Konzeption und Entwicklung einer Web-API am Beispiel des Geotagging-Services Grafflr

Levin Mauritz
2107232

BACHELORARBEIT

eingereicht am
Fachhochschul-Bachelorstudiengang

Medientechnik

in Hamburg

im Juni 2016

Diese Arbeit entstand an der
Hochschule für Angewandte Wissenschaften Hamburg

Fakultät Design, Medien und Information
Department Medientechnik

im

Sommersemester 2016

Betreuer:

Erstprüfer: Prof. Dr. Nils Martini
Zweitprüfer: Prof. Dr. Andreas Plaß

Erklärung

Ich versichere, die vorliegende Arbeit selbstständig ohne fremde Hilfe verfasst und keine anderen Quellen und Hilfsmittel als die angegebenen benutzt zu haben. Die aus anderen Werken wörtlich entnommenen Stellen oder dem Sinn nach entlehnten Passagen sind durch Quellenangaben eindeutig kenntlich gemacht.

Hamburg, am 27. Juni 2016

Levin Mauritz

Inhaltsverzeichnis

Kurzfassung

Diese Arbeit behandelt die Entwicklung und Umsetzung einer Web-API für einen Geotagging-Service. Das Ziel besteht darin, einen einfachen und soliden Weg zu finden, Daten über das Web mit Client-Applikationen auszutauschen. Die Arbeit beschäftigt sich mit einem Dienst, der für die reale Anwendung geplant ist. Anhand der Anwendungsszenarien des Dienstes werden die nötige Anforderungen an die API formuliert und aus bestehenden technischen Möglichkeiten ein Entwurf des Systems entwickelt. Der erarbeiteten Entwurf wird in Form eines Prototypen umgesetzt und anhand der gestellten Anforderungen evaluiert.

Abstract

This paper is about the development and implementation of a geotagging based Web-API. The intention is to design a simple and robust system to exchange data with client applications over the web. The paper describes the backend design of a service intended to be used in real-world operations. The use cases of the service provide the requirements of the API, on which the considered technology and design patterns are based on. A draft of the API is developed, implemented by a prototype and evaluated by the posed requirements of the application.

Kapitel 1

Einleitung

1.1 Motivation

Ende 2015 hatten zwei Komillitonen und ich die Idee, einen Dienst zu entwickeln mit dem sich Streetart kartographieren lässt. Uns fiel auf, dass überall in Hamburg versteckte Kunstwerke zu finden sind. An Hausfasaden hoch oben unter den Dächern oder hinter Ecken, Stromkästen oder Eingängen haben sich Künstler auf beeindruckende Weise verewigt. Die meiste Zeit übersieht man solche Kunstwerke und erst wenn man mit der Stadt und dem Gebiet bekannt ist, wird bewusst woran man täglich vorübergeht ohne es zu bemerken. Unsere Idee war vorerst einfach konzipiert. Wir wollten einen Dienst entwickeln der es möglich macht mit mobilen Geräten Streetart zu fotografieren, mit einem Geotag zu versehen und zu veröffentlichen. Auf diese Weise soll es möglicht werden seine täglichen Funde mit anderen zu teilen und selbst neue Kunst zu entdecken. Vor allem in fremden Städten wird es so möglich einen kleinen Einblick in die teils überbordernde Straßenkultur zu erhalten und Neues zu entdecken. În Anlehnung an die Graffiti-Malerei wählten wir den Namen Grafflr für den Dienst. Wir entschieden uns für einen Web-Dienst der auf möglichst vielen Endgeräten verwendet werden kann und plattformunabhängig ist. Es soll möglich sein, native Applikationen für mobilen Geräte zu entwickeln, gleichzeitig soll der Dienst über den Browser zugänglich sein. Aufgrund dieser Bedingungen entschieden wir uns den Dienst aufzuspalten, in eine serverseitige Scnittstelle in Form einer API, sowie die dazugehörigen Client-Applikationen, die mit der API kommunizieren und Daten und Interfaces auf dem Endgerät rendern. Diese Arbeit beschäftigt sich mit den Entwicklung der geforderten Web-API. Es wird untersucht welche Ansprüche an die API gestellt sind, welche technischen und architektonischen Möglichkeiten die Forderungen bestmöglich erfüllen, sowie eine technische Umsetzung des Entwurfs durch einen Prototyp.

1.2 Zielsetzung

Ziel dieser Bachelorarbeit ist es, eine Web-API zu entwickeln die serverseitige Ressourcen und Methoden clientseitigen Anwendungen zur Verfügung stellt und verarbeitet. Im speziellen soll die API die nötigen Funktionen beinhalten, die von einem Geotagging-Dienst gefordert werden. Diese gilt es zu definieren und aus den resultierenden Anforderungen eine API zu entwerfen, umzusetzten und zu evaluieren. Die Zielsetztung besteht aus folgende Unterpunkten:

1. Es soll geprüft werden, welche Funktionen und Daten für den Betrieb der Anwendung notwendig sind.
2. Es gilt zu prüfen, welche Möglichkeiten der Umsetzung zur Verfügungstehen und welche davon sich am besten für den konkret vorliegenden Fall eignen.
3. Es soll ein Entwurf der Web-API entwickelt werden, der unter Einbezug der untersuchten technischen Möglichkeiten die gestellten Anforderungen erfüllt.
4. Der Entwurf soll anhand eines Prototypen implementiert und die konkrete technische Umsetzung dokumentiert werden
5. Der entwickelte Prototyp soll in Bezug auf die gestellten Anforderungen evaluiert werden.

1.3 Aufbau der Arbeit

1. Vorerst werden in Kapitel 2 die allgemeinen Grundlagen zum Projekt und zum Thema Georeferenzierung vermittelt.
2. In einer Anforderungsanalyse in Kapitel 3 werden verschiedene Szenarien der Anwendung geschildert und eine Liste von grundsätzlich zu erfüllenden Funktionen der API erarbeitet.
3. Kapitel 4 vermittelt die für das Verständnis der Arbeit nötigen technischen Grundlagen und erläutert kurz einige architektonische Prinzipien für die Entwicklung von serverseitigen Schnittstellen
4. Es folgt die theoretische Entwicklung und Konzeption der API in Kapitel 5. In diesem Teil werden die technischen Möglichkeiten der Umsetzung verglichen und eine mögliche Implementierung erarbeitet.
5. Kapitel 6 dokumentiert die konkrete Umsetzung des Enwurfs in Form eines Prototypen. In diesem Abschnitt finden sich Auszüge aus dem Code und Schaubilder zur Umsetzung der Applikationslogik.

6. Die Gesamtarbeit wird in Kapitel 7 abschließend zusammengefasst. Anhand einer Evaluation des Prototypen bezüglich der gestellten Anforderungen wird geprüft, ob das formulierte Ziel der Arbeit erreicht wurde. In einem anschließenden Fazit werden die gewonnenen Erkenntnisse während der Entwicklung reflektiert. Die Arbeit schließt mit einem Ausblick auf mögliche, zukünftige Entwicklungen der Anwendung ab.

Kapitel 2

Grundlagen

2.1 Georeferenzierung / Geotagging

Unter Georeferenzierung versteht man die Zuweisung von Geokoordinaten zu einem Datensatz. Handelt es sich bei den zu markierenden Daten um Bilder oder Grafiken, wird häufig von Geotagging gesprochen. Die gespeicherten Metadaten bestehen, im Fall der dezimalen Notation, aus je einem Wert für Breiten- und Längengrad. Hierbei handelt es sich um geografische Koordinaten, welche die Lage eines Punktes auf der Erde beschreiben. Die Länge des globalen Koordinatensystems ist hierbei in 360 Grade gerastert, von -180° bis +180°, die Aufteilung der Breite in 180 Grade, von -90° bis +90°. Breitengrade verlaufen parallel zum Äquator, welcher auch den Meridian der Breite darstellt, also 0°. Längengrade hingegen verlaufen von Pol zu Pol. Der Nullmeridian ist willkürlich auf die Länge der Londoner Sternwarte Greenwich gelegt und wird daher häufig als Greenwich-Meridian bezeichnet.

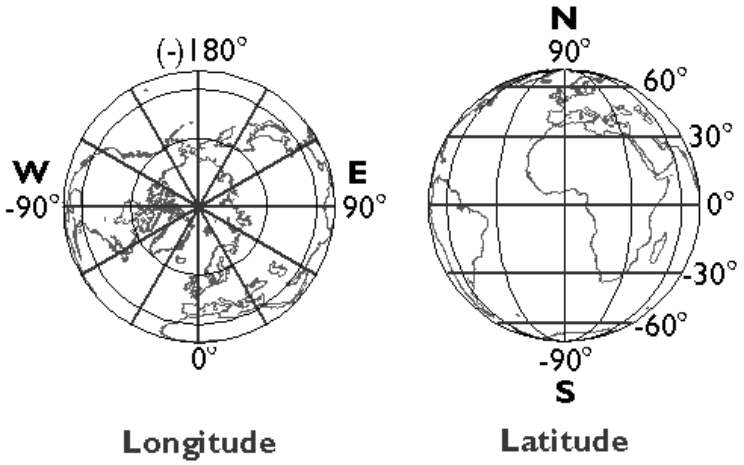

Abbildung 2.1: Aufteilung des Globus nach Längengrad (Longitude) und Breitengrad (Latitude) (https://www.e-education.psu.edu)

Die Erhebung von globalen Positionsdaten erfolgt häufig über satellitengestützte Systeme wie z.B. das Global Positioning System, kurz GPS. In heutigen Fotoapparaten und Mobiltelefonen ist die Lokalisierung durch GPS in vielen Fällen standardmäßig vorhanden und die Speicherung von Geokoordinaten als Metadaten erfolgt mit der entsprechenden Einstellung automatisch. Eine weitere Möglichkeit ist die manuelle Lokalisierung. Hierbei können über eine manuelle Zuweisung auf einer Kartenanwendung wie z.B. Google Maps, die Positionsdaten am Bild gespeichert werden. Neben den genannten Methoden zur Lokalisierung existieren noch viele weitere, auf die jedoch in diesem Kontext nicht weiter eingegangen werden soll. In heutiger Zeit existiert eine Vielzahl von Diensten die mit Georeferenzierung arbeiten.
Google Maps hat hierbei im Bezug auf Nutzerzahlen und Funktionsumfang wohl eine Sonderstellung. Neben der Webanwendung, über welche Orte, Bilder und eine Vielzahl weitere Objekte referenziert werden können, bietet der Dienst eine sehr umfangreiche API zur öffentlichen und automatisierten Nutzung der Karten- und Positionsdaten.

Weitere nennenswerte Dienste im Bezug auf Geotagging sind beispielsweise Flickr[1], ein Webdienst zum Hosten und Veröffentlichen von Bildern und Panoramio[2], ein größerer

[1] https://www.flickr.com/
[2] http://www.panoramio.com/

Geotagging-Dienst einschließlich Community, der inzwischen in Google Maps integriert wurde.

2.2 Der Service Grafflr

Bei dem in dieser Arbeit behandelten Dienst Grafflr handelt es sich ebenfalls um einen Geotagging Dienst. Das Prinzip der Referenzierung von Bildern anhand von geographischen Koordinaten, ist dasselbe wie bei den oben genannten Diensten. Allerdings nicht für jede Art von Bilder, sondern ausschließlich für Streetart Fotografien in Städten. Das Prinzip soll für die erste Version möglichst simpel gehalten werden und beschränkt sich darauf, Bilder über eine Karte abzurufen oder Fotografien mit Geodaten zu versehen und zu veröffentlichen. Die Idee der Anwendung ist grundsätzlich das Auffinden von Streetart, seien es selbst gefundene Malereien oder Funde anderer Nutzer. Zu den potentiellen Nutzern der geplanten Zielgruppe zählen in erster Linie Menschen die Streetart als Hobby ansehen und sich speziell dafür interessieren. Weiter soll sich der Dienst an Reisende wenden und es ermöglichen Straßenmalerei in fremden Städten zu entdecken. In beiden Fällen, die sich natürlich auch überschneiden, macht es Sinn die Anwendung auf mobilen Geräten wie Smartphones bereitzustellen, da die potentiellen User in den meisten Fällen unterwegs sein werden um die Malereien zu kartographieren oder zu suchen. Aus dieser Überlegung heraus wurde die Anwendung in die in dieser Arbeit behandelte API und die dazugehörigen clientseitigen Applikationen aufgeteilt. Derzeit befinden sich zwei Client-Anwendungen in Entwicklung. Ein in Java entwickelter nativer Android-Client, sowie eine auf Javascript basierende Brower-App. Letztere Anwendung wird in der Bachelorarbeit von Jan-Hendrik Kruse [Kru16] genauer beschrieben. Durch die erwähnte Aufteilung in API und Client-Anwendungen ist es in Zukunft problemlos möglich für weiter Endgeräte native Anwendungen zu entwickeln. Aufgrund der großen Verbreitung der Apple iOS Plattform wäre als nächster Schritt eine App hierfür denkbar und sinnvoll.

