monitoring-Bereich entsteht eine Vielzahl an Nachrichten, die dargestellt werden müssen.

3.2.2 Nicht-funktionale Anforderungen

> *Nicht-funktionale Anforderungen eines Softwaresystems haben einen großen Einfluss auf seine Entwicklung und Wartung, seine allgemeine Bedienbarkeit und seine Verwendung von Ressourcen. Sie haben ebenso einen Einfluss auf die Qualität und auf die Architektur einer Anwendung. Je größer und komplexer ein Softwaresystem wird, desto wichtiger sind die nicht-funktionalen Anforderungen.* *(Frank Buschmann [Buschmann, 1996, S. 404])*

Interoperabilität

Da das System für die Portalseite und Integrationsumgebung Teil des HAWAI-Projekts ist, ist es notwendig, dass das System mit anderen Systemen interagiert. Um Interoperabilität zu unterstützen, muss die Systemarchitektur so konzipiert werden, dass sie mit anderen System in der HAWAI-Landschaft kommunizieren kann. Eine Grundvoraussetzung hierfür ist, dass sich alle beteiligten Systeme auf aktuelle IT-Standards einigen und diese umsetzt. Bei Änderungen an der HAWAI-Landschaft ist so auch eine weitere Kommunikation aller Systeme möglich.

Erweiterbarkeit

Bei der Entwicklung von Softwaresystemen hat sich gezeigt, dass es eine zunehmende Häufigkeit für die Erweiterung oder Änderung bestehender Systeme gibt. Dies kann notwendig werden, wenn sich der aktuelle IT-Standard ändert oder neue Funktionen durch die Systemumgebung benötigt werden. Somit kann die Komplexität eines Systems schnell ansteigen und die Übersicht leiden.

Trotzdem muss einem Entwickler gewährt werden, die Zuverlässigkeit, die Sicherheit und die Benutzbarkeit des Systems möglichst einfach zu ändern. Des Weiteren sollten Schnittstellen definiert werden, durch die dem System weitere Funktionalitäten hinzugefügt werden können.

Zuverlässigkeit

Die Zuverlässigkeit eines Systems ist die Grundvoraussetzung, dass das System von einem Benutzer oder einem anderen System angenommen wird. So würde bei einer schlechten Zuverlässigkeit kein Nutzen aus dem System gezogen und das System wäre unbrauchbar. Daher muss zu jedem Zeitpunkt ein korrektes Verhalten des Systems ermöglicht werden. Sollte doch einmal ein Fehler auftreten, so muss der Übergang in einen gesicherten Zustand gewährleistet sein. Ein unverzeihlicher Fehler wäre es, wenn durch eine Fehlfunktion Informationen an Dritte preisgegeben würden.

Verfügbarkeit

Die Verfügbarkeit eines Systems sollte möglichst immer gegeben sein. Daher muss das System bei Störungen, möglichst schnell wieder in Betrieb genommen werden können.

Als Architekturansatz sollte eine Lösung gewählt werden, die es ermöglicht fehlerhafte Komponente auszutauschen, ohne dass das System eine längere Zeit ausfällt.

Sicherheit

Eine weitere Grundvoraussetzung für den Betrieb des Systems, muss die Sicherheit sein. Bei dem umzusetzenden System muss speziell auf zwei Bereiche der Sicherheit eingegangen werden. Zum einen mit dem Blick auf den Umgang der Daten und zum anderen mit dem Blick auf die Informationen über den Benutzer. Beide Bereiche der Sicherheit spielen eine große Rolle und dürfen nicht für Dritte sichtbar sein.

In Bezug auf die gespeicherten Daten, wäre es fatal, wenn jemand anderes als der Besitzer auf diese zugreifen und sie manipulieren könnte. Um dem entgegen zu wirken, muss mit den neuesten Sicherheitsstandards gearbeitet werden und so die Daten vor Fremdzugriff geschützt werden.

Die Informationen über den Benutzer sind ebenfalls vor der Außenwelt abzuschirmen. Hier könnte ein unberechtigter Zugriff ebenso verheerende Folgen nach sich ziehen. Würden die Informationen an einen Ddritten gelangen, würde es nicht nur die persönlichen Rechte des Benutzers angreifen, sondern es könnte auch zu einem Datendiebstahl kommen, der schwerwiegende Spätfolgen haben kann. Deswegen ist auch der Bereich des Benutzers in besonderer Art und Weise zu schützen.

Benutzbarkeit

Die Portalseite und Integrationsumgebung muss eine einfache Benutzbarkeit aufweisen. Ein wichtiger Gradmesser ist hierfür die Usability. Das heißt, dem Benutzer sollte es in kurzer Zeit möglich sein sich in dem System zurecht zu finden. Unter diesem Gesichtspunkt muss ein Blick auf die Verständlichkeit, die Erlernbarkeit und die Bedienbarkeit des Systems geworfen werden. Bei der Bedienbarkeit sollte zudem auf folgendes geachtet werden:

Bequemlichkeit Ausführen von Aktionen sollten möglichst einfach sein.

Geschwindigkeit Die Reaktionszeit beim Ausführen von Aktionen sollte möglichst gering sein.

Fehlervermeidung Der Benutzer sollte daran gehindert werden, Fehler im System zu verursachen.

3.3 Realisierbarkeitsanalyse

Auf Grund der bisher zusammengestellten Anforderungen, kann nun eine Analyse auf Realisierbarkeit des Projekts durchgeführt werden. Hierfür werden bestimmte Anforderungen an Hardware und Software gestellt. Diese werden in den nächsten Kapiteln, mit Blick auf die allgemeine Realisierbarkeit, diskutiert. Des Weiteren wird ein kurzer Überblick auf vergleichbare Softwarelösungen gegeben.

3.3.1 Hardware-Analyse

In der heutigen Zeit zielen Programmiertechniken und Programmiersprachen immer mehr darauf ab, effizient zu arbeiten. Ziel dieser Effizienz ist es, durch Optimierung weniger Speicher und Leistung auf dem ausführenden System zu verbrauchen. Dies wird zum Beispiel dadurch erreicht, einen benutzten Speicher frühestmöglich wieder frei zu geben, damit dieser wieder genutzt werden kann, oder durch einfachere Befehlssätze, um die Kompilierung zu beschleunigen. Diese Optimierung ist in der Regel soweit fortgeschritten, das aktuelle und angepasste Systeme keine größeren Schwierigkeiten haben, selbst komplexe Softwarelösungen zu betreiben.

Somit ist davon auszugehen, dass die Portalseite und Integrationsumgebung ebenfalls auf aktuellen Systemen betrieben werden kann. Der wichtigste Punkt, der beachtet werden muss, ist dem System genügend Speicherplatz zur Verfügung zu stellen. Hier könnte es beim Abspeichern von Dateien zu Problemen infolge von Speicherplatzmangel kommen.

Bei der Orchestrierung der Systemkomponenten, könnte Kubernetes zum Einsatz kommen. Kubernetes veröffentlicht sogenannte »Nodes« auf denen zum Beispiel Docker-Container laufen können.

3.3.2 Software-Analyse

Für die Umsetzung der Portalseite und Integrationsumgebung stehen eine Fülle von Programmiersprachen zur Verfügung. Da nicht jede Programmiersprache für die Umsetzung geeignet ist, muss eine Vorauswahl getroffen werden. So macht es zum Beispiel keinen Sinn, eine maschinencode-nahe Programmiersprache wie Assembler oder Very High Speed

Integrated Circuit Hardware Description Language (VHDL) zu wählen. Nichtsdestotrotz bleiben einen Menge Programmiersprachen übrig, die für den Einsatz geeignet wären.

Da in jedem Fall mindestens eine Datenbank vorhanden sein muss, würden sich hier die Sprachen für relationale Datenbanken, wie MySQL, Oracle, SQLLite anbieten. Alternativ wäre es auch denkbar NoSQL Datenbanken zu nutzen. Diese unterscheiden sich von den relationalen Datenbanken, indem sie keine festgelegte Tabellenschemata [Meier und Kaufmann, 2016]. Für die Web-Oberfläche der Portalseite und Integrationsumgebung würde sich eine Javascript-Sprache anbieten. Diese könnte Angular, Polymer, Ember.js oder React sein.

Wie man sieht, ist die Realisierung der Portalseite und Integrationsumgebung aus softwaretechnischer Sicht in jeden Fall umsetzbar.

3.4 Fazit

In diesem Kapitel wurde zuerst ein Blick auf das HAWAI-Projekt allgemein geworfen. Im nächsten Schritt wurden die Anforderungen an die Portalseite und Integrationsumgebung in funktionale und nicht-funktionale Anforderungen unterteilt. Durch die funktionalen Anforderungen, wurden die Anforderungen für die Portalseite und Integrationsumgebung aus Sicht des Benutzers und der Services untersucht. Ein Benutzer muss sich registrieren und einloggen können, danach soll er die für ihm zur Verfügung stehenden Services und Daten sehen können. Die Services sollen sich ebenfalls authentifizieren können und Daten speichern, löschen oder beantragen können.

Bei den nicht-funktionalen Anforderungen wurden die allgemeinen Anforderungen an die Portalseite und Integrationsumgebung diskutiert. Hier wurden unter anderem die Interoperabilität, die Erweiterbarkeit, die Zuverlässigkeit, die Sicherheit und die Benutzbarkeit genauer betrachtet. Durch die Betrachtung sind allgemeine Rahmenanforderungen entstanden, die bei der Realisierung der Portalseite und Integrationsumgebung berücksichtigt werden müssen.

Zum Schluss wurde noch ein Schwerpunkt auf die Realisierbarkeit der Portalseite und Integrationsumgebung mit Blick auf die Hardware und Software geworfen. Es hat sich herausgestellt, dass weder die Hardware noch die Software eine Realisierung des Systems verhindern.

4 Systementwurf

In diesem Kapitel wird der Systementwurf für die Portalseite und Integrationsumgebung diskutiert. Als Grundlage werden die zusammengestellten Anforderungen aus dem vorigen Kapitel genommen. Am Anfang wird der Systementwurf im Ganzen betrachtet und nach und nach detaillierter dargestellt. Dabei wird beschrieben, welche Programmiersprachen, Programmiertechniken und welche Aufgliederung für das System eingesetzt wurden.

4.1 Gliederung des Systems

Damit die Portalseite und Integrationsumgebung bestmöglich die Anforderungen (vgl. Punkt 3.2), wie Erweiterbarkeit, Zuverlässigkeit und Benutzbarkeit unterstützt, ist es gängig Funktionalitäten zu kapseln. Die zu kapselnden Funktionalitäten sollten in sich abgeschlossen sein. Das heißt, sie sollten nur einen Schwerpunkt behandeln und alleine lauffähig sein. Als Beispiel wäre zu nennen, dass alles, was mit der Verarbeitung von Dateien zu tun hat, eine gekapselte Funktionalität ist. Durch diese Gegebenheiten werden für die einzelnen Funktionalitäten Microservices verwendet. Dadurch, dass das System später im World Wide Web betrieben werden soll, werden diese Microservices, als Webservices (vgl. Punkt 2.2) entwickelt.

4.1.1 Aufteilung der Webservices

Die Aufteilung einer komplexen Softwarearchitektur kann durch viele Faktoren schwierig werden. Hierzu zählen sicherlich solche Faktoren wie:

- nicht klar zu definierende Funktionalitäten.
- ungenügend formulierte Anforderungen.

- das Ziel der Softwareumsetzung ist nicht verständlich.

Um dieser Problematik entgegenzutreten, wurde die Kapselungen der einzelnen Webservices in Gesprächen mit Kommilitonen ermittelt. Im Folgenden wird eine erste Übersicht über die separaten Webservices gegeben, die dann zu einem späteren Zeitpunkt eine noch genauere Betrachtung finden.

Dashboard

Damit ein Benutzer die Portalseite und Integrationsumgebung überhaupt nutzen kann, muss es eine Darstellungsform im World Wide Web geben. Somit ist es sinnvoll, alle Funktionalitäten, die sich mit der Darstellung des Inhalts der Portalseite und Integrationsumgebung beschäftigen, in einen eigenen Webservice mit eigener UI zu integrieren. Die UI ermöglicht eine Interaktion mit dem Benutzer des Hawai-Projekts. Der so separierte Webservice wird im Folgenden nur noch Dashboard genannt.

Auth-Server

Ein weiterer Webservice muss sich mit der Sicherheit der Portalseite und Integrationsumgebung beschäftigen. Darunter fallen Aufgaben wie den Benutzer zu registrieren und einzuloggen. Ebenso müssen Microservices von Projekten (Clients), die sich in der HAWAI-Landschaft befinden, hier registriert sein, damit sie Zugriff auf die geschützten Funktionalitäten erhalten.

Auth-Persistence

Damit der Webservice für die Sicherheit effektiv genutzt werden kann, benötigt er eine Persistence für die Datenspeicherung. In diesem Fall eine Datenbank, in der die Benutzerdaten und die registrierten Webservices abgespeichert werden können.

File-Manager

Der nächste bestimmte Webservice beschäftigt sich mit der Verarbeitung von Dateien. Dieser Webservice ist vor der Außenwelt und unbefugtem Zugriff durch den Auth-Server geschützt. Der File-Manager Webservice bietet die Möglichkeit, Dateien abzulegen, zu löschen und anzufragen. Des Weiteren können hier alle registrierten Clients und verfügbaren Dateien abgerufen werden.

File-Persistence

Ebenso wie bei der Auth-Persistence, wird von dem File-Manager ebenfalls eine Persistence benötigt. Zum einen für eine Datenbank, in der vermerkt wird, welche Dateien zu welchem Client (Webservice) gehören. Zum anderen muss ein Ort für die tatsächliche Lagerung der Dateien bereitgestellt werden.

4.1.2 Übersicht über das Gesamtbild

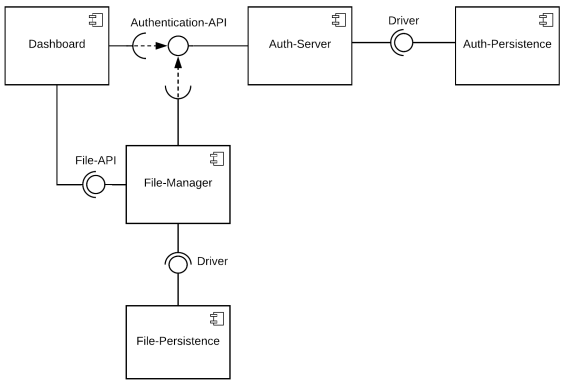

Abbildung 4.1: Übersicht der Komponenten und ihrer Schnittstellen

Die Abbildung (4.1) zeigt eine Übersicht, der einzelnen Komponenten der Portalseite und Integrationsumgebung. Die Komponenten werden als Black-Box dargestellt und die Schnittstellen, die sie nach außen hin anbieten oder die von anderen Komponente bezogen werden müssen, gezeigt. Die angebotenen Schnittstellen werden als »Lollipop« (Kreis) und benötigte als »Socket« (Halbkreis) dargestellt.

Dashboard

Die Dashboard-Komponente benötigt die angebotene Schnittstelle »Authentication-API« der Auth-Server-Komponente, um den Benutzer zu authentifizieren. Ebenfalls benötigt die Dashboard-Komponente die angebotene Schnittstelle »File-API« der File-Manager-Komponente, um mit dieser interagieren zu können. Die benötigten Schnittstellen der

Dashboard-Komponente, können auf gleicher Weise von anderen Services aus der HAWAI-Landschaft genutzt werden. Aus diesem Grund kann die Dashboard-Komponente stellvertretend für andere Services betrachtet werden.

Auth-Server

Die Auth-Server-Komponente bietet eine Schnittstelle »Authentication-API« für die Authentifizierung von Services an. Diese wird in der Abbildung (4.1) von der Dashboard-Komponente und der File-Manager-Komponente benötigt. Auf der anderen Seite benötigt der Auth-Server die Schnittstelle »Driver« für die Kommunikation mit der Auth-Persistence-Komponente.

Auth-Persistence

Die Auth-Persistence-Komponente stellt eine Schnittstelle »Driver« für die Auth-Server-Komponente zur Verfügung. Diese Schnittstelle wird von der Auth-Server-Komponente genutzt, um Daten in der Auth-Persistence-Komponente abzubilden.

File-Manager

Die File-Manager-Komponente bietet eine Schnittstelle »File-API« , die von Services, in der Abbildung (4.1) durch die Dashboard-Komponente dargestellt, benötigt wird. Diese Schnittstelle ermöglicht es, Services auf die Daten der File-Manager-Komponente zugreifen zu lassen. Des Weiteren benötigt die File-Manager-Komponente, die Schnittstelle »Authentication-API« der Auth-Server-Komponente, um eine Authentifizierung für die »File-API« Schnittstelle durchzuführen. Ebenso wird eine Schnittstelle »Driver« zur File-Persistence-Komponente benötigt. Über diese Schnittstelle können Daten in der File-Persistence verwaltet werden.

File-Persistence

Die File-Persistence bietet, wie die Auth-Persistence-Komponente, eine Schnittelle »Driver« an. Diese wird von der File-Manager-Komponente genutzt, um Daten in der File-Persistence-Komponente abzubilden.

In den nächsten Unterkapiteln werden die einzelnen Komponenten detaillierter entworfen. Es wird begründet, warum sich für eine Programmiersprache entschieden wurde, wie der

allgemeine Aufbau der Programmstruktur aussieht und es wird am Ende ein Schwerpunkt auf die Kommunikation zwischen den einzelnen Komponenten gesetzt.

