

Konzeption und prototypische Entwicklung einer Social-Storytelling App für iOS

Abschlussarbeit

zur Erlangung des akademischen Grades Bachelor of Science (B.Sc.)

an der

Hochschule für Technik und Wirtschaft Berlin
Fachbereich Wirtschaftswissenschaften II
Studiengang Angewandte Informatik

1. Prüferin: Prof. Dr. Christin Schmidt
2. Prüfer: Dipl. Ing. (FH) Ingo Wiederoder

Eingereicht von:

Malte Schonvogel

Matrikel-Nr.: 538535
E-Mail: malte@schonvogel.de

Berlin, 17. Februar 2015

Inhalt

1 Einleitung	1
1.1 Motivation und Problemstellung	1
1.2 Zielstellung	1
1.3 Vorgehensweise	2
2 Grundlagen	3
2.1 Webservice	3
2.2 JSON	4
2.3 Node.js	5
2.4 Skalierbarkeit	8
2.5 Graph Datenbanken.....	9
2.6 Cloud Computing	12
2.7 Amazon Web Services	13
2.8 Smartphone-App	14
2.9 iOS Software Development Kit.....	15
2.10 Entwurfsmuster	17
2.11 ViewController	19
3 Analyse	23
3.1 Definition "Story"	23
3.2 Funktionale Anforderungen.....	23
3.3 Nichtfunktionale Anforderungen	26
3.4 Anwendungsfälle	28
3.5 Zielgruppe	28
3.6 Einsatzgebiete.....	29
4 Entwurf des Prototypen	30
4.1 Architektur	30
4.2 Datenhaltungsschicht.....	31
4.3 Anwendungsschicht	37
4.4 Präsentationsschicht - iPhone App	42
5 Realisierung des Prototypen	53
5.1 Werkzeuge & Entwicklungsumgebung.....	53
5.2 Anwendungsschicht	53
5.3 Datenhaltungs-Schicht	60
5.4 Präsentationsschicht.....	62

6 Tests	73
6.1 Webservice	73
6.2 iPhone-Applikation	75
7 Deployment	78
7.1 Konfiguration Anwendungsschicht	78
7.2 Konfiguration der Datenhaltungsschicht	80
8 Ergebnis	82
8.1 Zusammenfassung.....	82
8.2 Abschließende Betrachtung	82
8.3 Ausblick.....	83
A Literaturverzeichnis	85
B Glossar	93
C Eigenständigkeitserklärung	94
D Anhang	95

i. Abkürzungsverzeichnis

ACL	Access Control List
API	Application Programming Interface
AWS	Amazon Web Services
CDN	Content Delivery Network
CPU	Central Processing Unit
CRUD	Create Read Update Delete
DBMS	Database Managing System
DKIM	DomainKeys Identified Mail
ES6	ECMAScript 6
GDBM	Graph Database Managing System
HTML	Hypertext Markup Language
HTTP	Hypertext Transfer Protocol
IC	Inline Cache
JIT	Just in time
JSON	JavaScript Object Notation
LoC	Lines of Code
OLTP	Online Transaction Processing
RAM	Random Access Memory
ROA	Resource Oriented Architecture
SDK	Software Development Kit
SOA	Service Oriented Architecture
SOAP	Simple Object Access Protocol
UDDI	Universal Description, Discovery and Integration
URI	Uniform Resource Identifier
URL	Uniform Resource Locator
W3C	World Wide Web Consortium
WSDL	Webservice Description Language
TCP	Transmission Control Protocol
UDP	User Datagram Protocol