Kapitel 3

Anforderungsanalyse

Um herauszufinden welche Anforderungen an die API gestellt werden, sollen im folgenden die Use Cases erfasst werden. Unter Use Cases werden in dieser Arbeit die abstrahierten Szenarien verstanden, in denen sich ein User befindet ,um eine Aufgabe mithilfe der Anwendung zu lösen. Aus den festgestellten Use Cases können dann im weiteren die funktionalen und nicht-funktionalen Anforderungen [Som07] an die API spezifiziert werden. Die Überlegungen zum Design und der Umsetzung im weiteren sollen die erarbeiteten Anforderungen erfüllen und an ihnen evaluiert werden.

3.1 Use Cases

Im Folgenden werden die Szenarien dargestellt die sich bei der Verwendung der grundsätzlichen Funktionen eines GeoTagging Dienstes ergeben. Was hierbei als grundsätzlich definiert wird soll nicht als Teil dieser Arbeit verstanden werden, sondern beispielhaft die Entwicklung der Applikation unterstützen.

3.1.1 Bilder finden

Die Hauptfunktion der Applikation ist die Suche nach Bildern auf einem Kartenausschnitt anhand von Geokoordinaten. Ein mögliches Szenario sieht folgendermaßen aus: Ein Nutzer lädt sich die native App auf sein mobiles Gerät oder ruft die Webseite auf. Auf der Startseite des Frontends soll eine Karte eingeblendet sein, welche die Bilder innherhalb eines bestimmten Umkreises des Nutzerstandorts anzeigt. Der Nutzer bekommt eine, bestenfalls paginierte, Auswahl von Bildern in seiner Nähe zurück und kann anhand der empfangenen Daten sich auf die Suche nach der angezeigten Streetart machen. Dieses Szenario setzt sich auf Applikationsseite durch die folgenden Unterpunkte zusammen:

- Ein User möchte eine Übersicht von Bilder in seiner Gegend oder von einem gewählten Kartenausschnitt finden und ansehen.
- Ein User möchte sich ein einzelnes Bild ansehen, mit Beschreibung, Uploader und sonstigen, am Bild gespeicherten Daten.

3.1.2 Bilder hochladen, Daten ändern, Bild löschen

Die Möglichkeit Bilder hochzuladen, zu verändern und zu löschen setzen eine Authentifizierung vorraus. Die Szenarien zur Authentifizierung und Accountverwaltung sind im Punkt 3.1.3 User aufgeführt. Die zweite wichtige Funktion des Dienstes ist die Möglichkeit, gefundene Streetart an den Server zu senden und zu veröffentlichen. Sobald ein Inhalt geteilt wird, stellt sich auch die Frage der Verwaltung des Inhaltes. Es wird notwendig, Daten des Bildes zu verändern, z.B. Position und Beschreibung, sowie die Möglichkeit das Bild zu löschen. Zusammengefasst handelt es sich um Create, Read, Update und Delete (CRUD) Aktionen an der Ressource Image, in Unterpunkten:

- Ein User möchte ein neues Bild erstellen, mit Geodaten markieren und hochladen.
- Ein User möchte Daten am Bild verändern.
- Ein User möchte ein Bild löschen.

3.1.3 User

Die in Punkt 3.1.2 genannten CRUD Operationen erfordern die Möglichkeit einen User zu authentifizieren, um sicherzustellen das nur der Besitzer bzw. Uploader des Bildes berechtigt ist die Ressource zu verändern. Dieser Umstand macht eine grundlegende Userverwaltung notwendig. Die folgenden Punkte führen die Standard-Szenarien des Useraccounts auf:

- Ein User möchte einen neuen Account anlegen.
- Ein User möchte Account-Daten einsehen und ändern.
- Ein User möchte seinen Account löschen.
- Ein User möchte sich an seinem Account an- oder abmelden.

Diese Szenarien liegen jeder herkömmlichen Userverwaltung zugrunde und können als CRUD Operationen an der Ressource User angesehen werden. Eine weiter spezielle Funktion soll gegeben sein, falls der User Bilder favorisieren möchte. Die Operation bietet sich an, da es durch die Applikation möglich sein soll Bilder wiederzufinden. Zusätzlich soll also noch gegeben sein:

- Ein User möchte ein Bild favorisieren.

- Ein User möchte ein Bild entfavorisieren.

Auch hierfür ist die Identifizierung des Nutzers notwendig, um die Markierung des Bildes zuzuordnen und langfristig zu speichern.

3.2 Funktionale Anforderungen

Aus den erarbeiten Use Cases von 3.1 lassen sich nun die funktionalen und nicht-funktionalen Anforderungen an die API ableiten. Aus den User Szenarien bezüglich der Funktionalität ergeben sich folgende Anforderungen, wobei der aufgezählten Reihenfolge keine besodere Bedeutung zukommt:

1. Die API soll es ermöglichen mehrere Bilder anhand von Geo-Koordinaten abzurufen. Hierfür sollen die übergebenen Parameter aus zwei geografischen Koordinaten bestehen, welche die Eckpunkte des rechteckigen Kartenausschnitt auf Clientseite beschreiben.
2. Die API soll einzelne Bilder mit Details zurückgeben können.
3. Die API soll grundlegende Funktionen zur Userverwaltung bereitstellen. Dazu gehören die Möglickeiten einen User Account zu erstellen, Daten daran zu verändern und den Account zu löschen.
4. Die API soll es dem User ermöglichen sich mit seinen Zugangsdaten zu authentifizieren.
5. Die API soll authentifizierten Usern die Möglichkeit bieten neue Bilder hochzuladen, im nachhinein Daten am Bild zu verändern und Bilder zu löschen.
6. Die API soll es authentifizierten Usern ermöglichen Bilder als Favoriten zu markieren und diese Markierung wieder aufzuheben.
7. Die API soll es Usern ermöglichen, Favoriten und Uploads von sich und anderen Usern einzusehen.

3.3 Nicht-funktionale Anforderungen

Um eine zufriedenstellende Nutzung der API sowie deren Funktionen zu gewährleisten, soll die Schnittstelle folgende Merkmale erfüllen:

- **Einfachheit**: Die API soll für Entwickler von Client-Anwendungen möglichst leicht erschließbar sein. Hierzu gehört ein logischer Aufbau sowie der Einsatz von allgemein bekannten und gut dokumentierten technischen Implementierungen, z.B. bei Protokollen und Nachrichtenformaten. Auf diese Weise soll auch auf die Bereitstellung

einer Client-Library verzichtet werden.

- **Stabilität**: Die Endpunkte der API sollen fest definiert sein. Änderungen sollen durch Versionierung umgesetzt werden, um so den Entwicklern die Möglichkeit zu geben den Code zu einem späteren Zeitpunkt anzupassen.
- **Performance**: Die API soll Anfragen ohne nennenswerte Verzögerungen beantworten, um eine flüssige Interaktion zu ermöglichen.
- **Sicherheit**: Die API soll über eine verschlüsselte Verbindung mit dem Client kommunizieren.
- **Dokumentation**: Die API soll möglichst ausführlich durch eine Dokumentation beschrieben werden. Es sollen die Serverendpunkte beschrieben werden, die damit zusammenhängenden Funktionen, die akzepierten Datentypen, sowie die zurückgegebenen Repräsentationen.

Kapitel 4

Technische Grundlagen

Das folgende Kapitel soll dazu dienen, einige in dieser Arbeit behandelten Begriffe und Technologien näher auszuführen und versuchen das notwendige Verständnis für die in den nächsten Kapiteln getroffenen Überlegungen zu vermitteln. Hierbei beschränke ich mich auf die nötigsten Grundlagen, da alles weitere den Rahmen dieser Arbeit übersteigen würde.

4.1 Web-API

Ein Application Programming Interface (kurz API) bezeichnet im allgemeinen eine Schnittstelle zu einem System. Das *Oxford Dictionary of Computer Science* [BN] formuliert eine allgemeine Zusammenfassung folgendermaßen:

> „An Interface that is defined in terms of a set of functions and procedures, and enables a programm to gain access to facilities within an application [...] The use of such facilities enables users to customize the application for their own purpose and to integrate the application into a customized development environment."

APIs finden sich in vielen Bereichen der Softwareentwicklung. Ein Zweck für die Verwendung ist das *Information Hiding* [Par72], ein Muster der Modularisierung, bei welchem sensible Daten nach außen abgeschirmt und nur die für die Interaktion wichtigen Funktionalitäten und Daten veröffentlicht werden. Ein weiterer Zweck ist die Möglichkeit komplexe System zu abstrahieren und über die Schnittstelle vereinfachte und fest definierte Funk-

tionalität bereitszustellen. Die Linux Kernel API[1] für den Zugriff auf System-Ressourcen kann als bekanntes Beispiel gennant werden, ebenso stellt die OpenGL API[2] ein bekanntes Interface zu Hardware und Bibliotheken für 3D Anwendungen dar.
In den letzten Jahren tauchen immer häufiger Web-APIs für den Zugriff auf Ressourcen im World Wide Web auf. In den meisten Fällen handelt es sich dabei um eine Schnittstelle, die über HTTP angesprochen werden kann und Dienste und Ressource eines Services zurückgibt. Es handelt sich dabei um das gleiche System wie bei APIs zu Softwarebibliotheken oder Betriebssystemen. Die Umsetzung unterscheidet sich allerdings stark, da der Zugriff nicht innherhalb des Applikationskontext, sondern über ein Netzwerkprotokoll stattfindet. Ebenso varieren die Implementierungen dieses Zugriffs, welcher über einen Vielzahl von Protokollen oder Architekturen erfolgen kann.
Am Beispiel der *Google Maps Geocoding API*[3] kann ein beispielhafter Zugriff auf eine Web-API demonstriert werden.
Die API bietet die Funktion an, Addressen in Geokoordinaten umzuwandeln. Dies geschieht anhand eines einfachen Aufrufs:

```
GET /maps/api/geocode/json?address=Finkenau+35+Hamburg&key=[...]
http/2.0
HOST:https://maps.googleapis.com
```

Der obenstehende HTTP Request übergibt die als GET Parameter *address* verfasste Adresse *Finkenau+35+Hamburg*, sowie einen Authentifizierungsparameter *key* an die Google Web-API.
Die HTTP Response der API liefert schlicht die geografischen Daten welche mit der übergebenen Adresse assoziert sind, in diesem Fall im JSON Format:

```
HTTP/2.0 200 OK
{
   "results" : [
      {
```

[1] *The Linux Kernel API*. 2016. URL: https://www.kernel.org/doc/htmldocs/kernel-api/ (besucht am 29. 05. 2016).
[2] Khronos Group. *OpenGL API Documentation*. 2016. URL: https://www.opengl.org/documentation/ (besucht am 29. 05. 2016).
[3] *Die Google Maps Geocoding API*. 2016. URL: https://developers.google.com/maps/documentation/geocoding/intro?hl=de (besucht am 29. 05. 2016).

```
            [...]
            "formatted_address" : "Finkenau 35, 22081 Hamburg, Germany",
            "geometry" : {
               [...]
               "location" : {
                  "lat" : 53.5688667,
                  "lng" : 10.0335559
               }
               [...]
      ],
      "status" : "OK"
   }
```

Die erhaltene Antwort kann im weiteren von der Client Applikation verarbeitet werden und z.B. als Datenquelle in ein HTML Template gerendert werden. Die Umsetzung über HTTP dient hierbei nur als Beispiel, prinzipiell kann jedes geeignete Protokoll verwendet werden, allerdings ist das HTTP Protokoll eine weit verbreitete Umsetzung.

4.2 REST

Das Pradigma des *Representational State Transfer* hat die Entwicklung von Web-APIs stark beeinflusst. 2016 sind knapp 80 % der auf programmableweb.com[4] registrierten APIs nach dem REST Design erstellt. Der Architekturstil *REST* wurde 2000 von Roy Thomas Fielding im Zuge seiner Dissertation *Architectural Styles and the Design of Network-based Software Architectures* [Fie00] entwickelt und ist stark and den Prinzipien von HTTP und dem World Wide Web angelehnt.
Es handelt sich hierbei um ein Design zur Kommunikation von Verteilten Systemen. Fielding versucht sich einer Maschine-zu-Maschine Kommunikation auf Basis der Prinzipien des WWW anzunähern, duch die Anwendung von einer Liste von sogenannten *Constraints* (deutsch: Einschränkungen oder Anforderung).
Die Anwendung der von Fielding aufgezeigten Constraints werden teils kontrovers diskutiert. Häufig fällt die Frage wann sich ein System REST-konform nennen darf und wann es dem Prinzip wiederspricht. Da es sich bei REST um ein Paradigma handelt gibt es allerdings keinen Zwang alle Vorgaben einzuhalten und in der reallen Umsetzung hat sich der Begriff *restful* für Applikationen etabliert die einen Großteil der Anforderungen erfüllen.

[4] URL: http://www.programmableweb.com/protocol-api (besucht am 04. 06. 2016).

Im folgenden werden die erwähnten Constraints kurz beschrieben.

Client-Server
Alle Formulierungen und *constraints* beziehen sich auf Applikationen nach dem Client-Server Modell. Dies bringt eine Trennung der Aufgabendomänen von Client- und Serveranwendungen und ermöglicht eine unabhängige Entwicklung beider Teile.