4.2 Auth-Server

Zum Schutz der Informationen und Dienste eines Webservices, hat sich das Protokoll OAuth 2.0 (OAuth2) als Standard durchgesetzt. Dieses Verfahren nutzen einen Identity Provider um eine Autorisierung (authorization) für den Zugriff auf verschiedene Web-APIs zu ermöglichen. Gängig ist es OAuth2 auch für einen Login (Authentisierung) und Signle Sign-on (SSO) zu verwenden. Beim SSO kann ein Benutzer nach der Authentifizierung auf alle Dienste einer Anwendung zugreifen, für die er die Autorisierung bekommen hat, ohne sich bei jedem Dienst erneut anmelden zu müssen. Um diese Funktionalität zu unterstützen, wird OAuth2 um OpenId Connect oder Security Asertion Markup Language (SAML2.0) erweitert. Die Nutzung von SAML2.0 hat sich allerdings als eher schwer zu beherrschen herausgestellt. Dies liegt hauptsächlich an der Verwendung von X.509-Zertifikaten. Das ist auch der Grund, warum SAML2.0 in dieser Arbeit keiner genaueren Betrachtung unterzogen wird.

4.2.1 Das OAuth2-Protokoll

Wie erwähnt verwaltet und gewährt das OAuth2-Protokoll einen sicheren Zugriff für Clients (hier seien unter anderem Desktop und Mobile Clients genannt) auf Web-APIs. Dabei haben sich zwei wichtige Konzepte herausgestellt. Zum einen eine föderierte Identität (Federated Identity) die einem Service-Provider erlaubt, sich mit den Benutzerdaten bei einem anderen Service-Provider anzumelden. Zum anderen die delegierte Identität (Delegated Identity). Diese ermöglicht es einem Service, mit der Zustimmung des Benutzers, auf einen bestimmten Bereich (Scopes) des Benutzers zuzugreifen.

Der vereinfachte allgemeine Ablauf einer Authorization-Anfrage kann in Abbildung (4.2) verfolgt werden.

1. Der User fragt beim Client nach (Request) einer Ressource.
2. Der Client benötigt zunächst eine Berechtigung und leitet daher die Anfrage an den Authorization-Server weiter.

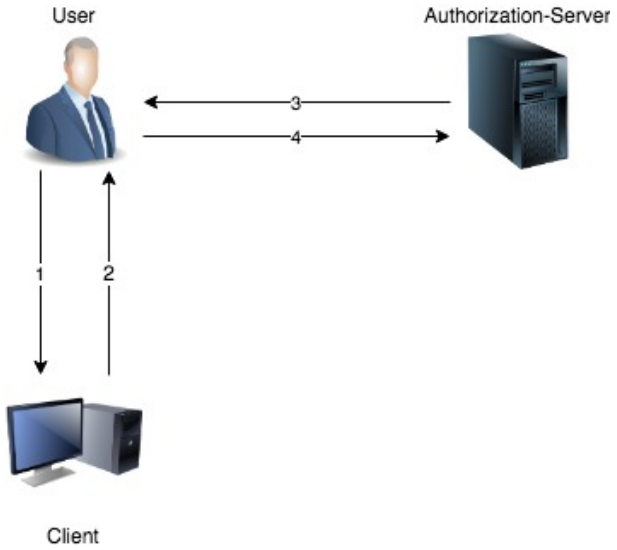

Abbildung 4.2: Vereinfachter allgemeiner Ablauf einer Authorization-Anfrage

3. Der Authorization-Server fragt den User, ob er dem Client die Berechtigung erteilen möchte.

4. Bei Bestätigung erhält der Client Zugriff auf die Ressource und der Benutzer kann die Ressource verwenden.

Der vierte Schritt ist bisher nur vereinfacht dargestellt worden. Um den Austausch von Information über die Berechtigung sicherer zu machen, wurden so genannte Grant-Types eingeführt. Von diesen gibt es zur Zeit vier Stück: Authorization-Grant, Implicit-Grant, Resource-Owner-Password-Credentials-Grant und Client-Credentials-Grant. Dabei finden der Authorization-Grant und Implicit-Grant die meiste Anwendung. Authorization-Grant wird in der Regel als serverseitiger Arbeitsablauf (Server-Side-Workflow) und der Implicit-Grant als clientseitiger Arbeitsablauf (Client-Side-Workflow) bezeichnet.

Der Client-Side-Workflow wird bei nicht vertrauenswürdigen Clients verwendet. Als nicht vertrauenswürdiger Client werden diejenigen eingestuft, bei denen es nicht sicher ist vertrauliche Daten auf der Clientseite zu speichern. Dies ist zum Beispiel bei HTML/JavaScript Anwendungen der Fall, da hier alle Information im Browser gespeichert werden müssen. Im Gegensatz dazu wird der Server-Side-Workflow bei vertrauenswürdigen Clients eingesetzt.

Bei allen Grant-Types wird für den sicheren Zugriff auf Ressourcen mit einem Access-Token und (optional) einem Refresh-Token gearbeitet. Der Authorization-Server stellt ein Access-Token für den Benutzer aus und mit diesem Access-Token kann ein Client auf die für ihn erlaubten Bereich zugreifen. Das Refresh-Token dient zur Erneuerung des Access-Token durch den Authorization-Server, wenn das Access-Token ausgelaufen und ungültig geworden ist.

In Abbildung (4.3) ist ein Authorization-Vorgang mit Implicit-Grant dargestellt.

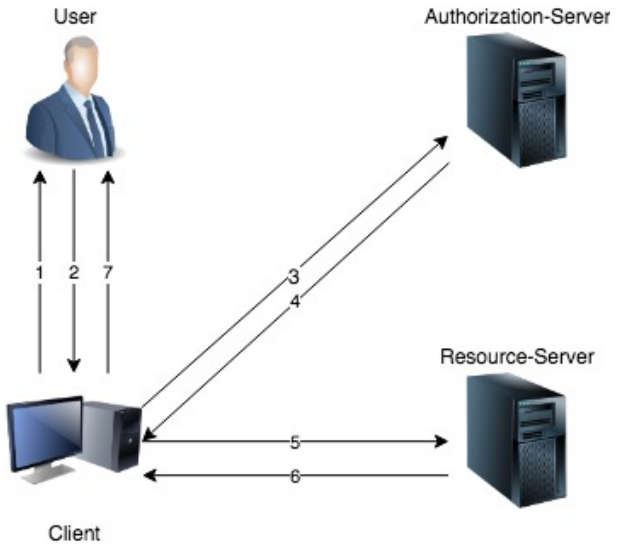

Abbildung 4.3: Vereinfachter allgemeiner Ablauf einer Authorization-Anfrage

1. Der Client fragt nach einer Berechtigung des Users (Benutzers) um auf eine Ressource zuzugreifen.
2. Der Benutzer erteilt dem Client die Berechtigung.
3. Der Client teilt die Berechtigung dem Authorization-Server mit.
4. Der Authorization-Server stellt für die Ressource ein Access-Token aus und übermittelt es an den Client.
5. Der Client übermittelt das Access-Token an den Ressourcen-Server.

6. Der Ressourcen-Server sendet, nach erfolgreicher Verarbeitung des Access-Token, Daten an den Client.
7. Der Client stellt die Daten für den Benutzer bereit.

Das Vorgehen beim Authorization-Grant ist ähnlich, nur dass hier mehr Informationen im Client gespeichert werden.

Sicherheit des OAuth2-Protokolls

Um einen OAuth2-Vorgang bestmöglich zu sichern, sollte nach [Bihis, 2015] auf folgende Punkte geachtet werden:

- Transpor Layer Security (TLS) für sichere Kommunikationskanäle verwenden.
- Es sollten nur kleine Scopes vom Client für das angefordert werden, was gebraucht wird.
- Beim Implicit-Grant sollten die Requests Read-Only sein.
- Tokens sollten nicht in Kontakt mit Benutzern kommen.
- So oft es geht den Refresh-Token verwenden.

Nichtsdestotrotz sind bei solchen Absicherungen immer noch Attacken auf den OAuth2-Mechanismus möglich. Charles Bihis [Bihis, 2015] beschreibt die bekanntesten Attacken und deren Verhinderung wie folgt:

Cross-site request forgery (CSRF) Als Beispiel für diesen Angriff, könnte eine Benutzer sich auf einer Webseite für Online Banking eingeloggt haben. Nun könnte eine Email geöffnet werden, die den Benutzer in einem weiteren Fenster des Browsers, auf die Webseite des Angreifers leitet. Im Hintergrund der Webseite des Angreifers, werden nun Requests an die Webseite der Online-Bank gesendet. Solange der Benutzer noch eine gültige Sitzung auf der Webseite der Online-Bank hat, werden die Requests als gültig angesehen. So können zum Beispiel Überweisung getätigt werden. - Zur Verhinderung dieser Attacke wird bei einem Request, ein »state« Parameter mit übergeben, der durch den Browser des Benutzers auf Gültigkeit geprüft wird.

Pishing Eine Internetseite tarnt sich als die eigentlich vom Benutzer aufgerufene Internetseite und greift so die geheimen Information des Benutzers ab. - Durch die Benutzung von Standard-Browsern, kann der Benutzer eine höhere Authentizität des richtigen Authorization-Endpunktes erlangen.

Redirection URI manipulation Wenn ein Client ein Request für den Benutzer beim Authorization-Server macht, wird eine Redirect-Uri mit übermittel, an die nach erfolgreichen Login weitergeleitet wird. Diese Redirect-URI könnte von einem Hacker manipuliert werden und der Authorization-Server würde auf eine andere (falsche) Internetseite weiterleiten. Diese Manipulation kann durchgeführt werden, wenn der Authorization-Server Wildcard Redirect-URIs zulässt. Wildcard Redirect-URIs sind Platzhalter für URIs, die zur Laufzeit durch Benutzer angelegt werden können. - Um dies zu verhindern, werden die Redirect-URIs vorab beim Authorization-Server registriert.

Für detailliertere Informationen über OAuth2 und die Verhinderung von Attacken vergleiche Mastering OAuth2 [Bihis, 2015] und OAuth2 in Action [Richer und Sanso, 2017].

4.2.2 OpenId Connect

OpenId Connect ist ein Protokoll, das auf dem OAuth2-Protokoll aufsetzt. Es erweitert das OAuth2-Protokoll um die Möglichkeit der Authentifizierung (authentication), so dass eine komplette Lösung für den Vorgang der Authentifizierung und Autorisierung möglich ist. Hierfür wird eine sogenannte Identity-Schicht auf das OAuth2-Protokoll aufgesetzt. Dies erlaubt einem Client die Identität eines Benutzers auf Basis des Autorisierungsvorganges festzustellen ohne sich um irgendwelche Passwörter kümmern zu müssen.

Der Ablauf eines OAuth2-Protokolls mit einem OpenId Connect-Protokoll in Bezug auf den Implicit-Grant ähnelt dem Vorgehen aus Abbildung (4.3). Der Request vom Client zum Authorization-Server wird um die OpenId Connect Scopes »openid«, »profile«, »email«, »adress«, »phone« erweitert. Wobei nur der »openid« Scope benötigt wird, alle anderen Scopes sind optional. Als Antwort (Response) erhält der Client vom Authorization-Server ein Access-Token und ein ID-Token. In dem ID-Token kann der Client dann Informationen über den User bekommen. Dies ist eine vereinfachte Darstellung des Ablaufes, da es für jeden Grant-Type mehre Arten der OpenId Connect Anfrage

gibt. Erwähnt sei hier, dass als »openid« Scope »code«, »token«, »id_token« und deren Kombinationen im Request übermittel werden kann. Für ein tieferes Einsteigen in OpenId Connect kann die Spezifikation von [Sakimura u. a., 2014] empfohlen.

4.2.3 Identity-Server

Für die Umsetzung eines Authentifizierungs-Webservice mit OpenID Connect und OAuth2 ist die Wahl auf das Open-Source-Projekt Identity-Server [Brock und Baier, 2016] in der Version 4 gefallen. Es gibt zwar auch noch Alternativen, wie OpenIddict [OIDCS, 2017]. Allerdings sind diese nicht so verbreitet und erfahren dementsprechend nicht so viel Unterstürzung. Beide Varianten laufen auf der Plattform von ASP.NET Core 2.

Features des Identity-Server 4

Authentification als Service Zentralisierte Login-Logik und Verarbeitung aller Anwendungen wie Web, Mobile, Services.

Single Sign-on / Sign-Out Single Sign-on / out für viele Anwendungstypen.

Zugangskontrolle für APIs Stellt Access-Tokens für eine API für viele Arten von Anwendungen aus.

Federation Gateway Unterstützung für einen externen Identity-Provider wie Google, Facebook usw.

4.2.3.1 Aufbau

Identity-Server ist eine Middleware, die einer ASP.NET Core-Anwendung OAuth2 und OpenId Connect hinzufügt und Endpunkte bereitstellt. Diese Endpunkte werden von Client-Anwendungen angesprochen. Abbildung (4.4) zeigt eine Übersicht, wie die Identity-Server-Middleware in eine ASP.Core Anwendung eingefügt wird und die Endpunkte bereitstellt. Die Endpunkte können noch durch mehrere Funktionalitäten erweitert werden.

Die Definition von bestimmten Rollen sind in der Literatur unterschiedlich. Deswegen wurde die Terminologie [Brock und Baier, 2016] beim Identity-Server wie folgt bestimmt:

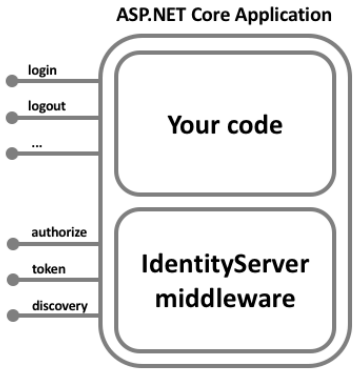

Abbildung 4.4: Übersicht des Zusammenspiels von Identity-Server und eine ASP.NET Core-Anwendung aus [Brock und Baier, 2016, The Big Picture].

Benutzer

Ein Benutzer ist ein Mensch, der einen registrierten Client nutzt, um einen Zugang zu Ressource zubekommen.

Client

Ein Client ist eine Softwareanwendung (Webservice), der Tokens vom Identity-Server anfordert. Entweder für die Authentifizierung des Benutzers (Identity-Token) oder um einen Zugang zu den Ressourcen (Access-Token) zu erhalten. Bevor der Client Tokens anfordern kann, muss dieser beim Identity-Server registriert sein.

Ressourcen

Ressourcen werden durch den Identity-Server geschützt. In der Regel sind es die Daten des Benutzers (Identity-Daten oder auch Claims genannt) oder eine API. Jede Ressource muss einen eindeutigen Namen haben, damit ein Client Zugang zu der richtigen Ressource bekommen kann.

Identity-Token

Ein Identity-Token repräsentiert das Resultat eines Authentifizierungsprozesses. Das Identity-Token beinhaltet nur ein Minimum an Identifizierungsmerkmalen über den Benutzer und wann und wie der Benutzer sich authentifiziert hat.

Access-Token

Durch ein Access-Token ist es erlaubt, Zugang zu den Ressourcen einer API zu erlangen. Clients beantragen Access-Tokens und leiten diese an die API weiter. Access-Token beinhalten Informationen über den Client und optional Informationen über den Benutzer. APIs nutzen das weitergeleitete Access-Token um zu überprüfen, ob es dem Client erlaubt ist, auf die Daten der API zuzugreifen.

Access-Token können als Self-Contained und Referenz auftreten. Als Beispiel für ein Self-Contained Access-Token, ist ein JSON Web Token (JWT) zu nennen. JWT ist eine gesicherte Datenstruktur, in der sich Claims und eine Auslaufzeit des Token befinden. Hat eine API die Datenstruktur ausgelesen, kann sie den Self-Contained Access-Token selbst bestätigen und braucht keine Rücksprache mehr mit dem Aussteller des Tokens, über die Gültigkeit des Tokens zu halten. Das macht es schwer ein JWT abzulehnen. Sie bleiben solange gültig, bis sie ausgelaufen sind.

Bei der Verwendung eines Referenz-Token speichert der Identity-Server den Inhalt des Tokens in einer Datenbank und stellt nur eine einzigartige Kennung an den Client aus. Die API erhält den Referenz-Token vom Client und erkundigt sich bei dem Identity-Server, ob die Kennung gültig ist. Der Vorteil dieser Variante ist es, dass die Verwaltung von Laufzeiten des Access-Tokens genau kontrolliert und bestimmt werden kann. Dieser Vorgang erhöht die Sicherheit des Systems. Der Hauptnachteil besteht darin, dass die API immer wieder nachfragen muss, ob die Kennung gültig ist.

Für die Umsetzung der Portalseite und Integrationsumgebung wurde sich für die Variante des Referenz-Token entschieden. Das liegt darin begründet, dass die Sicherheit des System dem erhöhten Netzwerkverkehr vorzuziehen ist.

4.2.3.2 Entwicklung der Endpunkte

Die Endpunkte dienen als Schnittstelle für die Kommunikation mit anderen Webservices. Bei der Integration des Identity-Server in eine ASP.NET Core-Anwendung sind standard-

mäßig die Endpunkte für *Discovery*, *Token* und *Authorize* verfügbar (vgl. Abbildung 4.4). Da diese drei Endpunkte nur die minimale Konfiguration eines Identity-Server erlauben, werden in diesem Abschnitt die Standard-Endpunkte erörtert und eine Betrachtung auf weitere Endpunkte durchgeführt.