ii. Abbildungsverzeichnis

Abbildung 2.1 - Aufbau einer Zahl in JSON	5
Abbildung 2.3 - Beispielobjekt im JSON-Format.....	5
Abbildung 2.4 - Vergleich V8.....	6
Abbildung 2.5 - Eventloop in Node.js.....	7
Abbildung 2.6 - Koa Framework über NPM installieren.....	7
Abbildung 2.7 - Koa Framework über NPM installieren.....	8
Abbildung 2.6 - Beispiel eines Property-Graphen.....	10
Abbildung 2.7 - Cypherabfrage Bekanntschaften von Nutzern	11
Abbildung 2.8 - Cypherabfrage Bekanntschaften von Nutzern.....	12
Abbildung 2.9 - HTTP-Request zum Löschen einer Datei.....	14
Abbildung 2.10 - Organisation der iOS-Frameworks in vier Schichten.....	16
Abbildung 2.9 - Bei der iOS-Entwicklung zum Einsatz kommende Form des MVC-Patterns.....	17
Abbildung 2.10 - Schematische Darstellung eines Contentviewcontroller.....	19
Abbildung 2.11 - Collection Viewcontroller.....	20
Abbildung 2.12 - Containerviewcontroller	21
Abbildung 2.13 - View Hierarchie von Container Viewcontrollern.....	21
Abbildung 3.1 - Schematische Darstellung einer Story.....	23
Abbildung 3.2 - Anwendungsfälle der Applikation	28
Abbildung 4.1 - Systemarchitektur.....	30
Abbildung 4.2 - Drei-Schichten-Architektur.....	31
Abbildung 4.3 - logarithmisches Ranking der wichtigsten Graphdatenbanksysteme	32
Abbildung 4.4 - Entität User mit Beziehungstyp AUTHOR / LIKES	33
Abbildung 4.4 - Entität User mit Beziehungstyp FOLLOW.....	33
Abbildung 4.5 - Entität User	33
Abbildung 4.6 - Story Entität	34
Abbildung 4.7 - Verkettete Story Entität.....	34
Abbildung 4.8 - Section Entität.....	35
Abbildung 4.9 - Amazon S3 signierte URL.....	36
Abbildung 4.10 - Beispiel einer URL zum Erhalt eines Bildes mittels images.weserv.nl.....	37
Abbildung 4.11 - Gliederung der Anwendungsschicht	38
Abbildung 4.12 - Beispiel einer leicht zu verstehenden URI	39
Abbildung 4.13 - Beispiel einer URI, dessen Bedeutung nicht erkennbar ist.....	39
Abbildung 4.14 - pm2 Cluster	42
Abbildung 4.15 - Mockups ausgeloggter Nutzer	44
Abbildung 4.16 - Mockups ausgeloggter Nutzer	45
Tabelle 4.7 - Mögliche Gesten in der Storyview	48
Abbildung 4.21 - „Feed“- und „Entdecken“-Ansicht.....	50
Abbildung 4.22 - Klassendiagramm FoyerApi.....	52
Abbildung 5.1 - Konfigurationsdatei einer Ressource in der REST-Schicht	55