Stateless
Der Punkt der Zustandslosigkeit der Kommunikation stellt einen zentralen Punkt des Prinzips dar. Eine Anfrage an der Server muss ohne einen serverseitig gespeicherten Kontext verarbeitbar sein. Um diese Anforderungen zu erfüllen, muss die Anfrage alle für die Verarbeitung relevanten Informationen mitliefern, was zwar den Nachteil hat, dass die Netzwerklast durch redundante Information steigt, gleichzeitig aber die Komplexität der Verarbeitung bzw. deren Umsetzung stark verringert. Zusätzlich können die serverseitigen Anwendungen sehr einfach horizontal skaliert werden, da kein gemeinsamer Kontext benötigt wird um Anfragen zu bearbeiten. Als Beispiel: Im Restaurant muss der Kellner nicht wissen was unsere bevorzugte Speise ist, wir können unsere komplette Bestellung an jeden verfügbaren Kellner richten und dieser kann sie servieren.

Cache
Das Prinzip von clientseitigem Caching kann die Notwendigkeit einer überflüssigen Verarbeitung eines Requests eliminieren und so die Last der Serveranwendungen reduzieren. *REST* baut in diesem Fall erneut auf das *WWW* und geht von einem Caching auf HTTP Basis aus.

Uniform Interface
REST definiert das Konzept eines einheitlichen Interfaces der Applikation. Es werden einige Elemente definiert, welche es ermöglichen sollen das Interface von der eigentlichen Implementierung der Funktionalität zu trennen. Fielding beschreibt hierbei folgende Elemente:

- Eine *Ressource* dient als Mapping. Gemappt werden kann vielerlei, ein Service, eine Entität oder eine Gruppe von Entitäten, alles was als Ziel einer Hypertext Referenz dienen und benannt werden kann.
- Als *Resource Identifier* wird eine URL verwendet, welche semantisch sinnvoll das Konzept der zugewiesenen Ressource beschreiben soll. Während sich Ressourcen ändern können, sollte sich der korrespondierende Identifier nicht ändern. Als Beispiel:

Die Ressource des Indentifiers *example.com/autor/lieblingsbuch* kann sich über die Zeit ändern, die url zeigt nicht auf einen Datensatz sondern auf ein Konzept.

- Eine *Repräsentation* stellt den derzeitigen Zustand der Ressource in systematischer Form dar und dient zum Transport von Information zwischen Komponenten. Eine Repräsentation der Ressource ̈lieblingsbuch ̈kann z.B. ein HTML oder XML Dokument mitsamt den Metadaten des HTTP Requests sein.

Layered System
Das Gesamtsystem soll aus einem Schichtenmodell bestehen, wobei einzelne Komponenten nur die mit ihnen verknüpften Komponenten kennen sollen. Dies soll dazu dienen die Systemkomplexität zu senken und die Erweiterbarkeit des Systems vereinfachen.

Code-On-Demand
Dieser Constraint wird als optional angegeben. Die Möglichkeit ausführbaren Code an den Client zu senden soll gegeben sein.

Wie oben erwähnt gibt es keinen Zwang alle Forderungen von REST zu erfüllen. Beispielsweise wird beim REST Paradigma von der Verwendung von Cookies abgeraten, allerdings bezieht sich diese Aussage auf die Verwendung von Session Cookies welche den Zustand einer Applikation beschreiben. In der Realität werden Cookies auch in auf REST basierenden Systemen für die Authentifizierung eingesetzt.
Es wird kein wirklicher Bezug auf Authentifizierung genommen und schlicht geraten, die Architektur solle nicht für sicherheitsrelevante Daten verwendet werden. Authentifizierungsmechanismen können bzw. müssen also selbst definiert werden.

4.3 SOAP

SOAP (kurz für *Simple Object Access Protokoll*) ist ein Protokoll zum Austausch von Nachrichten zwischen verteilten Systemen. Es entstand 1999 aus dem von Microsoft mitentwickelten *XML-RPC* und wird heute in Version 1.2 [16a] vom W3C als Industriestandard empfohlen. Die Entwicklung von SOAP wurde durch den Wunsch der Industrie angetrieben, ein einheitliches Nachrichtenformat für den Austausch von Informationen zwischen unbekannten Systemen zu erreichen.
Grundsätzlich definiert SOAP ein Format für Nachrichten und ist unabhängig vom Transportprotokoll, allerdings ist die Verwendung von HTTP für den Transport durch die bereits bestehende Infrastruktur des WWW am weitesten verbreitet. SOAP Nachrichten

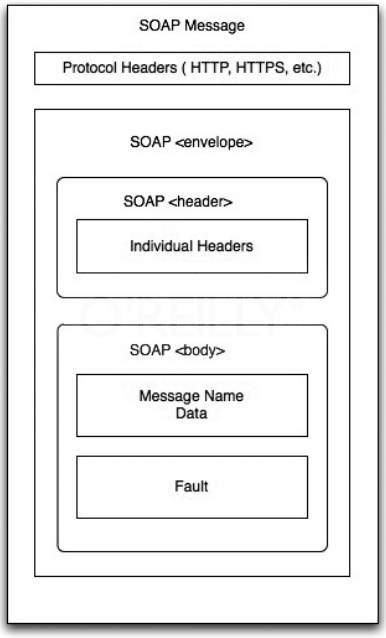

Abbildung 4.1: Schematisches Beispiel einer SOAP Nachricht [16b].

basieren auf XML und weisen ein festes, versionsabhängiges Schema auf. Hierbei wird ein Nachrichten-Header und -Body von einem SOAP-Envelope eingeschlossen, welcher durch ein offizielles XML-Schema des W3Cs validiert wird. Der Envelope stellt das Root-Element des XML-Dokuments dar und definiert den Namespace der Nachricht.

Es ist möglich anhand von SOAP direkten, wenn auch eingeschränkten Zugriff auf Objekte und Datenbanken in fremden Systemen zu bekommen und mit diesen zu interagieren. Die Endpunkte und Methoden sind nicht per Protokoll festgelegt und können anhand der *Web Service Description Language* vollständig beschrieben werden. WSDL ermöglicht dadurch eine automatisierte Code Erzeugung für Client Anwendungen und Tests.
Im Gegensatz zu REST definiert SOAP Authentifizierungsmechanismen. Mithilfe des *WS-Security* Standards ist es möglich einen Security Token in eine SOAP Nachricht einzubinden, sowie das verschlüsseln uns signieren von Nachrichten.
Das SOAP Protokoll für Webservices hat sich vor allem in der internen Kommunikation von Firmen etabliert, sowie in Bereichen die hohe Anforderungen an die Steuerung der Kommunkation und die Flexibilität der angebotenen Methoden stellen. Unter Entwicklern gilt das SOAP Protokoll als eher unbeliebt, da selbst für einfache Anwendungen ein relativ hohes Maß an Entwicklungszeit notwendig ist.

4.4 Symfony Framework

Um den Aufbau der Anwendung verstehen zu können, sollte der Ablauf und die Komponenten des Symfony Frameworks kurz angeschnitten werden.
Das Herz des Frameworks ist die AppKernel Klasse, sie lädt registrierte Erweiterungen, sog. Bundles und Konfigurationen und verarbeitet den Request. Im Einstiegsskript web/app.php wird ein Request Objekt aus den globalen PHP Parametern des eingehenden HTTP Requests erstellt. Eine Kernel Instanz erzeugt ein Response Objekt, indem die Funktion *handleRequest()* mit dem Request als Parameter aufgerufen wird. Die Funktion dispatcht einen Request-Event, auf den die einzelnen Komponenten der Anwendung als Listener subscriben. Eine schematisch Sequenz eines Aufrufs findet sich in Abb. 4.2, allerdings sind hier nur drei Listener als Beispiel angegeben.

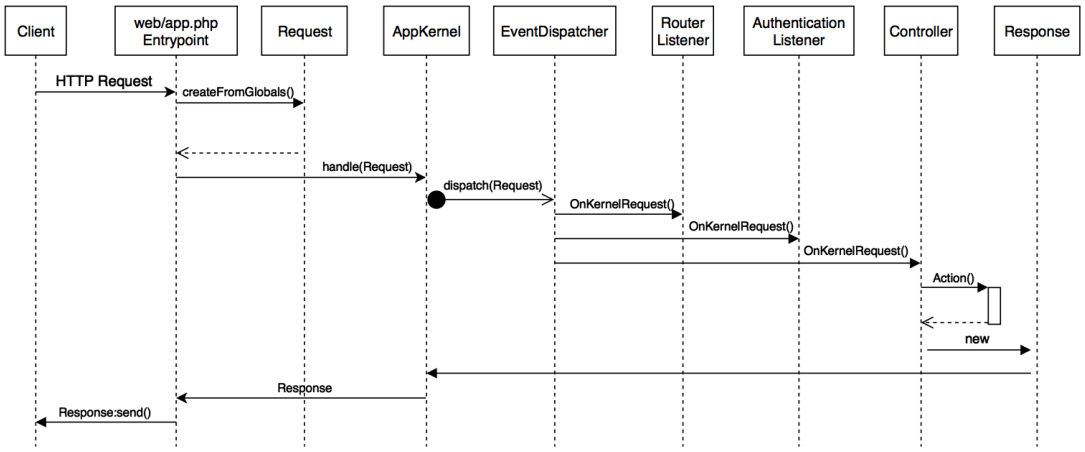

Abbildung 4.2: Sequenzdiagramm der Verarbeitung eines Requests in Symfony

Symfony baut stark auf Events und Dependency Injection. Durch ein ListenerInderface können eigene Listener für verschiedene Events eingehakt werden, für die Grafflr API wurden zwei eigene Listener hinzugefügt (s. 6.4 und 6.6). Durch Dependency Injection können eigene Services einem applikationsweitem Service Container hinzugefügt werden. Wenn kein Bedarf einer speziellen Anpassung besteht können die Komponenten über Yaml Dateien konfiguriert werden.

Kapitel 5

Entwurf und Architektur

Nachdem die Anforderungen nun definiert sind, soll im folgenden der Aufbau der Anwendung entwickelt und die globale Systemarchitektur festgelegt werden. Insgesamt betrachtet handelt es sich bei dem Entwurf um ein Client-Server Modell, ein Prinzip das standardmäßig bei Netzwerkanwendungen Verwendung findet. Die Web-API stellt hierbei die Server-Komponente dar, die Clientanwendungen werden nicht behandelt.

5.1 API Design

Aus den gestellten Anforderungen soll in diesem Punkt das Design der API formuliert werden. Es gilt aus verschiedenen bestehenden Entwurfsmuster, Formaten und Umsetzungen die für den bestehenden Fall am besten geeigneten Lösungen herauszuarbeiten.

5.1.1 REST

Die API soll nach dem in Punkt 4.2 aufgeführten REST Paradigma gestaltet werden. Die Gründe dafür sind hauptsächlich die einfache Umsetzung auf Basis von HTTP. SOAP aus Punkt 4.3 als zusätzliches Protokoll auf HTTP bringt zusätzliche Komplexität in den Entwurf, die Vorteile hingegen finden keine Verwendung. Da die API nur für den internen Gebrauch vorgesehen ist, gibt es keinen Grund ein allgemein definiertes Schema zu verwenden und auf den zusätzliche Overhead in der Verarbeitung, wecher bei der Validierung der XML Dokumente anfällt kann aus diesem Grund ebenso verzichtet werden. Das wichtigste Argument gegen SOAP ist allerdings die schwindende Verbreitung, was zur Folge hat das wenige Entwickler heutzutage noch SOAP für öffentliche Web Dienste verwenden. Die geigneten Programmierer zu finden wird dadurch unnötig erschwert. REST hingegen ist derzeit weit verbreitet und findet auch bei führenden Anbietern im API Bereich Ver-

wendung.

REST als Paradigma schreibt kein Protokoll vor. Die Grafflr API wird mit HTTP umgesetzt, da das Protokoll bei Entwicklern weit verbreitet ist und die meisten Infrastrukturkomponenten wie Router und Firewall eine einfache Verarbeitung ermöglichen bzw. grundsätzlich dafür ausgelegt sind. Die Umsetzung muss allerdings, um nicht chaotisch und unübersichtlich zu werden, an den Vorgaben von REST orientiert sein. Eine einheitliche und konsistente Planung und Umsetzung der Schnittstelle ermöglichen ein vorhersehbares Verhalten, welches die Arbeitszeit in der Entwicklung deutlich reduzieren kann. Im folgenden sollen die von REST vorgegenen Constraints in die Architektur der API einfließen oder auch, wenn keine Notwendigkeit dafür besteht, bewusst ignoriert werden. Wie zu Anfang des Kapitels erwähnt, handelt es sich bei der gesamten Anwendung um ein Client-Server Schema, womit bereits der erste Constraint von REST erfüllt ist. Die Punkte des *Layered System* und *Caching* werden bei der Entwurf des Prototyps vorerst ignoriert, da das Gesamtsystem keine hohe Komplexität erreicht und ein Cache-System nachträglich implementiert werden kann. Der *Code-On-Demand* Constraint wird komplett ignoriert, da für die Funktion keine Notwendigkeit besteht.