Der Discovery-Endpunkt

Der Discovery-Endpunkt ist einer der drei Standard-Endpunkte. Er wird dazu genutzt Metadaten über den Identity-Server abzufragen. Als Antwort erhält man eine JSON-Nachricht, in der Informationen über den Ausstellernamen, unterstütze Scopes und verfügbare Endpunkte stehen. Dieser Endpunkt sollte als erstes von einem Webservice angesprochen werden. Angesprochen werden kann er über den Adresszusatz des Auth-Servers »/.well-known/openid-configuration«.

Der Token-Endpunkt

Der Token-Endpunkt wird verwendet, um eine Anfrage für einen Token durchzuführen. Dieser Endpunkt wird in der Regel nach dem Authorize-Endpunkt verwendet.

Der Authorize-Endpunkt

Dieser Endpunkt wird verwendet, wenn ein Webservice eine Anfrage nach einem Token oder einer Authorization stellt. Diese Anfrage erfordert, dass der Benutzer eingeloggt ist. Ist dies nicht der Fall, wird der Webservice zum Login-Endpunkt oder Registrierung-Endpunkt der ASP.NET Core-Anwendung umgeleitet. Hierbei muss der Client eine Redirect-URI angeben, zu der nach einem erfolgreichen Login oder Registrierung zurückgeleitet wird.

Die weiteren Endpunkte sind optional und müssen nicht unbedingt implementiert werden. Wie in Punkt (4.2.3.1) bestimmt, setzt dieser Entwurf auf einen Referenz-Token. Somit müssen einige der nächsten Endpunkte zwangsläufig umgesetzt werden.

Der UserInfo-Endpunkt

Innerhalb des Access-Token sollen so wenig Informationen wie möglich über den Benutzer mitgeteilt werden. Werden trotzdem von einem Webservice oder einer geschützten API Informationen über den Benutzer benötigt, wird der UserInfo-Endpunkt angesprochen. Die Komponente, die eine Anfrage stellt, muss ein gültiges Access-Token mitsenden.

Abhängig von den Scopes werden die Claims übertragen. Um den beschriebene Anforderungen (vgl. Punkt 3.2) der Portalseite und Integrationsumgebung gerecht zu werden, wird dieser Endpunkt umgesetzt.

Listing 4.1: Beispiel für einen Request und Response Verlauf für einen Endpunkt

```
Request:

GET /connect/userinfo
Authorization: Bearer <access_token>

Response:

HTTP/1.1 200 OK
Content-Type: application/json

{
    "sub": "248289761001",
    "name": "Bob Smith",
    "given_name": "Bob",
    "family_name": "Smith",
    "role": [
        "user",
        "admin"
    ]
}
```

Der Aufbau eines Request und eines Response am Beispiel des UserInfo-Endpunktes ist im Listing (4.1) zu sehen. Beim Request wird das Access-Token im *Header*, mit dem Tag Authorization, übertragen. Das Bearer in dieser Zeile besagt, dass das Access-Token in einem verschlüsselten String übertragen wird. Als Response-Nachricht wird ein JSON-String übertragen. Im Body der Response-Nachricht sind dann alle relevanten Informationen enthalten. Für alle Endpunkte ähnelt sich das Verfahren und der Aufbau. Der einzige Unterschied ist die Adresse des Endpunktes und der Inhalt der Response-Nachricht.

Introspection-Endpunkt

Der Introspection-Endpunkt ist eine Implementierung des RFC 7662-Standards [Richter, 2015]. Dieser Endpunkt wird verwendet, damit eine API ein Referenz-Token validieren

kann. Ebenso wie der UserInfo-Endpunkt benötigt der Introspection-Endpunkt eine Authentifizierung. Der Client sendet im *Header* das Referenz-Token bei einem Request an die API mit. Die API spricht den Introspection-Endpunkt des Identity-Servers an. Im *Header* des Requests, gibt die API den eindeutigen Namen der API und eine API-Secret an. Im Body wird das Referenz-Token, des Client übertragen. Nach der Überprüfung des Referenz-Token durch den Identity-Server, erhält die API eine Antwort mit dem Inhalt »active: true« für einen erlaubten Zugriff oder »active: false« für einen unberechtigten Zugriff.

End-Session-Endpunkt

Wenn sich ein Benutzer abmelden möchte, leitet der Webservice den Benutzer an den End-Session-Endpunkt weiter. Hierzu sind alle Webservices autorisiert, bei denen sich der Benutzer während seiner Sitzung angemeldet hat. Nach dem erfolgreichen Abmeldevorgang wird der Benutzer zurück zum Webservices geleitet, der die Anfrage an den End-Session-Endpunkt gestellt hat.

Check-Session-Frame-Endpunkt

Im Unterkapitel funktionale Anforderungen (Punkt 3.2.1) wurde festgelegt, dass es eine Funktionalität geben soll, die es allen Microservices ermöglicht zu erfahren, ob die Sitzung des Benutzers aktiv oder inaktiv ist. Diese Single Sign out-Funktionalität wird über den Check-Session-Frame-Endpunkt abgedeckt. Die Microservices schicken in regelmäßigen Abständen ein Request an diesen Endpunkt. Als Antwort erhalten sie ein »true« zurück. Sollte in der Zwischenzeit ein Webservice den End-Session-Endpunkt kontaktiert haben, bekommt ein Webservice beim nächsten Request an den Check-Session-Frame-Endpunkt eine »false« zurück. Somit weiß der Webservice, dass der Benutzer sich in einem anderen Webservice abgemeldet hat und und terminiert die aktuelle Sitzung des Benutzers bei sich selbst.

4.2.3.3 Identity-Middleware

Eine Identity-Middleware basiert auf dem ASP.NET Core Identity Framework und verwaltet die Identitäten eines Benutzer in einer Datenbank. Unterstützt wird die Identity-Middleware vom Entity-Framework. Das Entity-Framework ist ein Framework für objektrelationale Abbildung (ORM). Eine objektrelationale Abbildungen findet Anwendung

bei einer objektorientierten Programmiersprache. Hierzu findet ein Mapping zwischen den Objekten der Programmiersprache und den Tupeln der Datenbank statt. Die Aufgabe der Identity-Middleware ist es einer ASP.NET Core-Anwendung, in diesem Fall dem Identity-Server, bei der Verwaltung und Speicherung von Benutzerdaten in einer Datenbank behilflich zu sein.

Aufgaben des Identity Frameworks können wie folgt, beschrieben werden:

- Einfache Erweiterung der Profilinformationen über den Benutzer: Standardmäßig unterstützt OpenID Connect Informationen über den Benutzer wie Vorname, Nachname, Adresse oder Telefonnummer. Diese Benutzerinformationen können durch Identity einfach in der Datenbank abgebildet werden.
- Identity unterstützt die Verwendung von Rollen als Zugangsberechtigung zu bestimmten Bereichen. Es können etwa Rollen wie »Admin« oder »User« angelegt und den einzelnen Benutzern zugeordnet werden.
- Das Identity Framework basiert auf Claims: Claims sind mit einem Benutzer verknüpft und beschreiben die Identität des Benutzers genauer.
- Es wird eine Schnittstelle angeboten um OAuth-Schnittstellen von sozialen Seiten wie Facebook und Google anbinden zu können.

Somit erleichtert die Verwendung der Identity-Middleware die Verwaltung von Benutzern und die Kommunikation zwischen dem Identity-Server und einer Datenbank. Wegen dieser Vorteile fiel die Entscheidung die Identity-Middleware in den Identity-Server zu integrieren.

4.3 Auth-Persistence

Die Auth-Persistencen sollen in der Lage sein alle relevanten Daten, die vom Identity-Server genutzt werden, in einer Datenbank zu lagern und bei Bedarf an den Auth-Server zu übergeben. Für die Modellierung einer Datenbank gibt es eine Vielzahl an verschiedenen Datenbankmanagementsystemen (DBMS). Dabei können zwischen einem relationale oder einem nicht-relationale DBMS unterschieden werden. Die bekanntesten Vertreter eines relationalen DBMS sind MySQL, Oracle oder PostgreSQL. Diese werden in der Sprache SQL entwickelt. Die nicht-relationalen DBMS werden in der Regel NoSQL DMBS

genannt. Unterschiede finde sich in der Skalierbarkeit, Performance und dem Programmieraufwand [Binani u. a., 2016]. In der vorliegenden Arbeit wurde sich für das relationale DBMS MySQL entschieden. Die Entscheidung fiel auf MySQL, da es eine der verbreitetsten DBMS ist und für die Performance der Portalseite und Integrationsumgebung ausreichend ist.

Die Daten, die in die Datenbank ausgelagert werden müssen, können in zwei Hauptkategorien unterteilt werden. Zum einen, alle relevanten Daten für das ASP.NET Core Identity Framework und zum Anderen Konfigurationsdaten (Ressource, Clients) und Operations-Daten (Tokens, Codes, Consents), die direkt vom Identity-Server genutzt werden.

Durch die Verwendung des ASP.NET Core Frameworks und dem enthaltenen Entity Frameworks wird die Modellierung der Datenbank sehr erleichtert. Das Entity Framework bietet die Möglichkeit automatisch aus einem Schema eine Datenbankstruktur aufzubauen (Migration). Hierfür muss eine Datenbank angelegt werden. Beim Startvorgang des Identity-Servers generiert dieser die Struktur der kompletten Datenbank.

4.4 Dashboard

Für die Darstellung der Web-Oberfläche, bieten sich verschiedene Umsetzungsmöglichkeiten an. Zum einen gäbe es Single-Page-Applications (SPA) und Multi-Page-Application (MPA) und zum Anderen Server-Side-Application.

Bei der Server-Side-Application wird ein Großteil der Anwendungslogik auf der Serverseite bereitgestellt. Da die Kommunikation für Web-Applications im Mittelpunkt steht, ergibt sich für die Server-Side-Application ein hohes Maß an Datenfluss zwischen dem Client und dem Server, auf dem die Application liegt. Der Grund hierfür ist, dass für jeden Inhalt oder für jede Logik auf der Clientseite eine Nachricht zum Server geschickt werden muss.

Bei SPAs und MPAs wiederum wird die Anwendungslogik auf der Clientseite erstellt und die Kommunikation mit einem Server findet über eine Web-API statt. Der Vorteil hier ist, dass durch diese Variante viel weniger Kommunikation zwischen dem Server und der SPA oder MPA notwendig ist. Oftmals muss sich der Client nur für Daten an den Server wenden. Alles andere kann auf der Clientseite bearbeitet werden.

4.4.1 Unterschied SPA und MPA

Die Architektur von einer SPA ist so arrangiert, dass nur ein Teil der Inhalte aktualisiert wird, wenn man auf eine neue Seite zugreift. Somit ist eine erneutes Laden des gleichen Inhalts nicht nötig. Eine SPA besteht nur aus einer Web-Seite, welche bei Bedarf den Inhalt lädt. Sie verarbeitet die Markup-Languages (HTML und CSS) und Daten unabhängig und rendert die Ergebnisse direkt im Browser. Dies kann so umgesetzt werden, da SPAs mit einem JavaScript Framework arbeiten. Single-Page-Applications helfen dabei den Benutzer in einer komfortablen Webspace-Umgebung zu halten und den Inhalt auf eine einfache und verständliche Weise zu präsentieren.

Vorteile einer SPA:

- SPAs sind schnell, da Inhalte nur einmal in der Lebensdauer der Anwendung geladen werden müssen.
- Die Entwicklung ist sehr vereinfacht, da zwecks Anzeige auf Web-Seiten kein Code auf der Serverseite geschrieben werden muss.
- Es ist einfach eine Mobile-Anwendung zu entwickeln, da der Entwickler den gleichen Code für die Web-Anwendung und die Mobile-Anwendung verwenden kann.
- SPAs können jeden lokalen Speicher effizient verwenden. Es wird eine Anfrage an einen Server gesendet und alle Daten können dann in einem lokalen Speicher abgelegt und quasi »offline« verwendet werden.

Nachteile einer SPA:

- Durch den Aufbau von SPAs ist es sehr schwierig eine Optimierung für Suchmaschinen zu bewerkstelligen (Search Engine Optimization SEO). - Mit JavaScript haben die Suchmaschinen, wie Google und Bing, ihre Probleme und können somit nicht den Inhalt einer Seite komplett laden. Das führt dazu, dass die nicht geladenen Seite für die Suchmaschine nicht sichtbar ist und somit auch keine Sichtbarkeit für potentielle Benutzer gegeben ist.
- Ein Herunterladen des Client ist sehr langsam, da der Client durch viele Frameworks aufgebläht wird.

- Es ist notwendig, dass der Browser des Benutzers JavaScript aktiviert hat. Sollte JavaScript deaktiviert worden sein, wird die Anwendung nicht korrekt dargestellt.
- Im Vergleich zu »traditionellen« Webseiten ist eine SPA weniger sicher. Durch Cross-Site-Scripting (XSS) können Angreifer clientseitige Skripts, die häufig in JavaScript geschrieben werden, von anderen Benutzern in die Web-Anwendung einfügen und somit die Daten des Benutzers erfahren.

Bei einer MPA wird bei jedem Seitenaufruf der komplette Inhalt neu geladen. Aufgrund dessen sind Anwendungen dieses Typs größer als SPAs. Durch die große Menge darzustellenden Inhalts, hat eine MPA viele Level von UIs. Durch AJAX kann ermöglicht werden, dass nur bestimmte Bereiche der Anwendung erneuert werden müssen. Allerdings wird hierdurch die Komplexität der Anwendung gesteigert. Im Vergleich zu einer SPA, ist diese Komplexität schwieriger zu implementieren.

Vorteile einer MPA:

- Für eine große Anzahl an Web-Seiten und geschachtelte Menüs ist eine MPA die beste Variante.
- Die SEO kann leicht umgesetzt werden. - Da eine MPA aus mehreren Web-Seiten besteht, kann die Anwendung auf jeder Seite für ein Keyword optimiert werden.

Nachteile einer MPA:

- Es gibt keine Möglichkeit denselben Code auch für eine Mobile-Anwendung zu nutzen.
- Das Frontend und das Backend müssen zusammen entwickelt werden.

Jede Architektur hat ihre Vor- und Nachteile und ist optimiert für den Einsatz bestimmter Projekte. SPAs sind für den mobilen Einsatz und auf Schnelligkeit ausgelegt. Gleichzeitig besitzen sie allerdings eine schlechte SEO-Möglichkeit. MPAs sind besser geeignet für eine Webseite, die viele Seiten beinhalten. Als Beispiel sind ein Online-Shop oder Handelsbörsen zu nennen. Sie sind langsamer als SPAs, dafür haben sie einen Vorteil bei der SEO.

In der vorliegenden Arbeit wurde sich für die Verwendung einer SPA als Dashboard entschieden. Als Entwicklungssprache wurde die am weitesten verbreitet Sprache Angular verwendet.

4.4.2 Angular

Angular ist ein TypeScript-Framework, welches unter anderem von Google entwickelt wird. Es zielt auf die Entwicklung von Web-Anwendungen ab und legt großen Wert auf Struktur und Qualität. Der Fokus wird auf die Architektur, das Testen und isolierte Komponenten gelegt. Durch Methoden wie Dependency-Injection und einem ausgereiften Tooling ermöglicht Angular effiziente und wartbare Software-Entwicklung.

Dabei sind die wichtigsten Bausteine einer Angular-Applikation die Komponenten. Eine Angular-App gleicht dabei einer Baumstruktur. Am Anfang steht eine Root-Komponente, die sich durch angehängte Sub-Komponente zu einer Baumstruktur auffächert. Dabei gibt es einen Angular-Router, der den Status einer Anwendung in der URL abbilden kann und darauf basierend Komponentenbäume an bestimmten Stellen der Anwendung einhängen kann.

Die Komponenten bestehen aus einem Template, einer Komponentenklasse und einem Stylesheet. Dabei ist die Komponente für die View-Logik zuständig, die für den Betrieb der Benutzeroberfläche erforderlich ist. Das Template ist für die Strukturierung der Benutzeroberfläche und die Präsentation der Informationen zuständig. Die Business-Logik wird in einem normalen Fall in Serviceklassen ausgelagert und per Depency-Injection in die Komponenten eingebunden. Dies erhöht die Wiederverwendbarkeit der Programmteile und kommt der wichtigsten Anforderung an Komponente entgegen. Die Komponenten sollen so unabhängig wie möglich von einander sein. Listing (4.2) zeigt eine einfache Hallo-Welt-Komponente.

Listing 4.2: Ansicht einer "Hallo Welt"Komponente

```
1 @Component({
2  selector: 'app',
3  templateUrl: 'app.component.html'
4 })
5 class AppComponent {
6  public title = 'Hello World';
7 }
```

Module

Ein Module deklariert einen Kontext für die Kompilierung eines Satzes von Komponenten. Ein Modul kann seine Komponente mit zugehörigen Code wie zum Beispiel Services verknüpfen um Funktionseinheiten zu bilden. Jede Angular-Anwendung hat ein Root-Modul, normalerweise als AppModule bezeichnet. Das AppModule stellt den Bootstrap-Mechanismus bereit, der die Anwendung startet. Eine Anwendung beinhaltet in der Regel eine Vielzahl an funktionalen Modulen.

Data-Binding

Das Template ist für die Strukturierung der Benutzeroberfläche zuständig. Dabei kann es die HTML Elemente modifizieren, bevor sie dargestellt werden. Template-Direktiven stellen Programmierlogik bereit und verbinden HTML und die Daten der Komponenten mit dem Modell. Hierbei sind zwei Verbindungsarten zu unterscheiden:

Event binding Lässt die Anwendung auf Benutzereingaben reagieren, indem sie ihre Anwendungsdaten aktualisiert.