Abbildung 5.2 - Middleware zur Autorisierung eines Requests mittels user-token	55
Abbildung 5.3 - Exemplarische REST-Methode aus der Ressource ‚Stories‘	56
Abbildung 5.5 - Exemplarische Service-Methode aus der Ressource ‚User‘	58
Abbildung 5.5 - Exemplarische Persistenz-Methode aus der Entität ‚User‘ zum abonnieren eines Nutzers	58
Abbildung 5.7 - Erstellung der Indizes	60
Abbildung 5.8 - Abfrage der 20 neusten Stories ohne linked list	60
Abbildung 5.9 - Abfrage der 20 neusten Stories anhand einer linked list	60
Abbildung 5.10 - Abfrage der 20 neusten Stories anhand einer linked list	61
Abbildung 5.11 - Amazon AWS S3 Bucket Policy	61
Abbildung 5.12 - Singleton Entwurfsmuster in Swift	63
Abbildung 5.13 - Property-Observer	63
Abbildung 5.14- Methode aus der Klasse FoyerApi.....	64
Abbildung 5.15 - Router-Klasse innerhalb der Klasse FoyerApi	65
Abbildung 5.16 - Kernmethoden der Datenquelle.....	66
Abbildung 5.17 - Zelle der UICollectionView des FeedViewControllers.....	67
Abbildung 5.18 - Delegate-Methode des FeedViewControllers.....	68
Abbildung 5.19 - prepareLayout Methode des UICollectionViewLayouts	69
Abbildung 5.20 - Delegate-Methode referenceSizeForTitleInSection innerhalb der Collection View	70
Abbildung 5.21 - layoutAttributesForElementsInRect Methode des UICollectionViewLayouts.....	70
Abbildung 5.18 - Kennzeichnung einer zu übersetzenen Zeichenkette.....	71
Abbildung 5.23 - Fehler bei Aufruf von insertItemsAtIndexPaths an leerer Collection View	72
Abbildung 6.1 - Testen eines Objektes mit should.js	73
Abbildung 6.2 - Makefile	74
Abbildung 6.3 - Einfacher, exemplarischer Testfall	75
Abbildung 7.1 - Erstellung des API-Users.....	78
Abbildung 7.2 - Installieren von Abhängigkeiten	78
Abbildung 7.3 - Node.js Installation	78
Abbildung 7.4 - Zugriffsschlüssel einrichten	78
Abbildung 7.5 - Git-Repository klonen	79
Abbildung 7.6 - Prozessmanager pm2 Installation	79
Abbildung 7.7 - Starten des Prozessmanagers	79
Abbildung 7.8 - Firewall der Anwendungsschicht	79
Abbildung 7.9 - Konsolenbefehl zum bearbeiten der Datei rc.local	79
Abbildung 7.10 - Node.js-Server der vom Webhook aufgerufen wird	80
Abbildung 7.11 - Shellscript zur Aktualisierung des Repositories	80
Abbildung 7.12 - Erstellung des Datenbank-Nutzers	80
Abbildung 7.13 - Installation Neo4j und Abhängigkeiten.....	81
Abbildung 7.14 - Änderung Konfigurationsdatei Neo4j.....	81
Abbildung 7.15 - Firewall der Datenhaltungsschicht.....	81
Abbildung D.1 - Antwort auf die REST-Methode GET /stories.....	95
Abbildung D.2 - Klassendiagramm des iPhone-Prototypen.....	96
Abbildung D.3 - Navigationsbaum	97

Abbildung D.4 - Einrichtung eines Buckets in AWS	98
Abbildung D.5 - Einrichtung des Webhooks	98

iii. Tabellenverzeichnis

Tabelle 2.1 - HTTP-Methoden	4
Tabelle 2.2 -	6
Tabelle 3.1 - Muss-Anforderungen Schnittstelle	24
Tabelle 3.2 - Muss-Anforderungen iPhone-App	25
Tabelle 3.3 - Kann-Anforderungen Schnittstelle.....	25
Tabelle 3.4 - Kann-Anforderungen iPhone-App	26
Tabelle 4.1 - User Entität.....	34
Tabelle 4.2 - Story Entität.....	34
Tabelle 4.3 - Section Entität.....	35
Tabelle 4.4 - Item Entität	35
Tabelle 4.5 - Account Resource	40
Tabelle 4.6 - User Ressource.....	40
Tabelle 4.7 - Story Ressource	41
Tabelle 5.1 - Abhängigkeiten der Anwendungsschicht.....	54
Tabelle 5.2 - Projektstruktur des Xcode Projekts	62
Tabelle D.1 - Kenntlichmachung der Zusammenarbeit	99

1 Einleitung

1.1 Motivation und Problemstellung

Applikationen zur Fotografie erfreuen sich auf Smartphones einer großen Beliebtheit. Alleine durch die weit verbreitete App „Instagram“ werden täglich ca. 70 Millionen Bilder veröffentlicht, bei weltweit über 300 Millionen monatlich aktiven Nutzern.¹ Erweiterte Funktionalitäten wie Filter und Bildbearbeitung kombiniert mit Funktionen, die sich bereits in sozialen Netzwerken bewährt haben (u.a.: „social feed“, „kommentieren“, „teilen“, „gefällt mir“) verhelfen diesen Apps zu großem Erfolg und steigern erfolgreich deren Reichweite.