5.1.2 URI Konventionen

Die einzelnen Ressourcen der API sollen über einen *Unique Resource Identifier (URI)* angesprochen werden, was der Konvention des *Unified Interface* Constraint von REST entspricht. Was genau sich hinter den Namen der URIs verbirgt ist hier vorest nicht relevant, es gilt eine allgemeine und reproduzierbare Konvention zu finden und zu dokumentieren. Ob die Endpunkte semantisch Sinn ergeben, entscheidet am Ende darüber wie intuitiv ein Entwickler sich die Ressource der API herleiten und umsetzen kann.
Um mit einer Ressource zu interagieren muss sie zuerst am Typ identifiziert werden. Dies entspricht schlicht dem Namen der Ressource, also einem Nomen. Da es mehr als eine Ressource von einem Typ geben kann sollte es weiter möglich sein sie über einen eigenen *Unique Identifier (UID)* anzusprechen. Es kann zudem Subressourcen oder Eigenschaften einer Ressource geben welche direkt adressiert werden müssen. Dies kann nach dem gleichen Schema erfolgen und sich theoretisch rekursiv fortsetzen, wobei ab einer Tiefe von >2 die Länge und Komplexität der URI nicht mehr sinnvoll erscheint und vermieden werden sollte. Das Schema kann also wie folgt festgehalten werden:

```
host/resourceName/{id}/subResource/{subresourceId}[...]
```

Da es auch möglich sein soll Listen von Bildern zu erhalten, gilt der Aufruf der Ressource ohne einen UID immer als Aufruf einer Liste von Ressourcen.

In Abschnitt 3.3 wird auf die Notwendigkeit von Versionierung hingewiesen. Die Version der API wird also ebenfalls relevant für die Enpunkte und sollte zur URI zugefügt werden. Es entwickelt sich eine Baumstruktur zur Benennung der Endpunkte,dargestellt in 5.1.

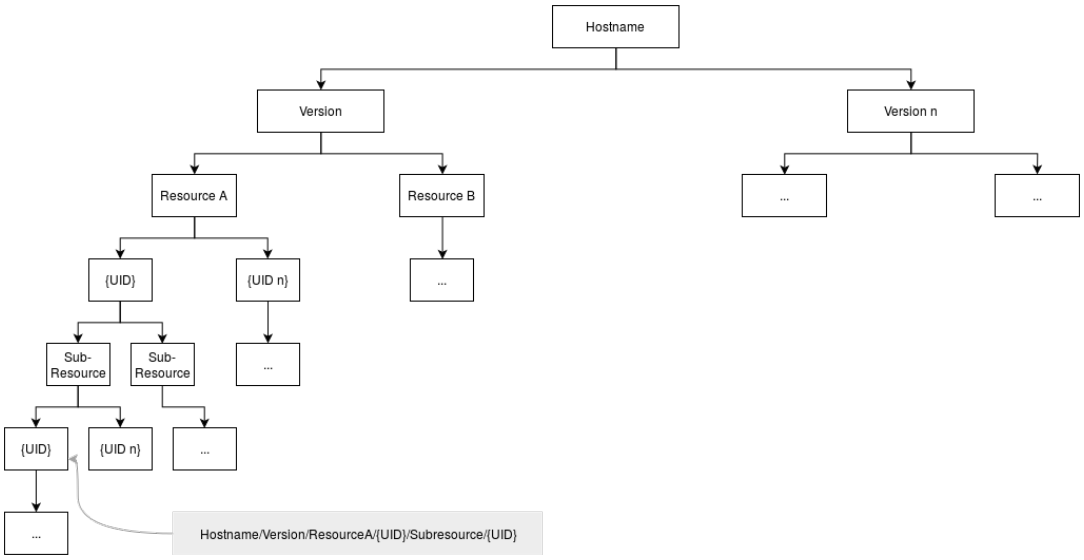

Abbildung 5.1: Baumstruktur zur Idenfikation einer Ressource bis Tiefe = 2

5.1.3 Methoden

Die Ressourcen können nun adressiert werden, es bleibt noch zu klären auf welche Weise mit ihnen interagiert werden kann. Es wäre natürlich möglich die Methode mit in die URI aufzunehmen. Allerdings wiederspricht dies dem Prinzip der schon im Namen *Unique Resource identifier* steckt, nämlich das eine URI eine Ressource nur identifizieren soll. Zudem sollen die URIs nicht unnötig lang werden.
Eine eleganteren und weitverbreitete Lösung ist, auf die bereits vorhandenen HTTP Methoden zurückzugreifen. Das Protokoll stellt in Version 1.1[1] insgesamt neun Methoden zur

[1] URL: https://tools.ietf.org/html/rfc7231 (besucht am 04. 06. 2016).

Verfügung. In diesem Fall sollen nur vier davon erwähnt werden:

- **GET** kommt einer *READ* Operation gleich, eine Repräsentation einer Ressource wird zurückgegeben.
- **POST** erstellt eine Ressource und besitzt einen Body zur Übertragung von Daten. Einzuordnen als *CREATE*
- **PUT** verändert eine Ressource, hat einen Body und ist gleichzusetzen mit *UPDATE*
- **DELETE** löscht eine Ressource.

Somit stehen über das HTTP Protokoll *CRUD (Create, Read, Update, Delete)* Operationen zur Verfügung. Die Methoden *OPTIONS, HEAD, TRACE, CONNECT, PATCH* sind für die API vorerst nicht relevant.
Da es in der Realität mehr Operationen an einem Objekt gibt als CRUD, können optional Methoden am Ende der URI stehen. Die endgültige Form der definierten URIs entspricht dem Schema:

```
[HTTP Method] host/resourceName/{id}/subResource/{subresourceId}/[...]/[customMethod]
```

5.1.4 Authentifizierung

In den Anforderungen in Punkt 3.2 wird die Möglichkeit einer User Authentifizierung formuliert. Da REST keine Empfehlung bezüglich der Client Authentifizierung definiert, muss eine eigene Methode festgelegt werden. Der Constraint der Zustandslosigkeit wird damit teilweise nicht erfüllt, da die Clientanwendung auf Serverseite nun die zwei Zustände *authentifiziert* und *nicht-authentifiziert* haben kann. Diese Verletzung des Paradigmas ist allerdings unumgänglich, da sonst keine Möglichkeit besteht Usern Ressourcen zuzuordnen und umgekehrt.
Für den Prototyp soll eine einfache Authentifizierung durch das senden eines *Access Tokens* umgesetzt werden. Der User authentifiziert sich gegen die API mit seinem Passwort und Usernamen, sind diese valide wird ein Token generiert welcher auf serverseite gesperichert und als Cookie an den Client gesendet wird. Der Token wird nun bei jedem Request mitgesendet und authentifiziert den User. Somit wird die Zustandslosigkeit teilweise erhalten, da jeder Request alle benötigten Daten mitbringt. Als Speicherort für den Token eignet sich eine Datenbank als Token Storage. Diese kann auch von mehreren Instanzen der Applikation angefragt werden und stellt somit kein Problem für die horizontale Skalierbarkeit des Systems dar. Auf die konkrete Umsetzung der Authentifizierung wird in Punkt 6.4 eingegangen.

5.2 Server Architektur

Um Anfragen über HTTP aus dem World Wide Web entgegennehmen zu können wird ein Webserver benötigt, welcher Client-Anfragen entgegennimmt und an die Applikation weiterreicht. Die Applikation, als eigentliche Web-API, kümmert sich um die Verarbeitung der Anfragen, regelt die Authentifizierung und übernimmt das interne Routing. Die Anwendung benötigt Zugriff zu einer Datenbank in der Daten wie Bilder und User persistent gespeichert werden können. Es wird im weiteren ein Storage für die temporären Authentifizierungsdaten der User benötigt, hierfür kann die Datenbank oder das Dateisystem verwendet werden. Eine bessere Lösung ist ein Storage in Form einer In-Memory-Datenbank, da diese über eine einzelne Instanz hinaus erreichbar ist und geringe Zugriffszeiten aufweist. Die globale Server-Architektur wird in 5.2 schematisch dargestellt.

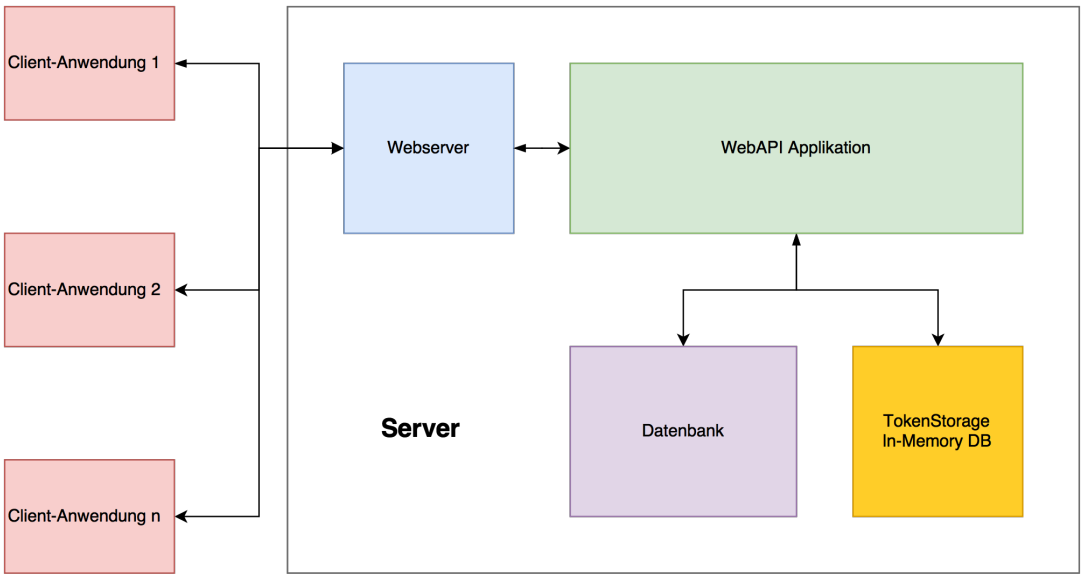

Abbildung 5.2: Schematischer Aufbau des Servers.

5.3 Ressourcen

Nachdem das generelle API-Design und die benötigten Komponenten des Servers geklärt sind, soll nun ein Enwurf für die benötigten Ressourcen erarbeitet werden. Ausgehend von den funktionalen Anforderungen in Punkt 3.2 sollen die geforderderten Ressourcen auf benötigte Entitäten gemappt werden und das URI Schema aus 5.1.2 angewandt werden.

5.3.1 Entitäten

Um die geforderte Funktionalität abzubilden sind für den Prototyp vorerst nur zwei Entitäten von Bedeutung:

- **Image** enhält alle Daten die mit einem Bild assoziert sind.
- **User** stellt den Nutzer dar. Sie umfasst Daten für die Authentifizierung und Assoziationen zu Bildern, z.B. Favoriten und Uploads.

Ein User kann viele Bilder hochladen und er kann viele Bilder favorisieren, es besteht eine doppelte One-to-Many Beziehung zwischen User und Image, dargestellt in Fig X.

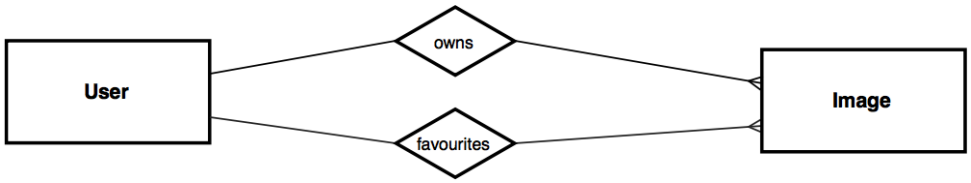

Abbildung 5.3: ER-Diagramm der benötigten Entitäten.

5.3.2 Mapping

Die erwähnten Entitäten können direkt als Ressource gemappt werden und zusätzliche Repräsentationen bestimmt werden.

5.3.2.1 Image

Für Endpunkte der Image Ressource sind einfache CRUD Operationen notwendig. Zusätzlich wird es noch notwendig Listen von Bildern zurückzugeben, welche über den Namen der Ressource ohne eine UID definiert sind. Im Fall der Image Ressource bedarf es außerdem noch die Möglichkeit ein Bild als Favorit zu markieren. Hierfür wird das Verb *like* an den PUT Request angehängt.

Für den Abruf einer Liste von Bilder anhand von Geokoordinaten, wird eine weitere Route benötigt.
Die *geobox* Route nimmt als Parameter zwei Eckpunkte eines Rechtecks in Form von Tupeln aus Breitengrad (lat) und Längengrad (lon) an und gibt eine Liste von Bildern innheralb der Fläche zurück. Lat und Lon können als x und y Koordinaten auf einem Koordinatensystem betrachtet werden.

Tabelle 5.1

Methode	Ressource	URI	Funktion	Auth
GET	Image	/image	Liefert eine Liste von Images	Nein
POST	Image	/image	Erstellt ein neues Image Objekt	Ja
GET	Image	/image/{id}	Liefert ein einzelnes Image	Nein
PUT	Image	/image/{id}	Ändert Daten an einem Image	Ja
DELETE	Image	/image/{id}	Löscht ein Image	Ja
PUT	Image	/image/{id}/like	Markiert Image als Favorit	Ja

Tabelle 5.2

Methode	Ressource	URI	Auth
GET	Image	/image/geobox?southwest={lat};{lon}&northeast={lat};{lon}	Nein

5.3.2.2 User

Die Ressource User ermöglicht es einen Nutzer anzulegen und zu verwalten. Zusätzlich werden noch Funktionen für die Authentifizierung benötigt. Da Operationen am der User Ressource nur vom authentifizierten Nutzer vorgenommen werden können, ist es nicht notwendig die Ressource über eine UID in der URI zu indentifizieren. Der Access Token dient in diesem Fall als Identifikation.