Property binding lässt Werte direkt im HTML interpolieren, die aus den Anwendungsdaten der Komponenten stammen.

Bevor eine Benutzeroberfläche angezeigt wird, evaluiert Angular die Direktiven und entscheidet über die Art der Verbindung zwischen den Daten und der Benutzeroberfläche. Dies wird als Zwei-Wege-Datenbindung (Two-Way-Data-Binding) bezeichnet.

Routing

Durch das Router-Modul von Angular, lässt sich ein Navigationspfad durch verschiedene Zustände der Anwendung und Benutzeroberflächen erzeugen. Es wurde nach den Standard Browser Navigationsprinzipien entwickelt.

- Es wird eine URL in die Adressleiste eingegeben und der Browser navigiert den Benutzer zu einer übereinstimmenden Seite.
- Durch das Klicken eines Links navigiert der Browser zu einer neuen Seite.
- Durch den Vor- und Zurückknopf des Browsers kann durch den Browser-Verlauf navigiert werden.

Abbildung (4.5) zeigt eine Übersicht der wichtigsten und hier vorgestellten Blöcke einer Angular Anwendung.

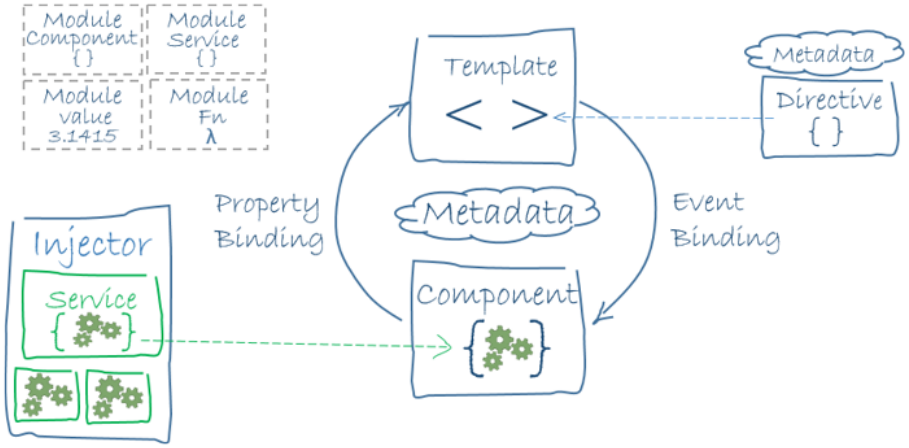

Abbildung 4.5: Übersicht der wichtigsten Blöcke einer Angular Anwendung [Angular Architecure, 2018]

Zusammengefasst bilden Komponente und Template eine Benutzeroberfläche. Ein Dekorator fügt die Metadaten zu einem Template hinzu. Durch das Two-Way-Data-Binding werden Daten im Template und in der Komponente modifiziert und angezeigt. Der Dependency-Injector bietet der Komponente bestimmte Services an. Dies kann eine Router-Service oder eine Datenverarbeitungslogik sein.

Die vorangegangene Ausführung ist nur eine kleiner Überblick [Angular Architecure, 2018] über umfangreichen Möglichkeiten von Angular. Für tiefere Informationen kann das Buch [Höller, 2017] verwendet werden.

4.4.3 Umsetzung der Anforderungen

Um die geforderten Anforderungen (vgl. Punkt 3.2) bestmöglich umsetzen zu können, werden neben den in Angular enthaltenen Funktionalitäten noch weitere benötigt.

Um eine reibungslose Kommunikation zwischen dem Dashboard und den anderen Komponenten der Portalseite und Integrationsumgebung zu ermöglichen, wurde die Entscheidung gefällt, dass Package *angular-oauth2-oidc* in die Angular Anwendung zu integrieren. Das *angular-oauth2-oidc* Package unterstützt den Entwickler, den Kommunikationsverlauf, unter Verwendung der Protokolle OAuth2 und OpenId Connect zu realisieren. Durch

die Verwendung einer SPA wurde das Kommunikationsverfahren als Implicit-Grant bestimmt.

Ein Ablauf der Kommunikation ist unter Punkt (4.7) zu finden. Abbildung (4.6) zeigt einen Oberflächenentwurf für das Dashboard um die bestimmten Anforderungen darzustellen. In dieser Abbildung ist davon ausgegangen worden, dass ein Benutzer in der Oberfläche angemeldet ist. Dies ist auf der rechten Seite durch den *UserName*, *das Profil* und *den Logout* dargestellt. Im Hauptfenster sollen die vorhandenen Services und die vorhandenen Dateien angezeigt werden.

Abbildung 4.6: Entwurf der Dashboard-Portalseite

4.5 File-Manager

Der File-Manager dient als Ansprechstation für die Verwaltung von Dateien und der Anzeige von Clients. Er muss durch den Auth-Server geschützt werden, so dass kein Fremdzugriff stattfinden kann. Hierzu muss eine API konzipiert werden, die die Anforderung an die Datenverwaltung und Sicherheit erfüllt. Im nächsten Abschnitt werden zwei Verfahren zur Gestaltung einer API vorgestellt und miteinander verglichen. Danach wird ein genauerer Blick auf den Entwurf der API geworfen.

4.5.1 Representational State Transfer (REST)

Der als REST bezeichnete Architekturansatz wurde erstmalig von Roy Fielding [Fielding, 2000] beschrieben. REST beschreibt wie verteilte Systeme (Webservices) miteinander kommunizieren können. Als RESTful werden Arten der Implementierung beschrieben die Standards wie HTTP/s und URIs verwenden. Ein Client sendet ein HTTP-Request an den Server, dieser bearbeitet den Request und nach Beendigung sendet der Server eine HTTP-Response mit dem angeforderten Inhalt an den Client zurück. Das Format des Inhaltes ist beim REST-Ansatz ungebunden. Das heißt, es können Formate wie JSON, XML oder Datei-Formate verwendet werden. Eine genaue Festlegung, wie REST entwickelt werden soll, gibt es nicht. Vielmehr gibt es einige Prinzipien [Tilkov, 2015, S. 26], denen einer REST Architektur folgen sollte:

Eindeutige Identifikation Die Ressourcen die ein Server anbietet, müssen eindeutig identifizierbar sein. Hierfür werden URIs verwendet. Als Beispiel:

- http://beispiel.de/user/12
- http://beispiel.de/produkte/
- http://beispiel.de/bestellung/artikel

Hypermedia Es kann vorkommen das Hyperlinks in einer Antwort eines Servers enthalten sind. Diesen kann von der Client-Anwendung gefolgt werden um zu einem bestimmten Produkt oder einem Benutzer zu gelangen. Die Client-Anwendung kann den Links einfach folgen, da diese Verknüpfungen anwendungsübergreifend funktionieren. Hierin liegt auch der entscheidende Vorteil von URIs. Es können Ressource in anderen Systemen referenziert werden.

Standardmethoden Jede Ressource unterstützt die gleichen Schnittstellen für die Anfragen eines Client. Diese werden durch die HTTP-Methoden (siehe Punkt 2.4) repräsentiert.

Repräsentation Über die bisher genannten Eigenschaften von REST können Clients eine Ressource über deren URI identifizieren und mit den bekannten HTTP-Methoden aufrufen. Ein Problem ist allerdings, woher weiß der Client wie er mit den zurückgegebenen Daten umgehen soll. Dieses Problem lässt sich lösen, indem die Clients das Rückgabeformat für die Kommunikation mit angeben. Durch HTTP-Content-Negotiation können Clients Repräsentationen in bestimmten Formaten abfragen.

Sendet ein Client ein Request an den Server, gibt er im *Accept-Header* an, welche Medienformate (MIME-Types) er erwartet.

Statuslose Kommunikation Konsequenterweise gibt es bei REST-konformen Anwendungen keinen Sitzungsstatus, der serverseitig über mehrere Client-Anfragen hinweg vorgehalten wird. Der Kommunikationszustand muss stattdessen im Client oder in der Repräsentation der Ressource gespeichert werden. Hierdurch wird die Kopplung zwischen Client und Server verringert. Diese Einschränkung bietet viele Vorteile: Es könnte zum Beispiel ein Server zwischen zwei Requests mit aktualisierter Software neu gestartet werden. Der Client bekommt davon nichts mit. Ebenso könnte zur Lastenverteilung ein Load-Balancer die Requests zu unterschiedlichen Serverinstanzen routen. Statuslose Kommunikation ist ebenfalls eine Voraussetzung dafür, dass jede URI als Einsprungspunkt dienen kann.

4.5.2 Simple Object Access Protocol (SOAP)

Bei SOAP werden alle HTTP-Requests per POST an einen einzigen Endpunkt gesendet. Ein vorgelagerter Dispatcher des Servers entscheidet dann, welche Funktion aufgerufen wird. SOAP ist eine Ansammlung von Protokollen. Bei der Entwicklung werden diese verwendet, um eigene Protokolle zu schreiben. Das Protokoll beschreibt einen genauen Aufbau einer Anfrage und einer Antwort. Sollen weitere Informationen geteilt werden, muss eine neue Schnittstelle implementiert werden. Um weiterhin die vorhandenen Clients lauffähig zu halten, muss ebenfalls ein neuer Webservice bereit gestellt werden. Das Format des HTTP-Response ist in XML zu versenden.

4.5.3 Unterschied REST und SOAP

REST hat durch die URIs einen breiten Adressraum, wobei SOAP nur eine Adresse nutzt um Anfragen zu verarbeiten. Das XML Schema bei SOAP ist eine sehr umfangreiche Sprache. Diese macht es schwierig das richtige Konstrukt zu identifizieren und ein Datenmodel so auszudrücken, dass es vollständig von allen SOAP Implementierungen unterstützt wird [Pautasso u. a., 2008]. Bei REST steht die Ressource im Vordergrund und einen Dispatcher gibt es im eigentlichen Sinne nicht.

4.5.4 Bestimmung des API-Ansatzes und der Entwicklungsumgebung

Als Verfahren für die Gestaltung der API wurde sich für den REST-Ansatz entschieden. Dies ist darin begründet, dass Ressourcen in der Darstellung geändert oder hinzugefügt werden können, ohne den Client zu ändern zu müssen. Der Client muss lediglich nur wissen über welche URI die neue bzw. geänderte Ressource erreichbar ist. Des Weiteren basiert der Auth-Server ebenfalls auf einer REST-Architektur, was die Kommunikation der Komponenten unterstützt.

4.5.4.1 Entwicklungsumgebung

PHP ist eine weitverbreitete und für den allgemeinen Gebrauch bestimmte Open-Source-Skriptsprache. Sie ist speziell für die Webprogrammierung geeignet und kann in HTML eingebettet werden. Damit eine Anwendung nicht aus unzähligen HTML-Anweisungen besteht, kann PHP genutzt werden um die HTML-Anweisung auf ein Minimum zu reduzieren. PHP unterscheidet sich von clientseitigen Sprachen wie JavaScript dadurch, dass der Code auf dem Server ausgeführt wird und die HTML-Ausgaben generiert. Diese werde dann an den Client geendet. Der Client erhält nur das Ergebnis der PHP-Skriptausführung und weiß daher nicht wie der eigentliche Code aussieht. Durch die weite Verbreitung und gute Dokumentation von PHP wurde sich entschieden, PHP als Basis der File-Manager API zu verwenden.

Framework

Um eine REST-API zu entwerfen, wurde sich beim Entwurf ebenfalls Gedanken um die bestmögliche Unterstützung bei der Realisierung gemacht. Hierfür wurde die große Anzahl von verfügbaren Frameworks für PHP sondiert. Dadurch sind die Frameworks Phalcon [Phalcon, 2017] und Symfony [SensioLabs, 2017] in die engere Auswahl gekommen. Sowohl Phalcon als auch Symfony verfolgen das Konzept der Model-View-Controller-Architektur (MVC). Phalcon ist nicht wie die meisten PHP-Frameworks in PHP selbst, sondern in der Programmiersprache *C* geschrieben. Dadurch konnte eine Optimierung bei der Ausführungsgeschwindigkeit und Ressourcennutzung erlangt werden. Diese Vorteile sind auch der Grund warum sich für das Phalcon-Framework, anstatt Symfony entschieden wurde.

Bei einer Implementierung einer REST-API ist es nicht nötig eine Web-Oberfläche zu entwickeln. Hierfür bietet Phalcon die Möglichkeit eine »entschlackte« Anwendung mit minimalen PHP-Code zu implementieren.

4.5.5 Entwurf der API

Beim Entwurf, mit Bezug auf die Anforderungen (gem. Punkt 3.2), haben sich fünf Hauptprobleme ergeben, die eine REST-API umsetzen muss.

Request Die REST-API muss in der Lage sein Requests zu verarbeiten.

Response Die REST-API muss nach einem Request eine Antwort liefern.

Authentifizierung Bevor ein Request bearbeitet und beantwortet wird, muss eine Authentifizierung erfolgen.

CORS In der lokalen Entwicklung, müssen Webservices erlaubt werden Cross-Origin-Requests durchzuführen.

Datenverarbeitung Die REST-API muss in der Lage sein Daten zu verarbeiten und anzubieten.

In den folgenden Unterkapiteln werden die Entwurfsmuster zur Lösung dieser Probleme erörtert.

4.5.5.1 Request

Damit Webservices einen Request an eine Schnittstelle der REST-API senden kann, müssen Schnittstellen öffentlich gemacht und definiert werden. Um die spezifizierten Anforderungen für Benutzer und andere Microservices umzusetzen, werden URIs eingesetzt. Diese dienen als Schnittstellen um eintreffende Requests zu verarbeiten.

Die Tabelle (4.1) zeigt eine Übersicht der entworfenen URIs. Dabei sind die Wörter, die von Anführungszeichen umschlossen sind, Platzhalter. Diese werden bei einem Request gegen entsprechende Daten ausgetauscht.

Über die beiden ersten URIs, können Listen der verfügbaren Clients angefragt werden. Ein Client kann Dateien unter unterschiedlichen »tags« einlagern. Bei der URI

HTTP-Methode	URI
GET	http://.../client/list
GET	http://.../client/tag/»tag-eines-clients«
POST	http://.../file/add/»file-name«
GET	http://.../file/download/»file-name«/»file-version«/»datum«
POST	http://.../file/delete/»tag-eines-clients«
GET	http://.../file/list
GET	http://.../file/list/»version«
GET	http://.../file/list/»year«/»moth«/»day«
GET	http://.../file/list/»tag«

Tabelle 4.1: Definierte URIs der API

»http://.../client/tag/tag-eines-clients«, kann ein »tag« am Ende eingefügt werden. Als Antwort wird der zugehörige Client ausgegeben.

Die erste HTTP-Methode POST ist für den Request einer Dateispeicherung zuständig. Über die URI »download« können Dateien heruntergeladen und zur Bearbeitung gespeichert werden. Bei der URI für das Löschen einer Datei (.../file/delte/»tag-eines-clients«) muss in dem mitgesendeten Body des Requests noch der Dateiname, die Version und das Datum angegeben werden. Über die URIs ».../file/list/...« kann eine Liste für die vorhandenen Dateien angefordert werden. Als Optionen können hier die Versionsnummer, das Datum oder ein bestimmter »tag« ausgewählt werden.

Bei der Umsetzung ist darauf zu achten, dass nur die diejenigen Clients oder Dateien ausgegeben werden, die auch zu dem angemeldeten Benutzer des anfragenden Webservices gehören. Bei den Vorgängen des Hinzufügens und Herunterladens ist ebenfalls darauf zu achten.

Nachdem eine URI per Request angesprochen wurde, kommuniziert die API mit der Datenbank in der File-Persistence (siehe Punkt 4.6) und je nach Befehl können Daten auf die Persistence gemappt werden.

4.5.5.2 Response

Nachdem eine Request eingegangen ist und eine Bearbeitung der Anfrage stattgefunden hat, ist es erforderlich das Ergebnis in eine Response-Nachricht zu verpacken und an den anfragenden Webservice zurückzusenden. Die Nachricht wird per HTTP-Protokoll übertragen. Standardmäßig wird als Format des Responses JSON verwendet. Eine Besonderheit ist in dem Fall zu beachten, dass eine Datei angefordert wird. Hier wird zwar auch das HTTP-Protokoll verwendet, allerdings wird im *Header* deutlich gemacht, um welches Format es sich bei der Datei handelt. Dieser Vorgang ist nötig, damit der Webservice die Datei, auf die richtige Art und Weise einordnen kann.

4.5.5.3 Authentifizierung

Es muss sichergestellt werden, dass die API vor der Außenwelt geschützt wird. Es dürfen keine unerlaubten Webservices Zugriff auf die Funktionalitäten der API erhalten. Um dies sicherzustellen, muss jeder Request auf seine Berechtigung hin geprüft werden. Die Überprüfung ist eine der Aufgaben des Auth-Servers. Dieser bietet eine Schnittstelle (Endpunkt) (vgl. Punkt 4.2.3.2) für diese Art der Autorisierung an. Ein Request stellt eine Anfrage an eine URI und übermittelt in seinem *Header* einen Authentifizierungscode. Die API wendet sich mit diesem Code an den Auth-Server und fragt, ob die Autorisierung genehmigt wird. Ist die Antwort erfolgreich, führt die API die geforderte Anfrage aus und sendet eine Response-Nachricht zurück.

4.5.5.4 Cross Origin Resource Sharing

Cross Origin Resource Sharing (CORS) ist ein Mechanismus um Clients Cross Origin Request zu erlauben und wird solange das System nicht veröffentlicht wurde genutzt. Cross Origin Request beschreibt eine Funktion mit der eine Webseite auf Daten einer anderen Domain zugreifen kann.CORS ermöglicht, dass Webbrowser domainübergreifend verschiedene Webanwendungen in einer Oberfläche kombinieren kann. Diese Aufrufe werden in der Regel durch moderne Browser wie FireFox oder Chrome durch die Same Origin Policy (SOP) verhindert. Durch CORS kann die SOP in bestimmten Fällen umgangen werden. Um dies zu tun, findet zwischen dem CORS der API und dem Client ein Informationsaustausch statt. Diese Kommunikation zwischen den Parteien, wird durch HTTP-Header ausgeführt. Bevor der eigentliche Request an die API gesendet wird,

erfolgt ein Preflight-Request. Der Preflight-Request wird mit CORS-Headern, in denen Informationen über den eigentlichen Request stehen, an die API gesendet. Dies geschieht, um zu erfahren, ob der eigentliche Request an die API gesendet werden darf. Ist die Antwort positiv, kann der Client seinen eigentlichen Request versenden. Ansonsten wird der Request unterbunden.

4.6 File-Persistence

Die File-Persistence, muss zum einen die Möglichkeit haben eine Datenbank für die Verwaltung von Dateien zu betreiben und zum anderen Speicherplatz für die Dateien zur Verfügung stellen. Die Bereitstellung dieser Möglichkeiten muss permanent sein, da es ansonsten zu Verlusten der Dateien oder der Datenbank kommen kann. Für die Datenbankmodellierung soll wie bei der Auth-Persistence (siehe Punkt 4.3) auf MySQL gesetzt werden. Um die geforderten Funktionalitäten an die Dateienverwaltung umzusetzen, wurde die Datenbankstruktur, die in Abbildung (4.7) zu sehen ist, entworfen.

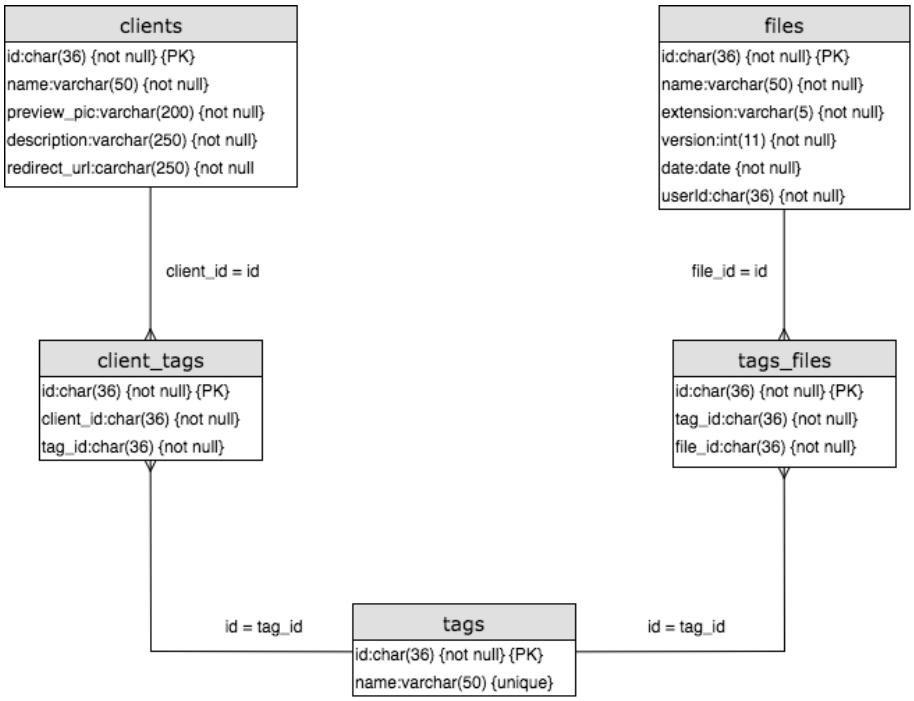

Abbildung 4.7: Übersicht der Datenbank Architektur

Es wurden die Tabellen »clients«, »tags«, und »files« definiert. Die Primary-Keys (PK) sind jeweils die IDs der einzelnen Tabellen. Die IDs müssen eindeutig identifizierbar sein. Das soll über Universally Unique Identitfier (UUID) geschehen. Die Beziehung zwischen den separaten Tabellen ist als Many-To-Many Beziehung definiert. Diese Beziehungen wurden gewählt, da ein Client viele »tags« haben kann und viele »tags« einem Client zugeschrieben werden können. Ebenso verhält sich die Beziehung von der Tabelle »tags« zu der Tabelle »files«. Um eine Many-To-Many Beziehung in einer Datenbank abzubilden, führt man sogenannte Junction-Tabellen ein. Eine Junction-Tabelle verbindet zwei oder mehrere Tabellen in eine neue Tabelle und referenziert deren PKs. In der Abbildung (4.7) sind das die Tabellen »client_tags« und »tags_files«. Die Junction-Tabellen werden mit jeweils einer One-To-Many Beziehung zu den Ursprungstabellen versehen. Das Gesamtbild kann als »clients« Many-To-Many »tags« Many-To-Many »files« Beziehung aufgefasst werden.

4.7 Interprozesskommunikation

Nach der Untersuchung und dem Entwerfen aller benötigten Komponenten müssen diese miteinander kommunizieren. Die Kommunikation zwischen allen Komponenten wird anhand des Beispiels in Abbildung (4.8) dargestellt. Für dieses Szenario wurde davon ausgegangen, dass das Authentifizierungsverfahren mit dem Implicit-Grant geschieht und ein Referenz-Token Anwendung findet. Nicht abgebildet wurde der Request zum Discovery-Endpunkt. Dieser Schritt geschieht noch vor dem Request zum Authorize-Endpunkt.

Im folgenden werden die einzelnen Schritte des Kommunikationsvorganges genauer erläutert. Dabei findet die komplette Kommunikation per HTTP-Protokoll und im JSON-Format statt.

1 Der Benutzer greift auf das Dashboard zu.

2 Das Dashboard wird aktiv und sendet ein Request an den Authorize-Endpunkt des Auth-Server. In dem Request müssen Parameter des Dashboards angegeben werden. Parameter sind zum Beispiel: »client_id«, »redirect_url«, »response_type« und Scopes.

3 Nach Erhalt des Requests, sendet der Auth-Server per ORM ein Suchbefehl an die Auth-Persistence. In diesem Befehl sind die entsprechenden Daten des Requests vom Dashboard.

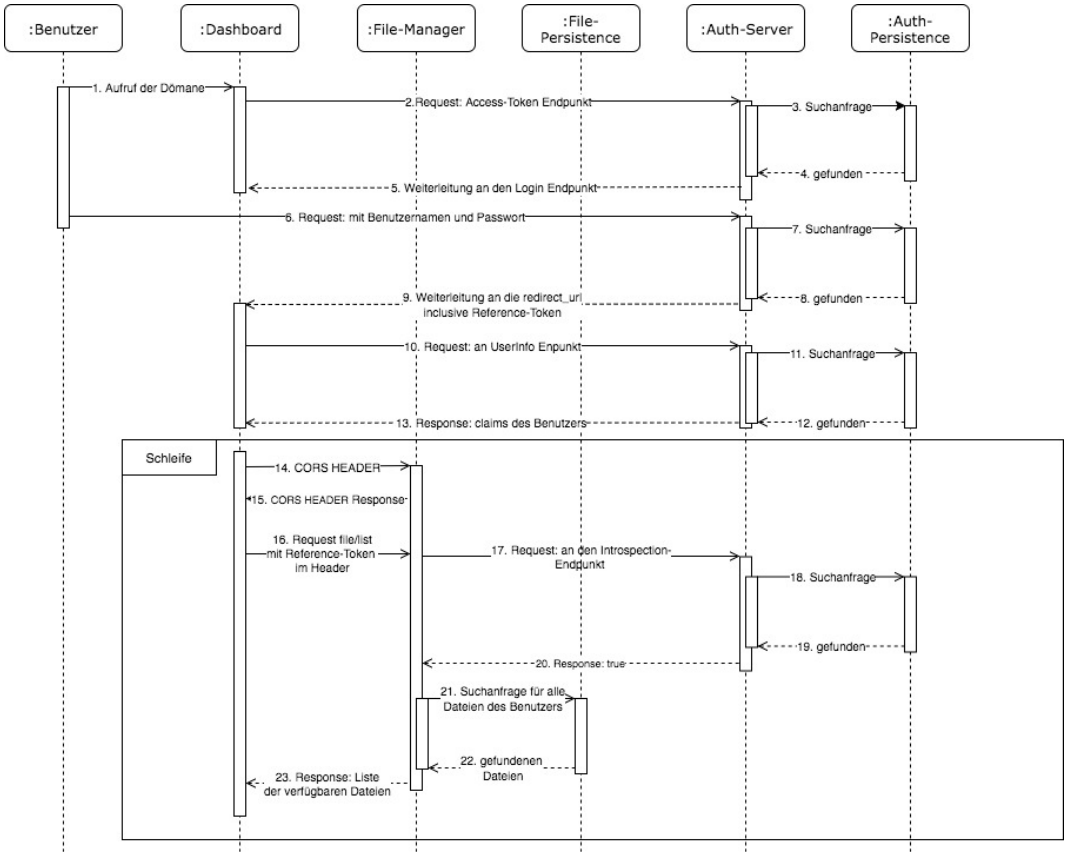

Abbildung 4.8: Sequenzdiagramm der Interprozesskommunikation

4 Bei Erfolg bekommt der Auth-Server ein Objekt des gefundenen Eintrages zurück und kann daran bestimmen, dass die Anfrage des Dashboards erlaubt ist.

5 Nach erfolgreicher Erkennung des Dashboards, leitet der Auth-Server das Dashboard an den Login-Endpunkt weiter.

6 Der Benutzer gibt seine Login-Daten ein.

7,8 Sind die gleichen Schritte wie bei 3-4. Nur das hier nach der Kombination aus Benutzername und Passwort des Benutzers gesucht wird.

9 Nach erfolgreicher Auffindung und Überprüfung der passenden Benutzernamen- und Passwort-Kombination wird der Benutzer zurück zum Dashboard geleitet und erhält einen Referenz-Code.

10 Das Dashboard sendet ein Request inklusive des Referenz-Token an den UserInfo-Endpunkt des Auth-Servers, um Informationen über den Benutzer zu erlangen.

11, 12 Analoges Vorgehen wie bei Schritt 3-4 mit dem Unterschied, dass die Claims des Benutzers abgefragt werden.

13 Der Auth-Server sendet ein Response an das Dashboard mit den vorher gefundenen und erlaubten Claims des Benutzers.

14 Das Dashboard möchte eine URI des File-Manager ansprechen und eine Liste aller verfügbaren Daten erhalten. Hierfür wird im Preflight-Request ein *Header* mit der Anfrage nach OPTIONS und die HTTP-Methode (in diesem Fall GET wegen der »...file/list« URI) an den File-Manager geschickt.

15 Als Antwort erhält das Dashboard durch die CORS-Funktionalität des File-Managers, ein Response mit allen erlaubten HTTP-Methoden im *Header*.

16 Nachdem festgestellt wurde, dass die GET-Methode erlaubt ist, wird der eigentlich Request an die URI ».../file/list« gesendet. Dabei befindet sich im *Header* der Referenz-Token.

17 Bevor der Request bearbeitet wird, übermittelt der File-Manager den Referenz-Token per Request an den Auth-Server. Ebenso wird ein Identifikationsname und ein geheimes Passwort übertragen.

18, 19 Das Vorgehen ist mit den Schritten 3-4 zu vergleichen. Es wird überprüft, ob das Referenz-Token aktuell ist und es dem Dashboard erlaubt ist diese Anfrage an den File-Manager zu stellen.

20 Als Response erhält der File-Manager ein »true « und die Benutzer- sowie die Client-ID des anfragenden Clients.

21, 22 Prinzipiell findet hier die gleiche Kommunikation wie bei Schritt 3-4 statt. Es wird eine Liste der Dateien des Benutzers unter Verwendung der Benutzer-ID, per ORM in der File-Persistence gesucht.

23 Der File-Manager sendet eine Response-Nachricht mit einer Liste der gefundenen Dateien an das Dashboard. Dieses kann nun eine Liste der vorhandenen Dateien des Benutzers anzeigen.

Die Schritte, die sich in dem Kasten mit der Bezeichnung Schleife befinden, können gegen andere Requests vom Dashboard an den File-Manager ausgetauscht werden. Es ändern sich nur die angesprochene URI und die HTTP-Methode. Der danach folgende Ablauf der Kommunikation bleibt der Gleiche.

4.8 Netflix Open Source Software (OSS)

Das Streaming-Portal Netflix hat Ihr System 2012 in hunderte Microservices aufgeteilt, um dem immer noch wachsenden Traffic gerecht zu werden. Damit die Microservices überwacht und die hohen Anforderungen an Ausfallsicherheit und Skalierbarkeit erfüllt werden können, hat Netflix technische Komponenten und Bibliotheken entwickelt. Die technischen Komponenten und Bibliotheken wurden im Netflix OSS zusammengefasst. Das Netflix OSS bietet Unterstützung bei der Service-Registrierung, der Interprozesskommunikation, dem Monitoring, erhöht die Ausfallsicherheit usw.

Mit Bezug auf das Monitoring für die Portalseite und Integrationsumgebung wurden zuerst Überlegungen angestrebt, diese mit dem Netflix OSS umzusetzen. Bei der Planung des Entwurfes wurde festgestellt, dass der Aufwand für die Umsetzung des Monitoring, den Umfang einer Bachelorthesis übersteigen würde. Deswegen wurde davon Abstand genommen.

Ebenso wurde das Triggern, d.h. einem Webservices bekannt manchen, dass Dateien für ihn bereit liegen, und ein Scheduler für Dateien, ein Ablaufplan für den Zugriff auf Dateien, wegen des Umfangs der Arbeit nicht weiter verfolgt.

4.9 Dockerize

Durch die Vorteile eines Docker-Container (vgl. Punkt 2.5) wurde sich dazu entschlossen, die vorher bestimmten und separierten Funktionalitäten der Portalseite und Integrationsumgebung in unabhängige Docker-Container zu verpacken. Hierzu folgt eine kleine Darstellung, welche Images in den einzelnen Containern integriert werden müssen.

Auth-Server

Der Auth-Server soll, wie unter (Punkt 4.2.3) beschrieben, mit der Authentifikation-Lösung des Identity-Servers laufen. Der Identity-Server läuft auf der Basis von ASP.NET Core. Deswegen muss der Docker-Container mit einem Image der ASP.NET Core Umgebung bestückt werden.

Dashboard

Als Grundbasis des Docker-Containers wird Node.js verwendet. Node.js ist eine Laufzeitumgebung für Javascript-basierte Sprachen und wird für die Verwendung von Angular benötigt. Als Erweiterung wird Angular in die Node.js-Umgebung installiert.

File-Manager

Für das Image des File-Manager wurde Apache gewählt. Apache ist die Grundlage für die Ausführung von PHP. Wie in dem Entwurf des File-Managers (unter Punkt 4.5) beschrieben, wurde PHP gewählt und das Phalcon-Framework als Entwicklungsumgebung bestimmt. Somit muss in das Apache Image ebenfalls Phalcon installiert werden.

Auth-Persistence und File-Persistence

Die Auth-Persistence und die File-Persistence sollen beide mit MySQL betrieben werden. Für eine einfachere Entwicklung können beide Persistences in einem Docker-Container betrieben werden. Spätestens beim Deploy-Vorgang sollten beiden Funktionalitäten in separate Container geteilt werden. Als Image des Containers wird direkt ein MySQL-Server verwendet.

Docker-Compose-File

Um die unverbundenen Docker-Container zu verbinden wird ein Docker-Compose-File (siehe Punkt 2.5.3) genutzt. In diesem File werden unter anderem alle bekannten Docker-Container eingetragen und Ports für die Kommunikation bestimmt. Der Nutzen dieser Datei ist, dass man alle Container über einen Befehl starten kann.

5 Implementierung des Systems

Basierend auf dem vorangegangenen Kapitel, wird in diesem Kapitel die Implementierung des Systems umgesetzt. Hierzu werden die deklarierten Komponente im Einzelnen beschrieben und Auszüge aus der Implementierung dargestellt. Aus zeitbedingten Gründen wurde die Implementierung nur für den lokalen Gebrauch umgesetzt. Bei einer Veröffentlichung des Systems müssen kleinere Anpassung an der Struktur und den Adressen, die zur Kommunikation dienen, umgesetzt werden.

5.1 Auth-Server

Der Auth-Server ist, wie unter (Punkt 4.2) beschrieben, mit ASP.NET Core und dem Identity Server Framework umgesetzt worden. Um alle Punkte des Entwurfes abzudecken, wurden noch weitere Abhängigkeiten in das Projekt integriert. Die Abhängigkeiten werden im Folgenden kurz beschrieben und ihre Hauptaufgabe benannt.

IdentityServer4.AccessTokenValidation Diese Abhängigkeit erlaubt den Einsatz und die Validierung von Referenz-Token und JSON Web Token (JWT). Sie wird benötigt, um ein Referenz-Token am Introspection-Endpunkt zu validieren.