Tabelle 5.3

Methode	Ressource	URI	Funktion	Auth
GET	User	/user	Liefert den eingeloggten User	Ja
POST	User	/user	Erstellt neuen User	Nein
PUT	User	/user	Ändert Daten des Users	Ja
DELET	User	/user	Löscht den User	Ja

Für die Generierung des Access Tokens wird eine Login Route benötigt, ebenso für das entfernen des Tokens und dem Beenden der Sitzung in Form eines Logouts

Um eine öffentliche Ansicht des Users zu ermöglichen wird eine Ressource benötigt welche auch von nicht angemeldeten Usern eingesehen werden kann. Dies wird in *profile* umgesetzt, mit den jeweiligen Subressourcen des User. Als Identifier wird in diesem Fall der Username gewählt, da dieser leicht im Gedächtnis bleibt.

Tabelle 5.4

Methode	Ressource	URI	Funktion	Auth
POST	User	/user/login	Erstellt Access Token & startet die Session	Nein
GET	User	/user/logout	Löscht Token & beendet die Session	Ja

Tabelle 5.5

Methode	Ressource	URI	Funktion	Auth
POST	User	/user/profile/{username}	Öffentliches Profil des Users	Nein
GET	User	/user/profile/{username}/uploads	Uploads des Users	Nein
GET	User	/user/profile/{username}/likes	Favoriten des Users	Nein

5.4 Repräsentation

Für die Darstellung der Ressource in der Response muss ein maschinenlesbares Format gewählt werden. Grundsätzlich ist es möglich verschiedene Formate als Repräsentation auszugeben, im Kontext dieser Arbeit soll der Einfachheithalber nur ein einziges Nachrichtenformat für Request und Response festgelegt werden.
Javascript Object Notation (JSON) und Extended Markup Language (XML) sind übliche Formate für den Austausch von Daten zwischen Systemen über HTTP. Da der Browser-Client von Jan-Hendrik Kruse [Kru16] in Javascript geschrieben ist, liegt es nahe JSON zu verwenden. Das JSON Format stellt ein valides Javascript-Objekt dar und kann ohne Zwischenschritte im Code verwendet werden.
Weiter sollte eine allgemeine Definition für den Inhalt der Nachrichten gefunden werden. Es macht Sinn, Daten und Meta-Informationen zu trennen, ein Enwurf für ein JSON Schema könnte folgendermaßen aussehen:

```
{
  "timestamp": "[...]",
  "code": "[HTTP-Statuscode]",
  "data": "[Objekt oder Array von Objekten]"
}
```

Für die Darstellung von sehr vielen Objekten wird eine Paginierung der Daten benötigt. Eine Möglichkeit der Umsetzung wäre die Verwendung von Links als Steuerinformationen für die Liste:

```
{
```

```
  "links": {
    "first": "/resource?page=1",
    "previous": "/resource?page=1",
    "current": "/resource?page=2",
    "next": "/resource?page=3",
    "last": "/resource?page=3"
  },
  "timestamp": "[...]",
  "code": "[...]",
  "data": "[...]"
}
```

Eine weitere Repräsentation für Fehlermeldungen ist notwendig. Hierüber können Client- Server oder Validierungsfehler dargestellt werden:

```
{
  "timestamp": "[...]",
  "code": "[...]",
  "message": "[HTTP Status Message]",
  "errors": "[Array von Fehlern]"
}
```

Welche Daten als Repräsentation der Ressource an den Client geschickt werden soll in der konkreten Umsetzung des Prototyps im folgenden Kapitel definiert werden.

Kapitel 6

Implementierung und Prototyp

Der im vorherigen Kapitel erarbeitete Entwurf und die Archtiktur sollen nun in Form eines Prototypen umgesetzt werden. Im Folgenden Teil der Arbeit werden die Einzelheiten der Umsetzung dokumentiert und veranschaulicht, der Fokus liegt hierbei auf der Erfüllung der gestellten Anforderungen aus 3.2. Die Web-API wurde in PHP mit Hilfe des Symfony Frameworks umgesetzt, zusätzlich wurden noch einige PHP Erweiterungen verwendet und über den Composer Paket Manager eingebunden. Eine genauere Erklärung zu den technischen Hilfsmitteln findet sich im Folgenden Abschnitt.

6.1 Technische Hilfsmittel und Frameworks

Hier sollen die verwendeten Anwendungen und Frameworks des System aufgelistet und ihr Sinn und Zweck kurz erklärt werden.

- **Nginx Webserver & PHP-FPM**
  Für das empfangen und versenden von HTTP Nachrichten wird der Nginx Webserver verwendet. Nginx ist modular aufgebaut, kann gut mit hohen Lasten umgehen und besitzt eine Vielzahl an Konfigurationsmöglichkeiten. Als FCGI Schnittstelle zu PHP wird PHP-FPM über TCP verwendet, eine Umsetzung die ebenfalls geeignet ist für hohe Lastanforderungen, eine Anbindung über einen unix-socket wäre ebenso denkbar.
- **PHP & Symfony**
  Aufgrund der Anforderungen an die Stabilität soll auf die neueste PHP Version 7 verzichtet werden und das derzeit stabile PHP 5.6 verwendet werden. Als Applikationsgerüst wird das Symfony Framework eingesetzt. Bei Symfony handelt es sich um ein aus Komponenten aufgbautes Framwork, welches in der Grundausfühung aus

Paketen für Caching, Events, Routing, Controller und diversen weiteren Hilfsmitteln besteht. Sehr nützlich für Web-Applikationen sind die Abstraktionen für Request- und Responseobjekte. Das Framework kann über den Composer Paketmanager aus aktuellen Quellen erweitert werden, mehr zur Symfony Appikation findet sich unter 4.4.

- **MongoDB Datenbank & Doctrine ODM**
  Als Datenbank wird die schemafreie NoSQL-Datenbank MongoDB verwendet. MongoDB arbeitet nicht-relational auf Basis von BSON Dokumenten und eignet sich für leselastige Anwendungen und denormalisierte Datenmodelle. Für die Grafflr API sind vor allem die raumbezogenen Query Funktionen und die Indexierung nach Geo-Koordianten von Bedeutung. Für das Mapping von Objekten auf Dokumente wird Doctrine als Object Document Manager (ODM) eingesetzt.
- **Redis In-Memory-Datenbank**
  Die Funktion des in 5.1.4 und 5.2 erwähnten Token-Storage übernimmt ein Redis Key-Value Store. Redis kann als Cluster betrieben werden und hat durch die Speicherung im Arbeitsspeicher sehr geringe Zugriffszeiten.

Neben den erwähnten Server Komponenten wurden folgende Symfony Pakete zur Entwicklung der API verwendet:

Tabelle 6.1

Paket	Funktion
nelmio/api-doc-bundle	Generierung einer API Dokumentation aus Annotations
nelmio/cors-bundle	Konfiguration von Cross-Origin Resource Sharing Headern
friendsofsymfony/user-bundle	Funktionen für User Management
vich/uploader-bundle	Hilfsmittel für Medien Uploads
jms/serializer-bundle	Erweiterte Serialisierung

Die Referenzen auf Dateien im Quellcode sind relativ zum beschriebenen Namespace, mit Außnahme der Symfony Konfigurationen. Der Namespace *grafflrAPIBundle* referenziert das Quellverzeichnis *src/grafflr/Bundle/APIBundle*.

6.2 Vorbereitung

Um die einzelnen Komponenten nutzen zu können, müssen die Module im Framkework konfiguriert werden. Dies geschieht über die Yaml Dateien in app/config und kann sich

für Entwicklungs- und Produktionsumgebung unterscheiden. Für die Grafflr API muss die Verbindung zur MongoDB Datenbank und zum Redis Storage konfiguriert werden. Weitere Einstellungen werden für die einzelnen Bundles benötigt.
Der nächste Schritt ist die Erstellung eines Bundles für den eigenen Applikationscode. Symfony verwendet den Begriff Bundle für Module und Komponenten, grundsätzlich besteht das gesamte Framework aus einer Zusammenstellung von Bundles. Eigener Code liegt im /src Verzeichnis, damit der Code geladen wird muss das Bundle in /app/AppKernel.php registriert werden. Nach diesen Vorbereitungen kann die eigentliche Umsetzung beginnen.

6.3 Modell

Die beiden Entitäten aus 5.3.1 werden als zwei Klassen umgesetzt. Das Mapping der Attribute kann mithilfe der Doctrine ODM als Metadaten in Annotations definiert werden. Ebenso können durch Annotations für den JMS Serializer Attribute als serialisierbar und öffentlich definiert werden.

Attribut	**Typ**	**MongoDB Typ**	**Exposed**
Id	mongoId	Id	Ja
createdAt	date	date	Ja
updatedAt	date	date	Ja
path	string	string	Nein
owner	:User	Embedded	Ja
location	:Location	Embedded	Ja
description	string	string	Ja
imageName	string	string	Ja
likes	array	collection	Nein
likeCount	int	int	Ja
can_like	bool	bool (not saved)	Ja

Tabelle 6.2: Attribute und Annotations der Image Klasse

Das Location-Attribut wird als Embbedded Object umgesetzt. Dies ermöglicht es eine eigene Klasse mit Subattributen zu definieren und einzubinden.

Attribut	Typ	MongoDB Typ	Exposed
lat	float	float, index	Ja
lon	float	float, index	Ja
country	string	string	Ja
city	string	string	Ja
district	string	string	Ja
street	string	string	Ja
streetNumber	string	string	Ja

Tabelle 6.3: Attribute und Annotations des Location Attributs

Attribut	Typ	MongoDB Typ	Exposed
Id	string	MongoId	ReadOnly
createdAt	date	date	Ja
updatedAt	date	date	Ja
username	string	string	Ja
plainPassword	string	not saved	No
email	string	string	Ja
newEmail	string	string	Ja
passwordToken	string	string	Ja
isConfirmed	bool	bool	Ja
likes	array	collection	Ja

Tabelle 6.4: Attribute und Annotations der User Klasse

Die User Klasse erbt von der BaseUser Klasse des FosUserBundles. Dadurch lassen sich Authentifizierungsfunktionen nutzen ohne die Klasse an die jeweiligen Interfaces anzupassen. Neben den beschiebenen Annotations in 6.4 werden Assertions für die Input-Validierung von username, password und email genutzt.

Die Verbindung zwischen Datenobjekten und den Dokumenten in der Datenbank wird von einem Doctrine Mapper verwaltet. Für die User und Image Klasse wurden zwei eigene Repositories geschrieben, damit können wiederkehrende Queries und Funktionen zur Datenbank schnell und ohne großen Aufwand verwendet werden.

6.4 Authentifizierung

Nun soll die in Punkt 5.1.4 erwähnte Authentifizierung am Prototyp umgesetzt werden. Hierfür wird ein Symfony Authentication Provider[1] verwendet, ein vorgegebenes Interface, mit dessen Hilfe sich eine eigene Authentifizierungsmethode in die Symfony Firewall integrieren lässt.
Das grundsätzliche Prinzip wird in 6.1 dargestellt. Beim Aufruf der Login Route und einer erfolgreichen Validierung von Passwort und Username wird ein Auth-Token erzeugt. Dieser wird im Redis Token Storage abgelegt und in einem Cookie an den Client zurückgesendet. Bei jedem folgenden Request des Clients wird der gesendete Token mit den gespeicherten Tokens abgeglichen und der User gegebenenfalls authentifiziert. Die Sitzung, also der Token, bleibt bestehen bis der Client die Logout Route aufruft, welche den Token im Storage und im Cookie löscht oder die Time to Live (TTL) des Tokens erreicht wird. Ist dies der Fall muss erneut die Login-Route aufgerufen werden.

[1] grafflrAPIBundle/Security/MultipleAccessUserProvider.php

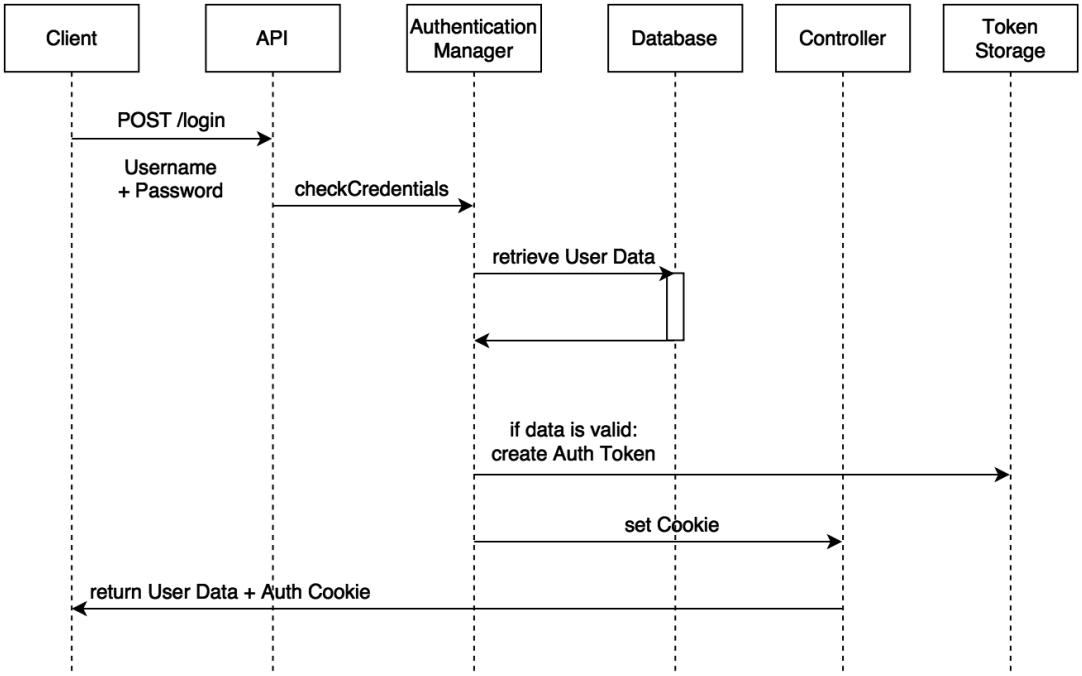

Abbildung 6.1: Sequenzdiagramm des Logins.