IdentityServer4.AspNetIdentity Die Hauptaufgabe dieser Abhängigkeit ist es, die Identität des Benutzer in einer Datenbank zu speichern und zu verwalten.

IdentityServer4.EntityFramework Dieses Framework ist für Unterstützung der dauerhaften Speicherung von Konfigurationsdaten in einer Datenbank zuständig. Standardmäßig wird SQL für die Umsetzung und Kommunikation der Datenbank verwendet.

Pomelo.EntityFrameworkCore.MySql Diese Erweiterung fügt dem EntityFramework die Möglichkeit hinzu eine Datenbank mit MySQL zu verwenden.

Der Aufbau des Auth-Servers entspricht einer üblichen ASP.NET Core MVC-Anwendung. MVC steht hier für eine Model-View-Controller-Architektur. Diese Architektur gehört, heutzutage zum Standard einer modernen Web-Anwendung und beschreibt die Zusammenhänge der Anwendungslogik und der Darstellung der Web-Oberfläche. Für einen detaillierteren Blick sei das Buch *Moderne Web-Anwendungen mit ASP.NET MVC und JavaScript* von Manfred Steyer [Steyer u. a., 2014, S. 15 u.f.] empfohlen. Im Folgenden wird ein Blick auf wichtigsten Klassen der Anwendung geworfen.

5.1.1 Startup.cs

Die »Startup.cs«-Klasse wird zur Laufzeit als erstes ausgeführt und dient zur Konfiguration von genutzten Diensten und der Anforderungspipeline einer ASP.NET Core Anwendung. Das Listing (5.1) zeigt einen Auszug aus der »Startup.cs« des Auth-Servers. Zu erkennen ist hier die Konfiguration des Introspection-Endpunktes.

Listing 5.1: Auszug aus der Startup.cs Klasse

```
1  services.AddAuthentication(IdentityServerAuthenticationDefaults.
       AuthenticationScheme)
2                  .AddIdentityServerAuthentication(options =>
3                  {
4                      options.Authority = "http://localhost:4200";
5                      options.ApiName = "FileServerApi";
6                      options.ApiSecret = "FileServerSecret";
7                      options.EnableCaching = true;
8                      options.CacheDuration = TimeSpan.FromMinutes
                           (10);
9
10                     options.SupportedTokens = SupportedTokens.Both;
11                 });
```

Die Zeilen 4-10 beschreiben hierbei die Konfiguration, welche die geschützte API, in ihrer Anfrage an den Introspection-Endpunkt beinhalten muss. Zur Identifizierung der API müssen dessen Name und ein API-Secret übertragen werden. Zeile 10 gibt an, in welcher Art Tokens validiert werden können. In diesem Fall sind es Referenz-Token und JWT.

Ebenso werden unter anderem in der »Startup.cs« die Einstellungen für die Datenbank und die Verwendung von Identity konfiguriert.

5.1.2 Config.cs

Die »Config.cs« dient als Konfigurationsdatei für die Ressourcen und Clients. Hier werden unter anderem festgelegt, welche OpenID Connect-Profile genutzt werden sollen, wie genau die Konfiguration der APIs aussehen soll und welchen Clients es erlaubt ist, Tokens anzufordern oder auf welche API sie zugreifen dürfen. Die hier getätigten Einstellungen werden beim Systemstart des Auth-Servers in die zuvor erstellte Datenbank geladen. Vergleiche hierzu Listing (5.2).

Listing 5.2: API-Konfiguration in der Config.cs

```
new ApiResource
        {
            Name = "FileServerApi",

            // secret for using introspection endpoint
            ApiSecrets =
            {
                new Secret("FileServerSecret".Sha256())
            },

            // include the following using claims in access token (
                in addition to subject id)
            UserClaims = { JwtClaimTypes.Name, JwtClaimTypes.Email
                },

            // Scopes for the API
            Scopes =
            {
                new Scope()
                {
                    Name = "FileServer.full_access",
                    DisplayName = "Full access to FileServer",
                }
            }
        }
```

In Listing (5.2) ist die Implementierung einer API-Ressource zu sehen. Zuerst wird ein eindeutig identifizierbarer Name zugeordnet und ein API-Secret vergeben. Dieses ist nach

dem Sha256-Verfahren verschlüsselt. Dieses Secret muss unter anderem bei der Kommunikation über den Introspection-Endpunkt angegeben werden. Des Weiteren werden die Claims angegeben, die bei der Erstellung eines Access-Tokens mit eingebunden werden sollen. In diesem Fall der Name des Benutzers und die Email-Adresse. Als letztes werden die Scopes (Berechtigungen) für einen Client festgelegt.

Alle Clients die Zugriff auf den Auth-Server bekommen sollen, werden ebenfalls in der »Config.cs«-Klasse deklariert. Hierfür werden ein eindeutiger Client-Name und eine eindeutige Client-ID vergeben. Als weiterer Schritt wird der Typ des Access-Tokens bestimmt und die Art des Grants festgelegt. Ebenso müssen hier die unterschiedlichen erlaubten Rediredct-URIs festgelegten werden. Zum Schluss wird noch definiert, welche Berechtigungen dem jeweilige Client gewährt werden.

5.1.3 Weitere Klassen

Als weitere Klassen, die additiv zum Projekt des Auth-Servers hinzugefügt wurden, sind die »ApplicationRole« -, »ApplicationUser.cs«- und die »ProfileService.cs« -Klasse zu nennen. Die beiden erstgenannten Klassen sind für den Identity-Teil des Identity-Servers zuständig. In der »ApplicationRole.cs« -Klasse werden die Definitionen der Rollen, die ein Benutzer haben kann, erstellt. Diese Klasse erbt ihre Funktionalitäten von dem Identity-Framework. In ähnlicher Weise ist die Klasse »ApplicationUser.cs« zu verstehen. Hier kann festgelegt werden, wie ein Benutzer definiert werden soll. Sie erbt ebenfalls ihr Funktionalitäten vom Identity-Framework.

Als dritte Klasse ist die »ProfilService.cs«-Klasse zu nennen. Sie erbt vom IProfilService des Identity-Server Frameworks und wird genutzt, um Informationen über den Benutzer an den Endpunkten UserInfo und Introspection zur Verfügung zu stellen. In dieser Klasse wird durch ORM Claims eines Benutzers aus der Datenbank geladen, und an den entsprechenden Endpunkt bereitgestellt.

5.2 Auth-Persistence

Bei dem Entwurf der Auth-Persistence wurde beschrieben, dass das Entity-Framework genutzt werden soll. Dieses wurde in der Implementierung berücksichtigt. Durch die Verwendung von Code First-Migration wird beim Systemstart des Auth-Server ein vor-

ab definiertes Datenbankschema aufgebaut. Diese Schemen sind in dem Ordner Data des Identity-Servers zu finden. Um das Datenbankschema aufzubauen werden zwei vordefinierte Speicher des Entity-Frameworks verwendet. Diese werden in der »Startup.cs« durch AddConfigurationStore und AddOperationalStrore deklariert. Für die Clients und Ressourcen ist die AddConfgurationStore-Konfiguration zuständig während die AddOperationalStore-Konfiguration sich für das Schema der Tokens, Codes, und Konsens verantwortlich zeichnet. Damit eine Kommunikation zwischen dem Identity-Server und den Datenbanken stattfinden kann, werden sogenannte DbContext-Klassen verwendet. Für jede Datenbank ist eine DbContext-Klasse vorhanden. Diese Klassen werden für die entsprechende Datenbanken verknüpft. Dadurch weiß die Anwendung, wann welche Datenbank angesprochen werden muss. Durch die Klasse »SeedData.cs« werden alle benötigten Datenbankschemen erstellt, die festgelegten Deklarationen aus der »Config.cs« gelesen und in die erstellte Datenbank eingetragen. Des Weiteren wird hier noch ein Admin-Profil angelegt und ebenfalls in die Datenbank als Benutzer eingetragen.

5.3 Dashboard

Das Dashboard wurde in Angular implementiert. Im Source-Ordner der Angular Anwendung sind Unterordner für Komponente, *models*, *services* und *shared* zu finden. Ebenso befinden sich in dem Ordner alle benötigten Klassen für die Hauptkomponente der Angular-Anwendung. Diese Klassen können an dem vorangestellten »app« erkannt werden. Zu diesen Klassen gehören unter anderem die »app-rounting.module.ts«, die »app.component.ts«, »app.module.ts« und die »auth.config.ts« Klasse. Abbildung (5.1) zeigt einen Ausschnitt aus dem Aufbau der Angular-Anwendung. Im Folgenden wird auf die wichtigsten Klassen eingegangen. Für die Kommunikation zwischen Dashboard und Auth-Server wurde das Package »angular-oauth2-oidc« zur Anwendung hinzugefügt.

5.3.1 auth.config.ts

Durch die »auth.config.ts«-Datei werden Einstellungen für die Kommunikation mit dem Auth-Server getroffen. Hier wird angeben, unter welcher Adresse der Auth-Server zu finden ist, wohin nach erfolgreichem Login weitergeleitet werden soll, die Client-ID, die Scopes, und der Response-Token. Für eine erfolgreiche Kommunikation müssen diese Angaben mit denen im Auth-Server übereinstimmen.

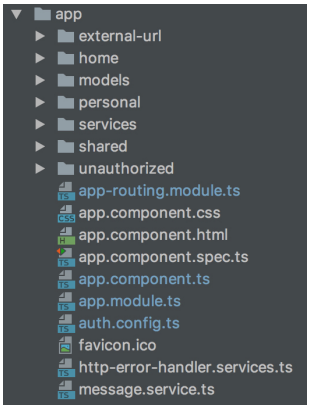

Abbildung 5.1: Ausschnitt des Aufbaus des Dashboards

5.3.2 app.component.ts

In der Hauptkomponente »app.component.ts« findet der komplette Kommunikationsvorgang mit dem Auth-Server statt (vgl. 4.8, Schritte 1-13). Hier wird unter anderem der Grant-Typ festgelegt und die Reihenfolge der Requests an die Endpunkte des Auth-Servers.

Discovery-Endpunkt Als erstes wird der Discovery-Endpunkt des Auth-Servers angesprochen, um unter anderem mitgeteilt zu bekommen, welche Endpunkte zur Verfügung stehen.

Login Nach dem Discovery-Endpunkt wird nachgeprüft, ob der Benutzer schon eingeloggt ist. Ist dies nicht der Fall, wird dementsprechend ein Request an den Access-Token-Endpunkt des Auth-Servers gesendet.

UserProfil Nach erfolgreichem Login-Vorgang, wird ein Request mit dem erhaltenen Referenz-Token an den UserInfo-Endpunkt gesendet, damit die Benutzerdaten empfangen und initialisiert werden können.

Redirect Zum Abschluss wird zur mitgeteilten Redirect-URI weitergeleitet.

Optional können auch ein *Silent-Refresh* und *session_ check* hier durchgeführt werden.

5.3.3 Services

Als Services stehen »auth.services.ts« und »fileServer.services.ts« zur Verfügung.

auth.services.ts

In der »auth.services.ts« wurden Funktionalitäten implementiert, die den Benutzer und das Verhalten der aktuellen Session betrifft. Hier stehen Funktionen für das Ausloggen, um ein Session-Refresh durchzuführen oder zur Abfrage des aktuellen Zustandes des Benutzers zur Verfügung. Ebenfalls werden hier die Informationsdaten des Benutzers abgefragt und in einem User-Model gespeichert.

fileServer.services.ts

Die »fileServer.services.ts« wurde entwickelt, um die Kommunikation mit dem File-Manager bereit zu stellen. In dieser Klasse wurden drei Funktionen umgesetzt. Die erste Funktion erstellt einen *Header* mit dem, vom Auth-Server bekommenden Referenz-Token. Diese Funktion wird bei allen anderen Funktionen für deren Requests verwendet. Des Weiteren sind die Funktionen »getClients()« und »getFiles()« implementiert worden, um jeweils eine Liste der verfügbaren Clients oder der verfügbaren Dateien eines Benutzers zu erhalten. Listing (5.3) zeigt die Umsetzung eines Requests für eine Liste von Dateien. Es wird die URI »file/list« des File-Managers mit einer GET-Methode angesprochen.

Listing 5.3: fileServer.services.ts: Request für eine Liste der verfügbaren Dateien

```
1 getFiles(): Observable<Files[]> {
2     return this.http.get<Files[]>(this.fileServerUrl + 'file/list',
          this.header())
3       .pipe(
4         catchError(this.handleError('getFiles', []))
5       );
6   }
```

5.3.4 Home-Komponente

Die Home-Komponente dient als Hauptseite der Angular-Anwendung. Nach erfolgreichem *Login* wird zur Home-Komponente weitergeleitet. Dadurch dient sie als Hauptansichtsseite. In der »home-component.ts« werden die Request-Funktionen des »fileServer.services.ts« aufgerufen und auf einen Response gewartet. Ist eine Liste mit Inhalt empfangen worden, werden diese durch das Data-Binding von Angular auf der Weboberfläche der Home-Komponente angezeigt.

5.4 File-Manager

Der File-Manager wurde in PHP mit dem Framework Phalcon implementiert. Hierbei wurde das Hauptaugenmerk auf die, im Entwurf (vgl. Punkt 4.5.4) des File-Managers herausgearbeiteten, Funktionalitäten gelegt. Abbildung (5.2) zeigt eine Übersicht des aktuellen Aufbau des File-Managers.

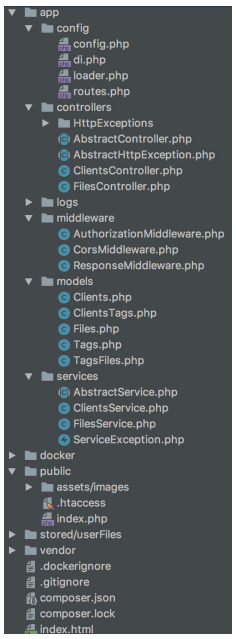

Abbildung 5.2: Übersicht über den Aufbau des File-Managers

Hierbei ist die in dem »public«-Ordner enthaltene »Index.php« die Konfigurationsdatei, in der alle Einstellungen, die die API-Anwendung betreffen, getätigt werden. Als Beispiel

wäre die typische »Micro()«-Deklaration einer Phalcon Anwendung bei der Entwicklung einer API zu nennen. Des Weiteren werden hier die Konfigurationsdateien im Ordner »config« eingelesen und die Middleware eingehängt. Sprich: Es werden alle Abhängigkeiten der Anwendungen hinzugefügt. Nachfolgend werden die Hauptordner und ihrer Funktionen beschrieben und ein Blick auf ausgewählte Dateien geworfen.

5.4.1 Config

In dem »Config«-Ordner befinden sich die nötigen Dateien, um die API nach den Anforderungen und dem daraus resultierenden Entwurf umzusetzen.

»config.php« In der »config.php« ist die Verbindung zur File-Persistence deklariert.

»di.php« »di« steht für Dependency Injection und beschreibt alle Abhängigkeiten, die in der erwähnten »index.php« in die Anwendung integriert werden müssen.

»loader.php« die »loader.php« ist dafür zuständig, der Anwendung mitzuteilen, wo sich die Pfade zu den einzelnen Unterordnern und deren Dateien befinden.

»routes.php« In der »routes.php« werden die URIs, wie unter (Punkt 4.5.5.1) bestimmt, definiert.

Ein Auszug aus der »routes.php«, ist in Listing (5.4) zu sehen. Über die in Phalcon definierte Funktionalität *Collection*, werden alle URIs zusammen gefasst, die mit dem Präfix */file* verbunden sind [Zeile 3]. Darüber hinaus wird in dieser Datei definiert, welche HTTP-Methode die URI haben soll. Beispielsweise ist in Zeile 4 eine HTTP-Methode mit POST deklariert worden. Da eine URI eindeutig identifizierbar sein muss, wurde »/add/flqq als weitere Adressierungsart hinzugefügt. Das »tag:[a-z]+« zeigt an, dass auf das »add« noch ein String mit Buchstaben kommen muss. Die Anweisung nach dem Komma in dieser Zeile zeigt an, welche Funktion in der Controller-Klasse ausgeführt werden soll, wenn ein Request an dieser URI eintrifft. Zum Schluss wird die definierte Collection von URIs in die Anwendung eingebunden [Zeile 15].

Listing 5.4: Auszug der URIs-Deklaration

```
1  $filesCollection = new \Phalcon\Mvc\Micro\Collection();
2  $filesCollection->setHandler('\App\Controllers\FilesController',
       true);
3  $filesCollection->setPrefix('/file');
4  $filesCollection->post('/add/{tag:[a-z]+}', 'addAction');
5
6  $filesCollection->get('/download/{name}/{version}/{year:([0-9]{8})}
       ', 'downloadAction');
7
8  $filesCollection->post('/delete/{tag:[a-z]+}', 'deleteAction');
9
10 $filesCollection->get('/list', 'getFilesAction');
11 $filesCollection->get('/list/{version:[0-9]+}', '
       getFilesVersionAction');
12 $filesCollection->get('/list/{year:([0-9]{4})}/{month:([0-9]{2})}/{
       day:([0-9]{2})}','getFilesDateAction');
13 $filesCollection->get('/list/{tag:[a-z]+}', 'getFilesTagAction');
14
15 $app->mount($filesCollection);
```

5.4.2 Controllers

Die Controller-Dateien werden ausgeführt, nachdem ein Request an einer URI empfangen wurde. Als Controller sind hier vor allem die »ClientsController.php« und die »FilesController.php« aufzuführen. Die »ClientsController« bearbeitet alle Requests, die über eine URI mit dem Präfix »client« angekommen sind. Die »FilesController.php« behandelt dazu analog alle Requests, die über ein URI mit dem Präfix »files« angekommen sind. In diesen Dateien findet eine Prüfung auf Korrektheit des Requests statt. So wird unter anderem geprüft, ob im Body alle benötigten Information vorhanden sind. Wenn nicht wird der Request abgelehnt. Ist die Überprüfung ohne Beanstandungen durchgeführt worden, werden alle nötige Informationen für die Bearbeitung des Requests an die entsprechenden Services weitergegeben. Die übrigen Controller-Klassen werden für die Fehlerbehandlung benötigt.