Bei jedem Request auf eine in der Firewall als geschützt definierte Route kann der Client nun an dem gesendeten Token identifiziert werden.
Im Fall der Grafflr API soll es möglich sein auf verschiedenen Endgeräten gleichzeitig eine Sitzung zu starten. Der Authentication Manager generiert aus diesem Grund einen Token pro Client-Anwendung. hierfür wird bei der Verarbeitung des Requests durch den Auth. Listener[2] der custom Header *X-GRAFFLR-CLIENT* überprüft und ein korrespondierender Token erzeugt.

6.5 Controller

Das Datenmodell und die Authentifizierung sind umgesetzt, es kann nun mit der Implementierung der Basisfunktionalität begonnen werden. Hierfür wird das Routing der Ressourcen auf Controller Funktionen gemappt. Der Controller gibt die Response an den

[2] grafflrAPIBundle/Security/MultipleAccessListener.php

Kernel zurück und dieser an den Client.
Für die Umsetzung der Funktionalität wurden drei Controller[3] Klassen angelegt, welche von der Symfony Basis Controller Klasse erben (s. Abb. 6.2). Der BaseRestController dient dazu den erbenden Klassen das jeweilige Repository zur Verfügung zu stellen und enhält die Response Funktionen der verschiedenen Repräsentationen. Für die beiden Entities wurde jeweils ein Controller erstellt.

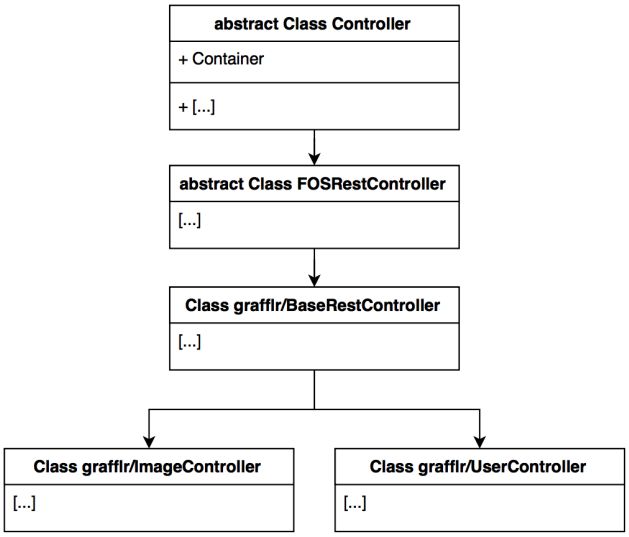

Abbildung 6.2: Vererbung der Grafflr Controller

Damit die Controller beim Request Matching berücksichtigt werden, sollten sie in der Routing Konfiguration registriert werden[4]. Die Actions im Controller können nun durch Annotations auf Routen gemappt werden. Folgenes Mapping stellt bspw. die Annotations für die /image{id} Route dar:

```
/**
 * Single Image by Id
 * @Route("/image/{id}")
 * @Method("GET")
**/
```

[3] grafflrAPIBundle/Controller/
[4] app/config/routing.yml

```
public function getImageSingleAction($id)
   {...}
```

Die @Method() Annotation steht hierbei für die zu verwendende HTTP Methode. Auf diese Weise können die in in Punkt 5.3.2 beschriebenen Ressourcen-Mappings nacheinander umgesetzt werden.
In 5.3.2 wurde die Notwendigkeit einer Authentifizierung für bestimmte Ressourcen erwähnt. Dies kann in der Symfony Firewall[5] definiert werden und über Regular Expressions kann eine Route durch die jeweilige Firewall geschützt werden. In Symfony sind standardmäßig die Rollen *IS_AUTHENTICATED_ANONYMOUSLY* und *ROLE_USER* definiert, diese werden als Zugangsvorraussetzung einem Pattern zugeordnet. Wurde der Request authentifiziert wurde die Rolle ROLE_USER zugeordnet.
Im folgenden Abschnitt wird wegen des Umfangs auf die Beschreibung jeder einzelnen Action verzichtet werden, für genauere Informationen kann hierfür der Quellcode auf der DVD hinzugezogen werden. Stattdessen soll die Verarbeitung eines Requests schematisch erklärt werden und die zugezogenen Funktionen am Beispiel der *getImageSingleAction()* kurz erläutert werden.

```
145 public function getImageSingleAction($id)
146 {
147
148     $image = $this->getRepository()
149         ->findImageById($id);
150
151     if (!$image) {
152         return $this->errorResponse(
153             new ErrorRepresentation(
154                 ErrorRepresentation::NOT_FOUND
155             ),
156             Response::HTTP_NOT_FOUND
157         );
158     }
159     return $this->successResponse($image);
160 }
```

Je nach Methode werden die gesendeten Parameter aus dem Request und der Route ex-

[5] app/config/security.yml

trahiert, im Beispiel wird der Routenparameter *id* als UID eines Image Objekts verwendet. Durch die Elternklasse BaseRestController ist das benötigte Repository im Konstruktor gesetzt worden und im Controller verfügbar. Je nach Operation wird die entsprechende Query-Funktion verwendet, in diesem Fall die durch die in der ImageRepository Klasse definierte findImageById($id) Funktion. Im nächsten Schritt in Zeile 251 wird überprüft, ob die Ressource gefunden wurde. Im Fall von *false* wird die Representation *ErrorRepresentation::NOT_FOUND* geladen und zurückgegeben. Im Fall von *true* wird das Objekt durch *successResponse($data)* zu einem JSON Dokument serialisiert und mit dem HTTP Code 200 (*success*) an den Client geschickt. Abb. 6.3 zeigt ein Flowchart zur schematischen Verarbeitung einer CRUD Operation.

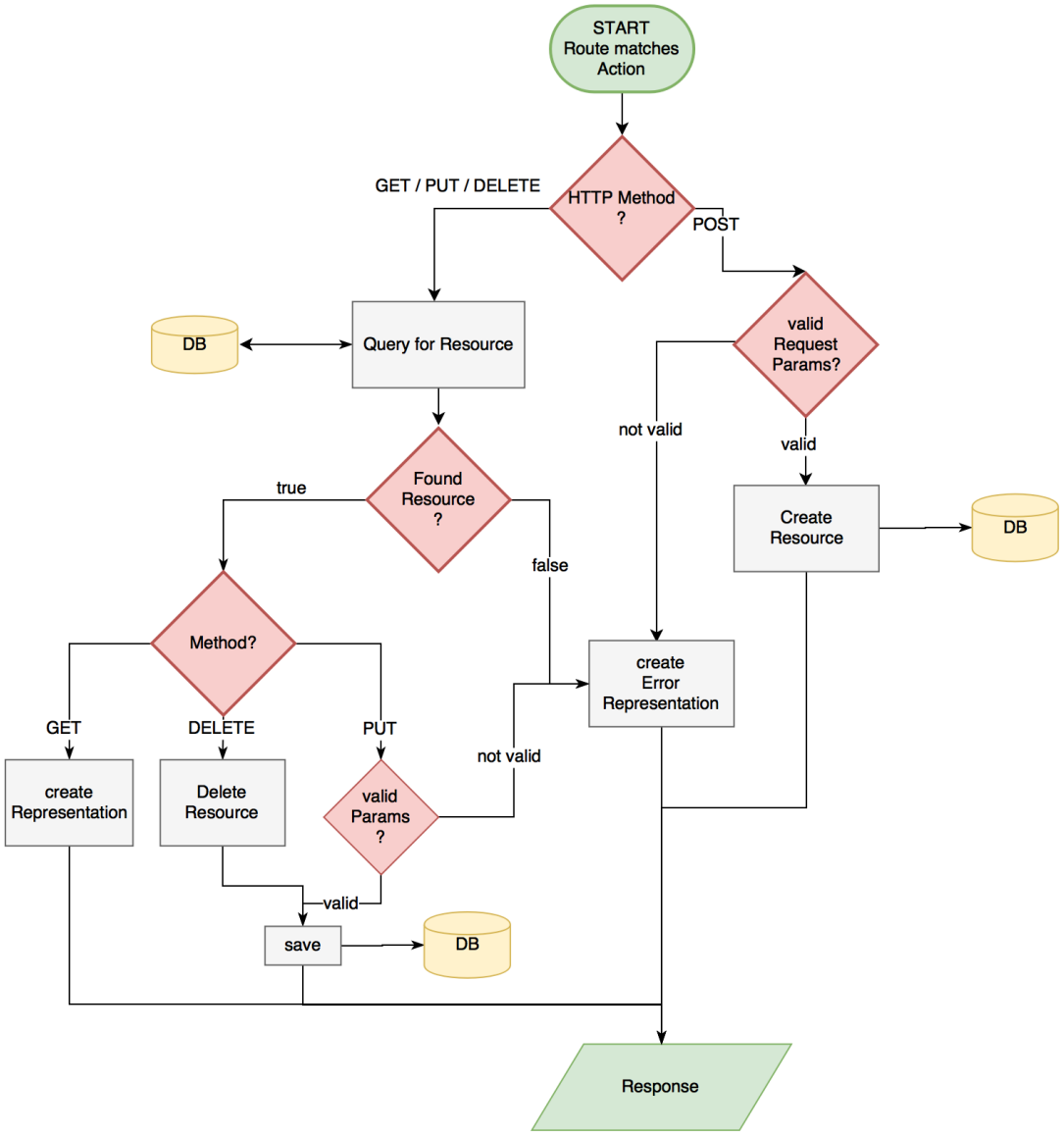

Abbildung 6.3: Flowchart der Controller CRUD Actions.

6.5.1 Geobox

Eine spezielle Operation stellt der Query nach Bildern anhand von Geokoordinaten dar. Als Ressource dient hierfür der Endpunkt */image/geobox*, welcher Bilder innheralb eines räumlichen Rechtecks abruft. Der Ablauf ist grundsätzlich derselbe wie im Beispiel */image*, allerdings werden die Parameter zu Längen- und Breitengrad aus den URL-Parametern des Requests extrahiert. Ein Request an den Geobox Endpunkt kann folgendermaßen

aussehen:

```
GET http://api.grafflr.com/image/geobox/?southwest=53.565;10.029&northeast
    =53.573;10.041
```

Die Parameter *southwest* und *northeast* beschreiben die Endpunkte der Diagonalen eines Rechtecks, der erste Wert der Parameter definiert die geografische Länge, der zweite die geografische Breite, in diesem Fall ein Bereich um den Campus Finkenau mit einer Seitenlänge von je ca. 850 Metern. Mit den übergebenen Punkten kann nun ein Query an die Datenbank geschickt werden, welcher anhand der indexierten *Location* Property alle Bilder im Bereich der Fläche findet. Für den Query kann die Funktion *findByGeoBox()* des ImageRepositorys[6] verwendet werden.