5.4.3 Services

Die Services sind für die Verarbeitung eines Requests zuständig. Auch hier sind die beiden Klassen »ClientsService.php« und »FilesService.php« näher zu betrachten. Eine Funktionalität der »FilesService.php«-Klasse wird durch die »FilesController.php«-Klasse aufgerufen und bekommt alle benötigten Daten übergeben. Die entsprechende Funktionalität beginnt mit der Bearbeitung ihrer Aufgabe. Diese Aufgabe kann, wie im Entwurf (vgl. Punkt 4.5.5.1) ermittelt, eine Abfrage über alle verfügbaren Dateien des Benutzers sein. Um eine erfolgreiche Bearbeitung zu gewährleisten, findet ein *mapping* zwischen dem File-Manager und der File-Persistence statt. Hierbei werden gegebenenfalls Daten abgefragt, oder neue Daten unter Einbeziehung der Benutzer-ID eingepflegt. Nach erfolgreicher Bearbeitung wird das Ergebnis zurück an die entsprechenden Controller geliefert und durch die Response-Middleware an den Request-Steller versendet.

5.4.4 Models

Für die Kommunikation vom File-Manager zur File-Persistence werden die Dateien des Models-Unterordners verwendet. Die Benennung und der Aufbau der Dateien entspricht der aus dem Entwurf der File-Persistence bereits bekannten Abbildung (4.7). In den Model-Dateien werden die Beziehungen zwischen den einzelnen Tabellen deklariert. Eine »ManytoMany«-Beziehung wird beispielsweise in der Datei »Clients.php« initialisiert. Listing (5.5) zeigt eine solche Beziehung. Dabei wird die »id« der »Clients.php«-Datei, mit der »Client_id« der Datei »ClientsTags.php« verknüpft. Durch die »tag_id« der »ClientsTags.php«-Datei, findet dann eine Verknüpfung zur »id« der »Tags.php«-Datei statt. So können einfach Beziehungen zwischen den einzelnen Tabellen hergestellt werden. Ein weiterer Vorteil ist, dass auf der File-Persistence nicht mehr kompliziert mit *Foreign-Keys* gearbeitet werden muss. In den anderen Dateien des Models-Ordners sehen die Beziehungen vergleichbar aus.

Listing 5.5: Many-To-Many-Beziehung in der »Clients.php«-Datei

```
1  public function initialize()
2      {
3          $this->hasManytoMany(
4              'id',
5              'App\Models\ClientsTags',
6              'client_id', 'tag_id',
7              'App\Models\Tags',
8              'id',
9              array(
10                 'alias'=>'tags',
11             )
12         );
13     }
```

5.4.5 Middleware

Eine Middleware kann vor oder nach der Bearbeitung eines Request definiert werden. So sind die beiden Middlewares »CorsMiddleware.php« und »AuthorizationMiddleware.php« vor der Bearbeitung eines Request und die »ResponseMiddleware.php« danach in die Anwendung integriert worden. Sobald ein Request an einer URI eintrifft, muss als erstes die CORS-Middleware reagieren. Diese wurden nach dem beschriebenen Entwurf (4.5.5.4) umgesetzt. Nachdem das CORS-Verfahren erfolgreich überstanden wurde, kann der eigentliche Request gesendet werden. Bei diesem Request folgt die Ausführung der Authorization-Middleware. In dieser Middleware wird überprüft, ob sich im *Header* des Request ein Referenz-Token befindet, welches durch den Identifikator »auth-token« deklariert werden muss. Ist ein Referenz-Token gefunden worden, wird eine Anfrage an den Auth-Server gestellt. Bei erfolgreichem Response von Seiten des Auth-Servers werden die Benutzer-ID und die Client-ID gespeichert und für die Bearbeitung des Requests verwendet. Sobald eine Response versendet werden muss - sei es durch erfolgreiche Bearbeitung oder durch eine Fehlermeldung - wird die Response-Middleware verwendet.

5.5 File-Persistence

Die File-Persistence ist in MySQL umgesetzt worden und entspricht dem im vorigen Kapitel entwickelten Entwurf der File-Persistence (siehe Punkt 4.6). Listing (5.6) zeigt

die Implementierung von der Tabelle »files« mit ihren Attributen. Als Engine wurde InnoDB verwendet und die Codierung ist utf8. Die übrigen Tabellen sind analog.

Listing 5.6: Implementierung der »files«-Datenbank

```
1  DROP TABLE IF EXISTS `files`;
2  CREATE TABLE IF NOT EXISTS `files` (
3    `id` CHAR(36)  NOT NULL,
4    `name` VARCHAR(50) NOT NULL,
5    `extension` VARCHAR(5) NOT NULL,
6    `version` INT(11) NOT NULL,
7    `date` DATE NOT NULL,
8    `userId` CHAR(36) NOT NULL,
9    PRIMARY KEY (`id`)
10 ) ENGINE=InnoDB DEFAULT CHARSET=utf8;
```

5.6 Docker

Neben dem *Dockerfile* (2.5.2) für jede Komponente wurde eine Docker-Compose-Datei für den gemeinsamen Start- und Stopp-Vorgang entwickelt. In dieser sind alle verfügbaren Docker-Container deklariert. Ebenso wird ein Speicherplatz (volumes) für die Auth- und File-Persistence reserviert. Listing (5.7) zeigt einen Auszug aus der Docker-Compose-Datei und wie der File-Manager definiert, sowie der zugehörige Speicher für Dateien reserviert wurde [Zeile 6].

Listing 5.7: Auszug aus der Docker-Compose

```
fileserver:
    build: fileServer/docker
    container_name: fileServer
    volumes:
      - ./FileServer/:/usr/src/fileserver
      - dashboard-fileserver-data:/usr/src/fileserver/stored/
          user_Files
    ports:
      - "5001:80"
    links:
      - identityserver:localhost

volumes:
  dashboard-fileserver-data:
    driver: local
```

6 Evaluierung

Zur Evaluierung der Funktionalitäten des Systems, werden verschiedene Testfälle betrachtet und durchgeführt. Dabei werden die Testfälle in unterschiedliche Testverfahren unterteilt. Für den Komponententest werden einfache Requests an den File-Manager gesendet und ausgewertet. Beim Integrationstest wird das Zusammenspiel der einzelnen Komponenten des Systems getestet. Zum Schluss wird ein Systemtest durchgeführt. Es wird auch diskutiert, inwieweit die Anforderungen erfüllt wurden. Aufgrund des Umfangs der gesamten Testfälle konnten nicht alle durchgeführten Tests in dieser Arbeit berücksichtigt werden.

Für bestimmte Testfälle wurde ein Test-Client entwickelt, der einen anderen Webservice repräsentieren und Aktionen im System durchführen soll. Dieser kommt sowohl bei den Komponententests und den Integrationstests zur Anwendung.

6.1 Komponententest

Beim Komponententest werden verschiedene Funktionalitäten des File-Managers getestet. Die restlichen Komponenten des Systems, werden unter anderem beim Integrationstest geprüft.

Um die Funktionalitäten des File-Managers zu prüfen, wurden folgende Test durchgeführt.

URI-Management Es wird geprüft, ob nur die definierten URIs als Endpunkte dienen. Hierfür wird ein Request an eine nicht definierte URI gesendet.

Autorisation Es wird geprüft, ob die Bearbeitung des Requests unterbunden wird, wenn kein Referenz-Token angegeben wurde.

Datei hochladen Der Client soll in der Lage sein eine Datei auf den File-Manager hochzuladen.

Datei herunterladen Der Client soll in der Lage sein eine Datei vom File-Manager herunterzuladen.

Datein löschen Der Client soll in der Lage sein eine Datei zu löschen.

Dabei wird die Überprüfung der Funktionalitäten für die Dateien über den entwickelten Test-Client vollzogen.

URI-Management

Beim Testfall des URI-Managements, wird ein Request mit einem gültigem Referenz-Token an den File-Manager gesendet. Dabei wird der Request an eine nicht definierte URI gesendet. Dieser Request wurde durch das Programm *Postman* gesendet. In der Abbildung (6.1) ist zu sehen, dass als Response eine Nachricht mit dem *Status-Code 404 Not Found* im *Header* zurückgegeben wird. Im Body der Response-Nachricht wird ein Hinweis gegeben ,warum der Request ungültig war. Somit wurde dieser Test erfolgreich bestanden.

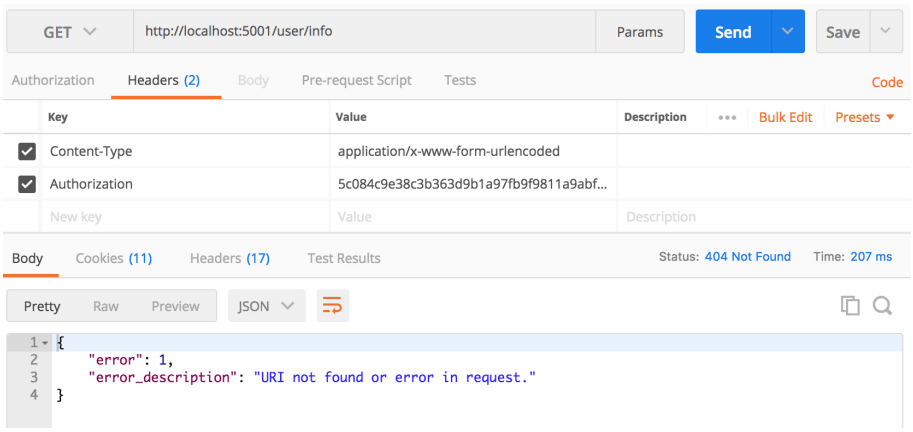

Abbildung 6.1: Response-Nachricht auf eine nicht definierte URI

Autorisation

Bei diesem Test wird wie beim URI-Management ein Request per *Postman* an den File-Manager gesendet. Der Unterschied hier ist, dass das verwendet Referenz-Token ungültig

ist. Dafür ist URI korrekt. Abbildung (6.2) zeigt die Response-Nachricht und den *Status-Code 401 Unauthorized*. Ebenfalls wird hier eine Nachricht im Response-Body gesendet. Durch diesen Test wurde gezeigt, dass ein unbefugter Zugriff auf den File-Manager unterbunden wird.

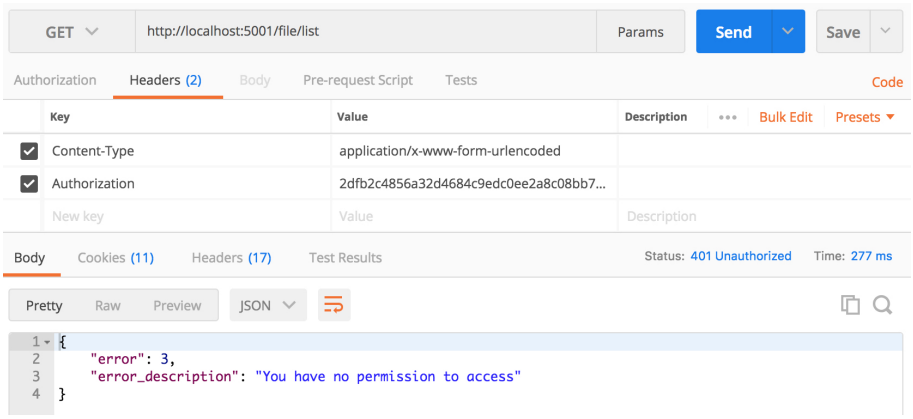

Abbildung 6.2: Response-Nachricht auf einen unautorisierten Request

Datei hochladen

Um die Funktionalität des hochladen einer Datei auf den File-Manager zu prüfen, kann der implementierte Test-Client verwendet werden.Die Abbildung (6.3) zeigt die Liste der verfügbaren Dateien, vor der Ausführung der Hochlade-Funktion.

Vorhandene Daten:

No.	Name	Endung	Version	Client	Datum ↓
1	test	txt	1	testClient	2018-09-06

Abbildung 6.3: Liste der Dateien vor der Aktion des Hochladens

Abbildung (6.4) zeigt die neue Liste nach der Ausführung der Hochlade-Funktion. Dadurch ist erkennbar, dass die Funktionalität des Hochladens von Dateien erfolgreich implementiert wurde.

Vorhandene Daten:

No.	Name	Endung	Version	Client	Datum ↓
1	test	txt	1	testClient	2018-09-06
2	test	json	1	testClient	2018-09-18

Abbildung 6.4: Liste der Dateien nach der Aktion des Hochladens

Datei herunterladen

Die Durchführung und Auswertung der Funktionalität des Herunterladens von Dateien, kann ebenfalls über den Test-Client erfolgen. Durch Angabe des *Dateinamen*, der *Version* und des *Erstellungsdatums* der Datei ist es möglich eine Datei zur weiteren Verarbeitung herunterzuladen. Dies wurde erfolgreich durchgeführt und die Dateien konnten durch die Verwendung des Test-Clients heruntergeladen werden.

Datei löschen

Die Löschfunktion wurde abermals durch die Verwendung des Test-Clients getestet. Auch hier wurde eine Datei unter Angabe des *Dateinamen*, der *Version* und des *Erstellungsdatums* der Datei erfolgreich aus dem System entfernt.

Somit wurde der Komponententest für den File-Manager erfolgreich durchgeführt und die Funktionalitäten überprüft.

6.2 Integrationstest

Der Integrationstest versucht alle vorhandenen Komponente des Systems in seinem Testfall mit einzubeziehen. Deswegen wird mit dem entwickelten Test-Client ein Login- und Logout-Vorgang durchgeführt.

Das Login-Verhalten soll dem *Single-Sign-In* entsprechen. Daher ist es das Ziel dieses Tests, festzustellen, ob ein Benutzer, nachdem er sich schon über das Dashboard bei dem Auth-Server authentifiziert hat, auch in einem anderen Webservice authentifiziert

ist. Bei der Authentifizierung beim Auth-Server wird ebenfalls die Funktionalität der Auth-Persistence überprüft.

Nachdem sich der Benutzer über das Dashboard beim Auth-Server authentifiziert hat, wird ihm auf der Web-Oberfläche des Dashboards eine Ansicht der verfügbaren Webservices angezeigt. Unter anderem ist hier der Test-Client zu finden. Durch Ausführen des Test-Client wird der Benutzer auf eine neue Webseite geleitet und bekommt dort die Inhalte des Test-Client angezeigt. Ebenso wird festgestellt, dass der Benutzer gleichfalls beim Test-Client angemeldet ist. Somit ist dieser Testfall erfolgreich vollzogen worden.

Durch den erfolgreichen *Single-Sign-In* wird auf der Web-Oberfläche des Test-Client eine Liste der verfügbaren Dateien angezeigt. Somit wurde auch eine erfolgreiche Kommunikation mit dem File-Manager und der File-Persistence überprüft.

Nachdem diese Tests erfolgreich durchgelaufen sind, kann nun der Logout-Vorgang gestartet werden. Der Logout-Vorgang soll nach Anforderungen und Entwurf ein *Single-Sign-Out* sein. Hierfür wird der Benutzer durch die Web-Oberfläche des Test-Client zur Web-Oberfläche des Auth-Server geleitet. Dort bestätigt der Benutzer seinen Logout-Wunsch. Nun müssen alle anderen Clients wie das Dashboard den Benutzer ebenfalls abmelden. Diese Schritte werden alle erfolgreich vollzogen, da beim nächsten Ansteuern des Dashboards eine erneute Authentifizierung vorgenommen werden muss.

An dieses beiden Testfällen ist zu erkennen, dass alle Komponenten des Systems, erfolgreich ihr Zusammenspiel durchführen.

6.3 Systemtest

Im Kapitel der Anforderungsspezifikation (3) wurden funktionale und nicht-funktionale Anforderungen erfasst. In diesem Kapitel wird untersucht, inwieweit die Anforderung umgesetzt werden konnten.

6.3.1 Funktionale Anforderungen

Benutzer kann sich registrieren

Diese Anforderung wurde erfolgreich mit dem Auth-Server umgesetzt. Der Benutzer muss bei der Registrierung die *Email-Adresse*, das *Passwort*, den *Vor- und Nachname* und die *Adresse* angeben. Einzig ein Zwischenschritt, in dem die Email-Adresse validiert wird, ist nicht umgesetzt worden.

Benutzer kann sich einloggen

Diese Funktionalität ist auf Saiten des Auth-Server umgesetzt worden. Ein Benutzer kann sich im System einloggen und ist bei allen verfügbaren Services eingeloggt.

Benutzer kann sich ausloggen

Die Anforderung des ausloggens ist ebenfalls durch den Auth-Server abgedeckt. Der Benutzer kann sich über die Log-Out-Funktion des Auth-Servers abmelden und allen Services, die den Zustand des Benutzers abfragen, bekommen mitgeteilt, dass der Benutzer nicht mehr aktiv ist.

Ansicht der verfügbaren Services

Diese Anforderung wurde durch den File-Manager umgesetzt. Ein Services kann ein Request an den File-Manager senden und als Response wird eine Liste der verfügbaren Services ausgegeben.

Authentifizierung von Services

Die Authentifizierung wurde im Auth-Server umgesetzt. Services müssen hier eingetragen sein, damit sie Zugriff auf den File-Manager erhalten können.

Services wird über die aktuelle Sitzung informiert

Die Services können den Endpunkt *Check-Session-Frame* des Auth-Servers ansprechen und den Status der aktuellen Sitzung des Benutzers abfragen.

Datenspeicherung

Diese Anforderung wurde durch den File-Manager erfüllt. Er bietet eine Schnittstelle, um Dateien auf den File-Manager hochzuladen und abzuspeichern.

Daten löschen

Diese Anforderung wurde ebenfalls durch den File-Manager erfüllt. Es wird ebenfalls eine Schnittstelle angeboten, um Dateien auf dem File-Server zu löschen.

Daten erhalten

Diese Anforderung wurde durch die File-Persistence erfolgreich umgesetzt. Die File-Persistence hat die Möglichkeit Dateien dauerhaft zu lagern.

Triggern von Microservices

Diese Anforderung wurde aus zeitbedingten Gründen nicht umgesetzt.

Verwaltung für Daten

Die Verwaltung für die Zugriffssteuerung auf die Dateien, konnte wegen des Umfanges der Arbeit nicht berücksichtigt werden.

Monitoring

Wie bei der Verwaltung für Daten, konnte das Monitoring wegen des Umfanges der Arbeit nicht berücksichtigt werden.

6.3.2 Nicht-funktionale Anforderungen

Anhand der erfüllten funktionalen Anforderungen konnte die Umsetzung einer Portalseite und Integrationsumgebung nachgewiesen werden. Im nächsten Schritt, wird eine Evaluierung der nicht-funktionalen Anforderungen hinsichtlich der Qualität durchgeführt

Die Erweiterbarkeit des Systems ist durch die getroffene Designentscheidung gewährleistet. Durch die Separierung der einzelnen Komponente ist es möglich Änderungen am

System einfach umzusetzen. So können die bisher nicht umgesetzten funktionalen Anforderungen durch einen eigenen Service und Docker-Container in das System eingehängt werden. Hierfür müssen keine großen Änderungen am bestehenden System vorgenommen werden. Ebenso können Änderungen und Aktualisierungen an den einzelnen Komponenten des Systems vorgenommen werden, ohne dass das ganze System »angefasst« werden muss.

Durch die Evaluierung der Testfälle konnte die Zuverlässigkeit des Systems nachgewiesen werden. Die Testfälle haben gezeigt, dass ein korrektes Verhalten des System zu jedem Zeitpunkt ermöglicht wird. Sollte durch einen falschen Request ein Fehler auftreten, wird dieser an der entsprechenden Komponente abgewiesen und der Request-Steller wird durch eine Response-Nachricht darauf hingewiesen.

Eine hohe Verfügbarkeit kann durch die Separierung der einzelnen Komponenten ebenfalls erlangt werden. Durch die Separierung ist es möglich einzelne Komponente bei Fehlverhalten oder Defekten schnell auszutauschen.

Hinsichtlich der Sicherheit wurde darauf geachtet, dass es keine Möglichkeit gibt unerlaubt auf den File-Manager oder das Dashboard zuzugreifen. Dazu wurde der Auth-Server umgesetzt. Dieser schützt den Zugriff auf die restlichen Komponenten des Systems. Da das OAuth2- und OpenId Connect-Protokoll als Sicherheitsstandard verwendet wurden, ist zu beachten, dass das System nur soweit geschützt ist, wie es OAuth2 und OpenId Connect zulassen.

Mit Blick auf die Benutzbarkeit und der Usability ist die Portalseite und Integrationsumgebung den Anforderungen entsprechend umgesetzt worden. Der Benutzer bemerkt die im Hintergrund ablaufenden Prozesse nicht. Dieses Verhalten fördert die Bequemlichkeit, die Geschwindigkeit und die Fehlervermeidung seitens des Benutzers.

6.4 Veröffentlichung

In dieser Arbeit wurde die Portalseite und Integrationsumgebung im lokalen Netzwerk erforscht und umgesetzt. Für eine Veröffentlichung des Systems kann Kubernetes verwendet werden. Eine Kubernetes-Plattform wird von HAW Hamburg betreiben. Kubernetes ist ein Open-Source-System zur Automatisierung der Bereitstellung , Skalierung und Verwaltung von Container-Anwendungen auf verteilten Hosts. Um die Orchestrierung der

Portalseite und Integrationsumgebung mit Kubernetes durchzuführen, müssten noch ein paar Anpassungen getätigt werden.

Außerdem muss noch ein Augenmerk auf die Sicherheit gelegt werden, wenn das System öffentlich gemacht wird. Hierfür können Hypertext Transfer Protocol Secure (HTTPS) und TLS verwendet werden. HTTPS stellt im World Wide Web eine Transportverschlüsselung dar, womit Daten abhörsicher übertragen werden können. TLS ist eine Technik des HTTPS um Daten verschlüsselt zu übertragen. Dabei finden die Verbindungen zwischen Client und Server über TLS-Serverzertifikate statt. Diese Zertifikate verwenden asymmetrische Verschlüsselungstechniken bei denen der Schlüssel zum Verschlüsseln ein anderer ist, als der zum Entschlüsseln.

6.5 Erkenntnisse der Evaluierung

Die Evaluierung des Systems hat gezeigt, dass die Umsetzung einer Portalseite und Integrationsumgebung durchführbar ist. Dies wird durch die Implementierung der funktionalen und nicht-funktionalen Anforderungen deutlich. Ebenso konnte veranschaulicht werden, dass die Architektur, die im Systementwurf beschrieben wurde, umgesetzt werden konnte. Durch die Separierung der einzelnen Komponenten ist es möglich die Portalseite und Integrationsumgebung möglichst einfach zu erweitern. Die Problematik hinsichtlich der Sicherheit und Datenverwaltung des File-Managers konnten zufriedenstellend gelöst werden.

Ebenso hat die Evaluierung gezeigt, dass auf der Grundlage der Portalseite und Integrationsumgebung weitere Entwicklungen von Komponenten stattfinden können. Sei es die nicht umgesetzten Triggering-Funktion, die Verwaltung der Zugriffsrechte auf die Dateien oder ein Webservice für administrative Verwaltung.

Durch die Evaluierung wurde aufgezeigt, dass für eine Veröffentlichung der Portalseite und Integrationsumgebung noch kleinere Änderungen am System durchzuführen sind. Erst durch die Veröffentlichung stellt die Portalseite und Integrationsumgebung einen Mehrwert dar.

7 Schluss

7.1 Zusammenfassung

In dieser Arbeit wurde nachgewiesen, dass eine Umsetzung einer Portalseite und Integrationsumgebung möglich ist. Als Voraussetzung für die Portalseite und Integrationsumgebung, sollte die HAWAI-Landschaft (vgl. Punkt 3.1) dienen. Die Portalseite und Integrationsumgebung sollte eine Web-Oberfläche für einen Benutzer bieten, der sich in der HAWAI-Landschaft zurecht finden möchte. Des Weiteren sollte die Portalseite und Integrationsumgebung eine Kommunikation- und Authentifizierungsschnittstelle für verschiedene Services bieten.

Am Anfang der Arbeit wurden die Anforderungen (siehe Punkt 3.2) an die Portalseite und Integrationsumgebung zusammengetragen. Hierfür wurden mehrere Gespräche mit den Beteiligten des HAWAI-Projekts geführt. Durch diese Gespräche wurden funktionale und nicht-funktionale Anforderungen bestimmt. Die Anforderungen umschlossen ein Gebiet von einfachen bis zu wesentlich komplexeren Funktionalitäten.

Im nächsten Schritt musste auf Grundlage der Anforderungen ein passender Systementwurf entwickelt werden. Diese geschah, indem man das gesamte System analysierte. Daraus ging hervor, dass für verschiedene Aufgaben verschiedene Komponente gebraucht wurden. Deshalb wurde beschlossen, nicht ein großes monolithisches System zu entwerfen sondern die Anforderungen des System in einige kleinere Systeme zu teilen (siehe Punkt 4.1). Nachdem die Aufgliederung vollzogen wurde, wurden die Eigenschaften der jeweiligen Komponente diskutiert. Dabei stellte sich heraus, dass die einzelnen Komponenten mit verschiedenen Programmiersprachen und Programmiertechniken zu bewerkstelligen waren. Des Weiteren wurde sich Gedanken über die Interprozesskommunikation (Punkt 4.7) zwischen den einzelnen Komponenten gemacht. Zum Abschluss des Systementwurfes wurden Gründe aufgezeigt, warum die separierten Komponenten in Docker-Container implementiert wurden.

Auf Basis des Systementwurfes, wurde die Portalseite und Integrationsumgebung umgesetzt. Hierfür wurden die vorab definierten Programmiersprachen und Programmiertechniken verwendet. Im Kapitel der Implementierung wurde auf die einzelnen Komponenten eingegangen und Ausschnitte der Programmiertechniken gezeigt.

Durch die letztendlich durchgeführte Evaluierung der Portalseite und Integrationsumgebung, ergaben sich Erkenntnisse, die zur Weiterentwicklung des Systems herangezogen werden können. Diese werden ausführlicher im nächsten Kapitel dargestellt.

7.2 Ausblick

Durch diese Arbeit wurde gezeigt, dass eine Portalseite und Integrationsumgebung gut an die Bedürfnisse eines verteilten Systems angepasst und zur Kommunikation von verschiedenen Services genutzt werden kann. Allerdings hat sich auch gezeigt, dass im Rahmen einer Bachelorarbeit nicht alle Funktionalitäten umgesetzt werden konnten.

Um das System im produktiven Maße nutzen zu können sind noch einige Anpassungen zu vollziehen. So könnten die nicht umgesetzten Anforderungen implementiert werden. Beispielhaft wären die Funktionalitäten des Triggern, der Verwaltung von Zugriffsrechten auf Dateien oder des Monitoring zu nennen. Ebenso könnte in einer späteren Arbeit ein Service für administrative Verwaltungsarbeit durchgeführt werden. Die Grundlagen für diese Erweiterungen sind mit dem vorliegenden System gegeben.

Essentiell für eine Nutzung des Systems ist die Sicherheit der Benutzer und der Daten des File-Managers. Um die Sicherheit in der Zukunft und bei einer Veröffentlichung des Systems zu erhöhen, wäre es denkbar den Auth-Server speziell generierte Sicherheitszertifikate ausstellen zu lassen und Transport Layer Security zu nutzen.

Inwieweit die Portalseite und Integrationsumgebung einen Mehrwert für die HAWAI-Landschaft haben wird, wird die Zeit zeigen. Als Integrationspunkt für Services bietet sich das System allerdings an und kann in späteren wissenschaftlichen Arbeiten weiter entwickelt werden.

Literaturverzeichnis

[Angular Architecure 2018] Angular Architecure: *Angular Architektur*. 2018. – URL https://angular.io/guide/architecture. – Zugriffsdatum: 11.09.2018

[Bihis 2015] Bihis, C: *Mastering OAuth 2.0*. Packt Publishing, 2015. – URL https://books.google.de/books?id=HjflCwAAQBAJ. – ISBN 9781784392307

[Binani u. a. 2016] Binani, Sneha ; Gutti, Ajinkya ; Upadhyay, Shivam: SQL vs. NoSQL vs. NewSQL-A Comparative Study. In: *Communications on Applied Electronics* (2016). – ISSN 23944714

[Boettiger 2015] Boettiger, Carl: An introduction to Docker for reproducible research. In: *ACM SIGOPS Operating Systems Review* (2015). – ISBN 0163-5980

[Brock und Baier 2016] Brock, Allen ; Baier, Dominick: *Identity Server 4*. 2016. – URL http://docs.identityserver.io/en/release/. – Zugriffsdatum: 11.09.2018

[Buschmann 1996] Buschmann, F: *Pattern-Oriented Software Architecture, A System of Patterns*. Wiley, 1996 (Wiley Series in Software Design Patterns). – URL https://books.google.de/books?id=gJjgAAAAMAAJ. – ISBN 9780471958697

[Docker 2018] Docker, Inc.: *What is a Container*. 2018. – URL https://www.docker.com/resources/what-container. – Zugriffsdatum: 04.09.2018

[Fielding und Reschke 2014] Fielding, R ; Reschke, J: *HTTP Status Codes*. 2014. – URL https://tools.ietf.org/html/rfc7231. – Zugriffsdatum: 14.09.2018

[Fielding 2000] Fielding, Roy T.: Architectural Styles and the Design of Network-based Software Architectures. In: *Building* (2000). – ISBN 0599871180

[Hawai 2018] Hawai: *Hawai Research Group*. 2018. – URL https://hawai-web.ful.informatik.haw-hamburg.de/. – Zugriffsdatum: 06.09.2018

[Höller 2017] Höller, Christoph: *Angular - Das umfassende Handbuch*. Rheinwerk Verlag GmbH, 2017. – 783 S. – ISBN 978-3-8362-3914-1

[Johannsen 2018] Johannsen, Jonas: *Umsetzung komplexer Geschäftsprozesse in Verteilten Systemen mit Docker*, HAW Hamburg, Dissertation, 2018. – URL http://edoc.sub.uni-hamburg.de/haw/volltexte/2018/4242/

[Krypczyk und Bochkor 2018] Krypczyk, V ; Bochkor, O: *Handbuch für Softwareentwickler: Das Standardwerk zu professionellem Software Engineering*. Rheinwerk Verlag GmbH, 2018 (Rheinwerk Computing). – URL https://books.google.de/books?id=u-vCtAEACAAJ. – ISBN 9783836244763

[Meier und Kaufmann 2016] Meier, Andreas ; Kaufmann, Michael: *SQL- und NoSQL-Datenbanken*. 8., überarbeitete und erweiterte Auflage. Springer Vieweg, 2016. – ISBN 978-3-662-47663-5 and 3-662-47663-0

[Mouat 2016] Mouat, A: *Docker: Software entwickeln und deployen mit Containern*. Dpunkt.Verlag GmbH, 2016. – URL https://books.google.de/books?id=2AC-jwEACAAJ. – ISBN 9783864903847

[OIDCS 2017] OIDCS: *OpenID Connect Server for ASP.NET Core*. 2017. – URL https://github.com/openiddict/openiddict-core. – Zugriffsdatum: 11.09.2018

[Pautasso u. a. 2008] Pautasso, Cesare ; Zimmermann, Olaf ; Leymann, Frank: RESTful Web Services vs . " Big " Web Services : Making the Right Architectural Decision Categories and Subject Descriptors. In: *Technology* (2008). – ISBN 9781605580852

[Phalcon 2017] Phalcon: *Phalcon - a full-stack PHP Framework*. 2017. – URL https://phalconphp.com/de/. – Zugriffsdatum: 13.09.2018

[Richer und Sanso 2017] Richer, J ; Sanso, A: *OAuth 2 in Action*. Manning Publications, 2017. – URL https://books.google.de/books?id=QNLEjwEACAAJ. – ISBN 9781617293276

[Richter 2015] Richter, J: *OAuth 2.0 Token Introspection*. 2015. – URL https://tools.ietf.org/html/rfc7662. – Zugriffsdatum: 12.09.2018

[Sakimura u. a. 2014] Sakimura, Natsuhiko ; Bradley, J ; Jones, M ; Medeiros, B de ; Mortimore, C: Openid connect core 1.0. In: *The OpenID Foundation* (2014).

– URL https://openid.net/specs/openid-connect-core-1_0.html. – ISSN 1537-6613

[SensioLabs 2017] SensioLabs: *Symfony a PHP framework*. 2017. – URL https://symfony.com/. – Zugriffsdatum: 13.09.2018

[Steyer u. a. 2014] Steyer, M ; Schwichtenberg, H ; Dr, H S.: *Moderne Web-Anwendungen mit ASP.NET MVC und JavaScript*. O'Reilly Verlag, 2014. – URL https://books.google.de/books?id=Nc-rAwAAQBAJ. – ISBN 9783955617417

[Taibi u. a. 2017] Taibi, D ; Lenarduzzi, V ; Pahl, C ; Janes, A: Microservices in agile software development: a workshop-based study into issues, advantages, and disadvantages. In: *Proceedings of the XP2017 Scientific Workshops* (2017). ISBN 9781450352642

[Tanenbaum und van Steen 2002] Tanenbaum, A S. ; Steen, M van: *Distributed Systems: Principles and Paradigms*. Prentice Hall, 2002. – URL https://books.google.de/books?id=LYQpAQAAMAAJ. – ISBN 9780130888938

[Tilkov 2015] Tilkov, S: *REST und HTTP: Entwicklung und Integration nach dem Architekturstil des Web*. dpunkt, 2015. – URL https://books.google.de/books?id=pJF-ngEACAAJ. – ISBN 9783864901201

[Toth 2015] Toth, S: *Vorgehensmuster für Softwarearchitektur: Kombinierbare Praktiken in Zeiten von Agile und Lean*. Hanser, Carl, 2015. – URL https://books.google.de/books?id=pON6rgEACAAJ. – ISBN 9783446443952

[Wolff 2016] Wolff, E: *Microservices : Grundlagen flexibler Softwarearchitekturen*. 1., korrigierter Nachdruck. Heidelberg : dpunkt.verlag, 2016. – URL http://d-nb.info/1074154290/04

Erklärung zur selbstständigen Bearbeitung einer Abschlussarbeit

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

Ort	Datum	Unterschrift im Original