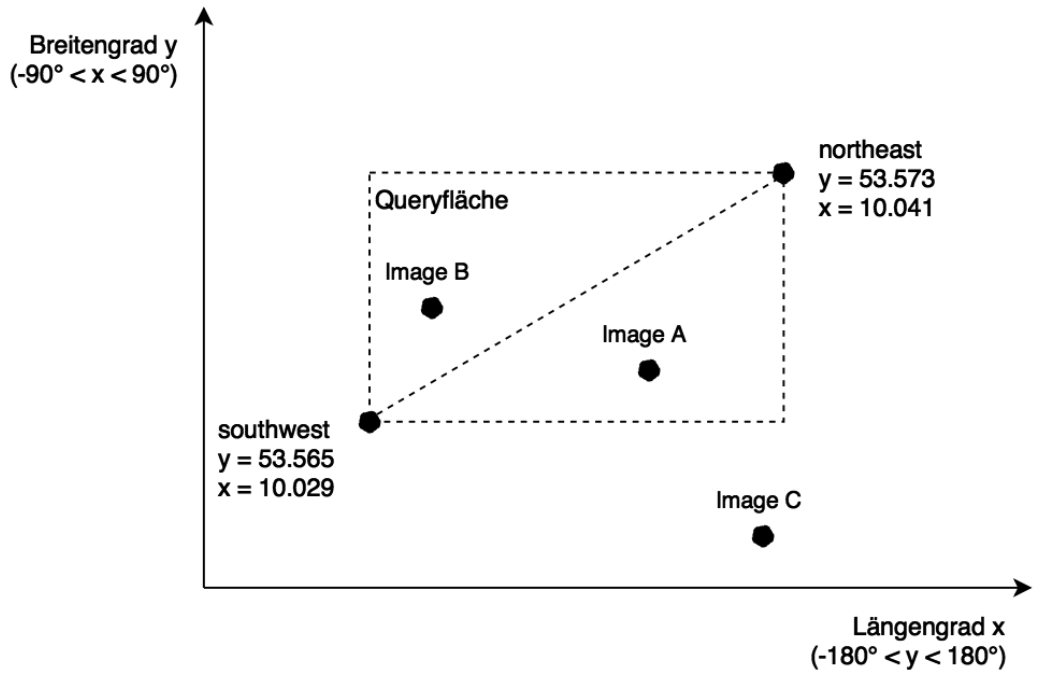

Abbildung 6.4: Query nach Bildern innerhalb einer Fläche

[6] grafflrAPIBundle/Repository/ImageRepository.php

6.5.2 Upload

Eine weitere spezielle Funktion ist der Upload von Bildern und den zugehörigen Geo-Koordinaten. Die Besonderheit in diesem Fall besteht im Empfang und der Verarbeitung einer binären Bilddatei, im Gegensatz zu den sonstigen Anfragen mit simplen JSON Strings als Payload. Theoretisch besteht die Möglichkeit das Bild als JSON zu senden, hierfür würde der Client das Bild in Base64 codieren und als String in einer JSON Nachricht an die API senden, allerdings geht dies mit einem größeren Volumen der Payload einher und ist daher keine ideale Lösung. Eine weitere Möglichkeit besteht in der Aufteilung der Nachricht in Datei und Metadaten. Für den Upload wird der HTTP Content-Type *multipart/form-data*[7] verwendet. Hierbei können im Request-Body verschiedene Content-Typen definiert werden und durch sogennante *Form-Boundaries* voneinander abgegrenzt werden. In diesem Fall wird in der Nachricht das Bild als Content-Type der jeweiligen Bilddatei definiert, z.B. *image/jpeg*. Die Metadaten des Bildes werden Im JSON Format in einem weiteren, durch eine Boundary getrennten Teil der Nachricht übertragen.
Auf Serverseite können nun die Daten aus dem Request ausgelesen werden. Zuerst wird anhand der deserialisierten Daten der JSON Payload ein neues Image Objekt erstellt, im Anschluss wird die Bilddatei auf der Festplatte gespeichert und der Pfad zur Datei am Image Objekt hinterlegt. Das Objekt kann mit diesen Daten in der Datenbank abgelegt und eine Success-Response an den Client gesendet werden.

```
POST /images HTTP/1.1
Host: localhost:8000
Accept: application/json
Content-Type: multipart/form-data; boundary=----WebKitFormBoundary7MA4YWxkTrZu0gW
----WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Disposition: form-data; name="image"; filename="grafflr-logo.jpg"
Content-Type: image/jpeg
----WebKitFormBoundary7MA4YWxkTrZu0gW
Content-Disposition: form-data; name="payload"
{
    "description":"test-description",
   },
   "location":{
     "lat":53.4568345,
```

[7] https://www.ietf.org/rfc/rfc2388.txt

```
        "lon":9.99123
      }
    }
    ----WebKitFormBoundary7MA4YWxkTrZu0gW
```

Listing 6.1: Beispiel einer HTTP Nachricht für den Bild-Upload

6.6 Repräsentation

Als Repräsentation einer Ressource gilt das Format in der sie ausgegeben wird. Da eine Ressource nur ein gedankliches Konstrukt darstellt, wäre es theoretisch möglich jedes gewünschte Format zurückzugeben. In Kapitel 5.4 wurde bereits festgelegt das im Fall der Grafflr API ausschließlich JSON als Content-Type der Representation zurückgegeben wird. Ein weiterer sinnvoller Schritt besteht darin, eine festgelegte Datenstruktur der JSON Dokumente zu definieren. Auf diese Weise wird über mehrere Ressourcen hinweg, konsistentes Verhalten erreicht und die Interaktion vereinfacht.
Die standardmäßige Repräsentation einer Ressource der Grafflr API sind die Felder *timestamp* und *data*, wobei letzteres die angeforderten Daten beinhaltet. Neben diesem Format gibt es noch Fälle die ein spezielles Format benötigen.

6.6.1 Paginierung

Listen von Bildern können in kürzester Zeit sehr groß werden. Eine Liste von 20 Image Objekten hat bereits eine Größe von 15 Kilobyte. Wird keine Art der Begrenzung von Listen eingesetzt, ist eine Liste von 2000 Objekten mit 1,5 Megabyte nicht unwahrscheinlich und kann zu langen Response-Zeiten führen. Ein einfacher Weg durch Listen zu navigieren stellt die Umsetzung einer Paginierung dar. Nach diesem Prinzip wird die gesamte Liste in Teile von z.B. 20 Objekten je Dokument aufgespalten und Metadaten zur Navigation durch den Datensatz mitgeschickt. In der Grafflr API wurde eine Paginierung durch Links in der Reponse bei den Ressourcen *image* und */image/geobox* umgesetzt.
Hierfür wird zuerst das derzeitige Dokument, sowie die Limitierung der Liste benötigt. Diese werden durch einen Request-Listener aus den URL-Paramter des Requests gelesen und im BaseRestController gesetzt. Auf diese Weise haben alle Controller-Klassen Zugriff auf die Parameter während der Verarbeitung. Im Query kann dadurch z.B. ein Offset und ein Limit der Abfrage angegeben werden. Die restlichen Felder können aus den Request-Parametern und der Gesamtgröße der Liste berechnet werden. Daraus ergibt sich folgender Aufbau des Dokuments:

```
  {
```

```
    "links": {
      "first": "/image?page=1",
      "previous": "/image?page=1",
      "current": "/image?page=2",
      "next": "/image?page=3",
      "last": "/image?page=4"
    },
    "pages": {
    "first": 1,
    "previous": 1,
    "current": 2,
    "next": 3,
    "last": 4
  },
    "timestamp": "[...]",
    "data": "[...]"
  }
```

6.6.2 Error

Ein weiterer Sonderfall ist die Darstellung der Fehlermeldung. In vielen Fällen müssen die erzeugten serverseitigen Fehler vom Client ausgelesen und dem Nutzer dargestellt werden, z.B. bei Validierungs- und Authorisierungsfehlern. Für die sichere Interpretation des Fehlers wird auf Konstanten in Form der HTTP Error Codes zurückgegriffen, sowie einer angepassten Fehlernachricht. Häufig auftauchende Fehler wie Validierung, Authorisierung und *404 - Resource not found* werden über Exception Events im Request-Listener[8] abgefangen und durch die *exceptionHandle($exception)* Funktion im BaseRestController verarbeitet.

6.6.3 Response Processor

In einem Sonderfall muss die Repräsentation an den Zustand der Sitzung angepasst werden. Es soll nur eingeloggten Usern möglich sein Bilder zu favorisieren, ebenso sollte die Client-Anwendung wissen, welche Bilder vom eingeloggten User bereits favorisiert wurden. Dies wird über das nicht gespeichertes Feld *can_like* am Image-Objekt dargestellt.

[8] grafflrApiBundle/EventListener/Requestlistener.php

Vor der Rückgabe von Images wird in der Klasse *ResponseProcessor*[9] überprüft ob der User eingeloggt ist. Handelt es sich um einen anonymen Nutzer wird an allen Images das *can_like* Feld auf *false* gesetzt. Bei einer Anfrage eines eingeloggten Users werden über die am User gespeicherten Favoriten überprüft, welche Bilder bereits favorisiert wurden. Bei allen anderen wird *can_like* auf *true* gesetzt. Auf diese Weise lassen sich auch in Zukunft Felder anhand des Zustandes eines User Objekts setzen.

```
{
    [...],
    "data": [
      {
        "id": "1",
        [...]
        "like_count": 15,
        "can_like": false
      },
      {
        "id": "2",
        [...],
        "like_count": 8,
        "can_like": true
      }
    ]
  }
```

Listing 6.2: Bild 1 wurde bereits favorisiert, Bild 2 noch nicht

6.7 API Dokumentation

Durch die freie Definition von Ressourcen im REST Paradigma ist eine vollständige Dokumentation der API zwingend notwendig, da von der Clientseite aus nicht nachvollziehbar ist, welche Entitäten oder Services sich hinter der Definition einer Ressource verbergen. Das Minimum einer Dokumentation stellt dar:

- verfügbare Ressourcen und Endpunkte
- verfügbare HTTP Methoden einer Ressource und notwendige Authentifizierung

[9] grafflrApiBundle/Service/ResponseProcessor.php

- notwendige und optionale Parameter eines Endpunkts
- Format und Werte der Repräsentation einer Ressource

Eine Umsetzung im Format der *Open API Specification (OAS/Swagger)*[10] erscheint sinnvoll. Das Swagger Framework ermöglicht es anhand eines in JSON oder YAML verfassten Dokuments eine API komplett zu beschreiben und in verschiedene Formate zu portieren und darzustellen. Die Grafflr API Docs wurden mithilfe des NelmioAPIDocs Bundle erstellt. Das Bundle ermöglicht es per Annotations einzelne Controller Acions zu beschreiben und generiert anschließend eine HTML Dokumentation. Es wird der Swagger Syntax verwendet, ebenso kann ein vollständiges Swagger Dokument über Konsolenbefehle generiert werden. Die Dokumentation kann unter */api/doc* eingesehen werden.

[10] URL: http://swagger.io/ (besucht am 07. 06. 2016).

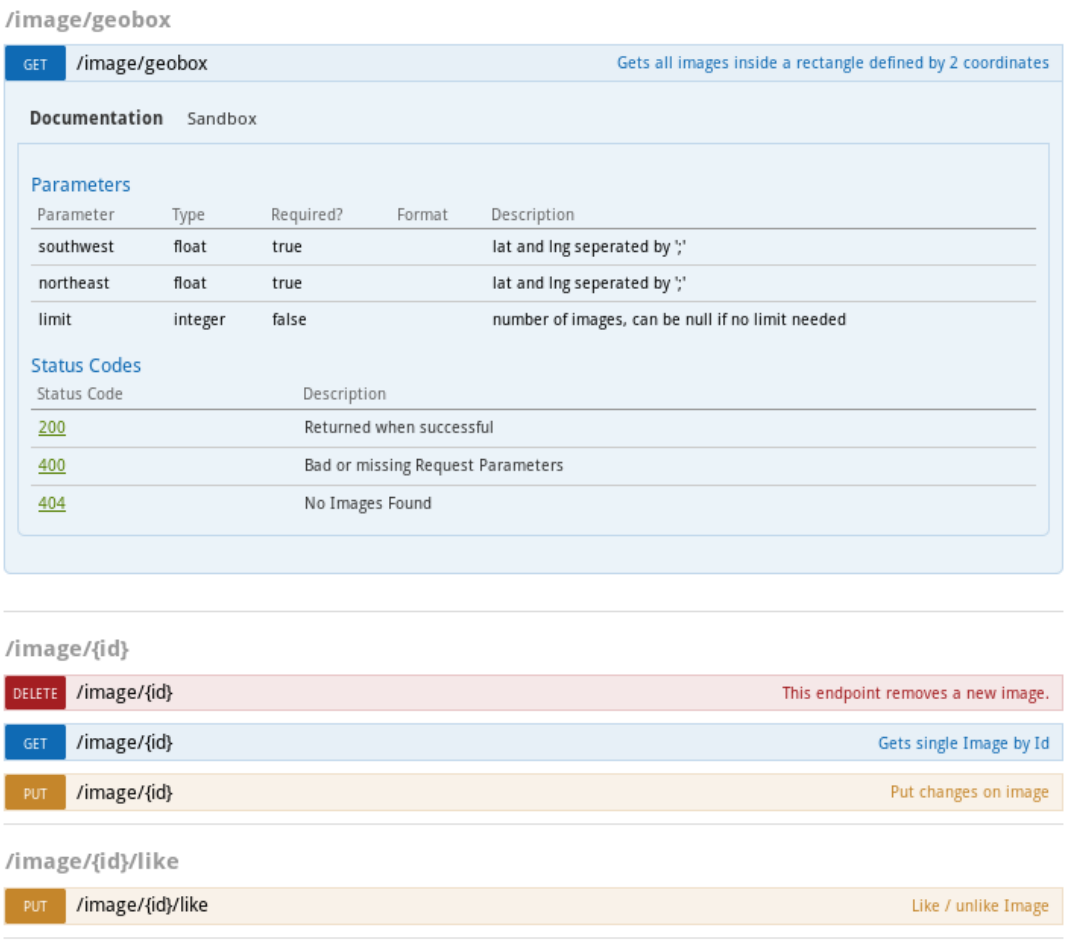

Abbildung 6.5: Generierte API Dokumetation aus Swagger Annotations

Kapitel 7

Zusammenfassung

In diesem letzten Abschnitt der Arbeit soll der erarbeitete Prototyp kurz evaluiert werden und ein Fazit zur Arbeit formuliert werden. Abschließend wird die mögliche weitere Entwicklung der API aufgezeigt.

7.1 Evaluation

Anhand der Anforderungen aus Kapitel 3 kann zum Abschluss die umgesetzte Funktionalität geprüft werden.

7.1.1 Funktionale Anforderungen

1. Die Abfrage von Bildern innerhalb eines räumlichen Rechtecks wurde mit der */image/geobox* Ressource erfüllt. Die Forderung eines Queries anhand der Geokoordinaten der Eckpunkte wurde umgesetzt.
2. Die Rückgabe einzelner Bilder wurde anhand der */image/{id}* Ressource implementiert.
3. Die */user* Ressource stellt ein rudimentäres Usermanagement dar. Ein neuer Nutzer kann über einen *POST* Request angelegt werden, Daten des Users können über *PUT* geändert werden und über *DELETE* kann ein User gelöscht werden.
4. Durch die Generierung eines Security-Tokens in Punkt 6.4 und der Rückgabe im Cookie, ist durch einen POST Request an */user/login* eine Authentifizierung des Users möglich.
5. Die */image* Ressource implementiert alle CRUD Operationen. Der Upload einer Bilddatei, sowie das Anlegen eines neuen Image Objekts ist durch einen POST Request

mit "*Content-Type: multipart/form-data*" und dem Bild, sowie den Metadaten im Body, möglich.

6. Das Favoriseren eines Bildes ist durch einen leeren PUT Request an den Endpunkt */image/{id}/like* möglich, ebenso das Entfernen der Markierung als Favorit. Für diese Operation muss der User authentifiziert sein.
7. Durch die Rückgabe von öffentlichen Nutzerdaten durch die Ressource */user/profile/{username}* ist die Einsicht von Profilen anderer Nutzer auch ohne Authentifizierung möglich. Die Ressourcen */user/profile/{username}/uploads* und */user/profile/{username}/likes* werden als Links mitgeschickt und stellen die Favoriten und Uploads des Users dar.

Die funktionalen Anforderungen an die Anwendung können somit als erfüllt gelten.

7.1.2 Nicht-funktionale Anforderungen

Neben dem grundsätzlichen Erreichen der Basisfunktionalität besteht noch die Frage, ob die Art der Umsetzung zufriedenstellend erfolgt ist.

- **Einfachheit**: In Bezug auf die Alternative zu SOAP kann dieser Punkt als erfüllt angesehen werden. Die Ressourcen sind nach reproduzierbaren Kriterien gewählt und es ist nicht notwendig ein weiters Protokoll außer HTTP zu beherrschen. Jeder Client der über eine HTTP Bibliothek verfügt, kann mit der API kommunizieren.
- **Stabilität**: Die Stabilität, also die Garantie zu jeder Zeit von einem Endpunkt die gleiche Repräsentation zu erhalten, ist durch Ressourcen nach dem REST Schema und einer Versionierung der API gewährleistet.
- **Performance**: Mit Response-Zeiten unter 100 Millisekunden, ohne eine Optimierung für Produktionsumgebungen, ist die Performance unter geringer Last akzeptabel.
- **Sicherheit**: Diese Anforderung wurde nicht erfüllt. Eine HTTPS Verbindung wurde aus zeitlichen Gründen nicht für den Prototypen umgesetzt. Ein weiteres Sicherheitsrisiko stellt der unverschlüsselte Authentifizierungscookie dar.
- **Dokumentation**: Die Dokumentation der API wurde über die Swagger Spezifikation umgesetzt und ist in verschiedenen Formaten verfügbar. Allerdings ist derzeit keine versionsspezifische Dokumentation vorhanden.

7.1.3 Probleme

Obwohl die im Vorraus gestellten Anforderungen an die Anwendung im allgemeinen zufriedenstellend erfüllt wurden, gab es während der Entwicklung und Umsetzung doch einige unvorhergesehen Problematiken und Fehler.

Die paginierte Rückgabe der Ressource Geobox scheint in der bisherigen Umsetzung nicht praktikabel. Bei größeren Flächen macht eine Limitierung auf 20 Bilder keinen Sinn, da auf Clientseite selbst bei einer Fläche von einem ganzen Kontinent nur 20 Bilder angezeigt werden. Der User wird im ersten Moment annehmen, dass nicht mehr Bilder vorhanden sind. Gleichzeitig ermöglicht die Umsetzung keinen Überblick wo Hotspots, also Regionen mit großer Aktivität, liegen. Hier wird ein Clustering System notwendig, welches auf Client- oder auf Serverseite alle Bilderkoordinaten im Kartenausschnitt zu Clusterpunkten zusammenrechnet. Diese können dann durchaus begrenzt werden, da sie insgesamt immer noch die Gesamtheit der Bilder im Ausschnitt darstellen.

Ein weiteres Problem stellt die Speicherung der Bilder in Originalgröße dar. Bei Dateien von teilweise 4 Megabyte wird erstens zuviel Bandbreite bei der Übertragung genutzt und zweitens wird der Arbeitsspeicher des Clients stark belastet. Beim Upload eines Bildes sollten verschiedene Größen berechnet werden, z.B. 300 x 300 Pixel für Thumbnails und 800 x 800 Pixel für Detailansichten. Die Originaldatei kann für eine weitere Verarbeitung auf dem Server bleiben, da Speicherplatz vorerst kein Problem darstellt.

Für die Update Operation an Bildern wurde aus zeitlichen Gründen keine ordnungsgemäße Serialisierung umgesetzt, außerdem werden die Eingaben nicht validiert. Dies stellt ein Sicherheitsrisiko dar.

Eine weiteres Sicherheitsrisiko stellt die Authentifizierung über einen nicht-signierten Cookie dar. Um sicherzustellen das der Wert des Cookies nicht verändert wurde, sollte aus dem Wert des Cookies ein Keyed-Hash Message Authentication Code (HMAC) generiert werden. Dieser wird serverseitig validiert und eine Manipulation kann ausgeschlossen werden.

7.2 Fazit

Das Ziel dieser Arbeit war die Entwicklung einer Anwendung, welche es ermöglicht serverseitige Ressourcen über das Web zur Verfügung zu stellen. Als Konzept wurde das Prinzip

einer Web-API als serverseitige Schnittstelle gewählt. Aufgrund der höheren Komplexität in der Umsetzung fiel die Entscheidung gegen einen SOAP Webservice und für eine restful Web-API auf Basis von HTTP aus. Der Schritt zum ressorcenorientierten Prinzip von REST stellt einen zentralen Punkt der Arbeit dar. Das freie Design der Endpunkte und den damit verbundenen Ressourcen eröffnet die Möglichkeit, eine Schnittstelle schnell und unkompliziert zu entwickeln.
Es wurde die grundlegende Funktionalität der Serverseite eines Geotagging-Dienstes erarbeitet. Das Ergebnis orientierte sich hierbei nicht an der Vollständig der Funktionaliät, sondern soll die grundlegenden Szenarien des Systems abdecken. Auf Basis dieser Grundlagen wurde die schrittweise Erarbeitung eines Konzeptes für die Anwendung aufgezeigt. Zu diesen Schritten gehört die Überlegung, welche Constraints der REST Architektur in der Applikation Sinn machen und welche nicht, wann das Konzept ignoriert werden kann und wann es ignoriert werden sollte. Beispielhaft hierfür war der Bruch mit der Forderung nach der vollständigen Zustandslosigkeit von Applikationen. Durch die Notwendigkeit einer Authentifizierung gibt es in diesem Fall keinen Grund das Paradigma in dieser Form zu befolgen. Weiter wurde eine Umsetzung von REST an einer realen Anwendung aufgezeigt, durch die Entwicklung eines Mapping-Schemas für die benötigten Ressourcen des Dienstes.
Insgesamt kann die gewählte Umsetzung nach REST über HTTP als positiv bewertet werden. Das Prinzip besticht durch seine Einfachheit, was es ideal für rasche Entwicklungszyklen macht. Allerdings muss erwähnt werden das restful APIs, durch die fehlenden Spezifikationen eines Protokolls, eine Interpretation durch einen Entwickler benötigen. Da die Endpunkte und deren Funktionalität frei gewählt werden können, ist eine automatisierte Generierung von Client-Code oder eine automatische Einbindung in Systeme, anhand eines beschreibenden Dokuments wie z.B. WSDL, beinahe unmöglich. Ein Schritt in diese Richtung stellt die Beschreibung der Grafflr API anhand eines Swagger Dokumentes dar, welches nicht nur zur Generierung einer Dokumentation für Entwickler dienen kann, sondern auch grundlegenden Client-Code generieren kann.
Die Entwicklung der Basisfunktionalität der API waren vom zeitlichen Aufwand her akzeptabel. Ein größerer Teil der Zeit und Arbeit ist in die Entwicklung der Authentifizierung, sowie in die Formatierung der Repräsentationen geflossen. Auch diese Erkenntnis kann auf die Umsetzung in REST bezogen werden, da keine derartigen Funktionen vordefiniert sind und selbst implementiert werden müssen.
Die entwickelte API stellt nur ein Beispiel einer Vielzahl von möglichen Umsetzung dar. Die verwendete Technologien, abgesehen von HTTP, haben für die prinzipielle Umsetzung und Architektur einer Web-API keine Bedeutung und können nach Belieben durch mo-

dernere Implementierungen ausgetauscht werden. Die Ressourcen der API hingegen sind die Schnittstellen zur Applikation, welche nach außenhin unverändert bleiben sollte.

7.3 Ausblick

Die entwickelte API stellt lediglich das Backend des Grafflr Dienstes dar, die Clientanwendungen bestehen aus der erwähnten Web App von Jan-Hendrik Kruse [Kru16] und einer nativen Android App. Die Gesamtanwendung soll Ende 2016 für die Öffentlichkeit zugänglich gemacht werden, vorher wird allerdings noch einige Arbeit an der API notwendig. Die aufgezählten Probleme in 7.1.3 müssen behoben werden. Eine größere Aufgabe wird das Clustering System der Bilder sein, gefolgt von weiteren User-Interaktionen. Ein wünschenswertes Feature des Dienstes stellt eine Kommentarfunktion an der Bildern dar, welches es den Nutzern ermöglicht Diskussionen über die Bilder zu führen. Davon abgesehen muss vor der Veröffentlichung das Usermanagement überarbeitet und Funktionen wie Passwort Reset und Mailing umgesetzt werden. Ebenso benötigt der Dienst noch eine Suchfuktion um Bilder und Orte direkt abzufragen.
All diese Änderungen beziehen sich auf die Funktionen der Applikation. Aus Sicht der API kann das Schema so fortgesetzt werden, sodass die Schnittstelle lediglich um Endpunkte erweitert werden muss. An dem Konzept der bisherigen API muss in dieser Hinsicht auch in Zukunft nichts verändert werden.

Anhang A

Inhalt der CD-ROM

- **/code** In diesem Verzeichnis befindet sich der Quellcode der Anwendung. Die Datei README.MD gibt eine Anleitung zur Installation.
- **/latex** Hier befinden sich die LATEX-Dateien der Bacheloarbeit, sowie die verwendeten Bilder.
- **/BA_LMauritz.pdf** Die Bachelorarbeit im PDF Format.

Abbildungsverzeichnis

Quellenverzeichnis

[1] URL: http://www.programmableweb.com/protocol-api (besucht am 04.06.2016) (siehe S. 13).

[2] URL: https://tools.ietf.org/html/rfc7231 (besucht am 04.06.2016) (siehe S. 20).

[3] URL: http://swagger.io/ (besucht am 07.06.2016) (siehe S. 42).

[4] 2016. URL: https://www.w3.org/TR/soap12-part1/ (besucht am 04.06.2016) (siehe S. 15).

[5] 2016. URL: https://msdn.microsoft.com/en-us/library/orm-9780596527563-01-10.aspx (besucht am 04.06.2016) (siehe S. 16).

[6] Andrew Butterfield und Gerard Ekembe Ngondi. *A Dictionary of Computer Science*. URL: //www.oxfordreference.com/10.1093/acref/9780199688975.001.0001/acref-9780199688975 (siehe S. 11).

[7] *Die Google Maps Geocoding API*. 2016. URL: https://developers.google.com/maps/documentation/geocoding/intro?hl=de (besucht am 29.05.2016) (siehe S. 12).

[8] Roy Thomas Fielding. „Architectural styles and the design of network-based software architectures“. Diss. University of California, Irvine, 2000 (siehe S. 13).

[9] Khronos Group. *OpenGL API Documentation*. 2016. URL: https://www.opengl.org/documentation/ (besucht am 29.05.2016) (siehe S. 12).

[10] Jan-Hendrik Kruse. *Anwendungsorientierte App-Entwicklung zur Geolokalisierung von Fotos unter Einbindung von Social Media Technologien*. Bachelorarbeit. 2016 (siehe S. 6, 25, 48).

[11] D. L. Parnas. „On the Criteria to Be Used in Decomposing Systems into Modules“. In: *Commun. ACM* 15.12 (Dez. 1972), S. 1053–1058. URL: http://doi.acm.org/10.1145/361598.361623 (siehe S. 11).

[12] Ian Sommerville. *Software engineering*. Addison-Wesley, 2007 (siehe S. 7).

[13] *The Linux Kernel API*. 2016. URL: https://www.kernel.org/doc/htmldocs/kernel-api/ (besucht am 29. 05. 2016) (siehe S. 12).