Als ich Anfang 2012 das soziale Wohn- und Einrichtungsnetzwerk *living.is* gründete, eine Webseite auf der Nutzer Ihre eigenen Vier-Wände präsentieren, habe ich den Einfluss von Smartphone-Apps auf Services, die von der Aktivität der Nutzer leben, wie Instagram oder eben *living.is*, unterschätzt. Denn jeder (100%) Nutzer ab 14 Jahren eines Smartphones in Deutschland macht mit seinem Gerät Fotos und neun von zehn Smartphone-Nutzern (92%) nutzen es auch für spontane Schnappschüsse.² Fotoportale wie *living.is*, deren Hauptinteresse es ist, es dem Nutzer so einfach wie möglich zu machen, Inhalte einzustellen, können von diesen Möglichkeiten enorm profitieren: ist ein Portal nur über den Webbrowser verfügbar, so ist der Aufwand und somit die Hemmschwelle für den Nutzer, Inhalte einzustellen, bedeutend größer. *living.is* hat sich nicht nur durch die technische, sondern durch die Umsetzung als reine Webseite als äußerst unflexibel herausgestellt, auch die Konzentration auf die Themen Wohnen und Einrichten schränkt den Service weiter ein.

Dies motivierte mich, einen schlankeren und flexibleren Service für den Austausch von Fotos zu erstellen. Einen Service, der sich nicht nur auf ein Foto oder ein Themengebiet beschränkt, sondern den Nutzern die Möglichkeit gibt, selbst ausgesuchte Bilder beliebig anzuordnen und zusätzlich versehen mit einigen Wörtern somit einen lebendigeren Einblick in das Erlebte ermöglicht. Die Zusammenstellung mehrerer Aufnahmen zu einem individuellen Album soll vom Nutzer mittels eines Editors eigenständig erstellt werden können. Der Anspruch an den Editor ist vor allem eine einfache Bedienbarkeit, wobei dem Nutzer ein möglichst großer Freiraum zur eigenen Gestaltung gegeben werden soll. Die Applikation soll jedoch nicht nur für die Veröffentlichung von privaten Erlebnissen in Frage kommen, sondern auch für Anwendungsfälle, die normalerweise über Blogsoftware bedient werden - wie z.B. Modeblogs, Kochrezepte, Tutorials.

1.2 Zielstellung

Ziel der Arbeit soll es sein, diesen Service als verteilte Anwendung zu entwerfen. Zum einen umfasst dies, die Konzeption und Umsetzung einer Applikation für das Apple iPhone, die es dem Nutzer erlaubt, Fotos und Text zu authentischen und visuell ansprechenden Beiträgen („Stories“) zu verbinden. Um die Applikation verwenden zu können, soll eine Anmeldung erforderlich sein. Jeder angemeldete Nutzer soll über ein Nutzerprofil verfügen und die Möglichkeit haben, Stories zu erstellen, andere Nutzer zu abonnieren („folgen“), Stories zu kommentieren, sie mit einer „gefällt mir“-Angabe zu versehen („liken“) sowie sie in anderen sozialen Netzwerken zu

¹ vgl. [INSTA]

² vgl. [BIT01]

teilen. Zum anderen gilt es ein verteiltes System zu entwickeln und umzusetzen, welche jene Funktionalitäten der Applikation über eine Schnittstelle bereitstellen soll, die einer zentralen Verarbeitung bedürfen. Dabei ist sicherzustellen, dass das System ein hohes Maß an Skalierbarkeit bietet.

Dabei

1.3 Vorgehensweise

Der Hauptteil der Arbeit unterteilt sich in sechs Abschnitte: Grundlagen, Analyse, Entwurf des Prototypen, Realisierung des Prototypen, Test und Deployment.

Zu Beginn sollen die erforderlichen theoretischen **Grundlagen** und verwendeten Technologien vorgestellt werden, um einen gemeinsamen Wissensstand für die weiteren Kapitel zu schaffen. In der darauffolgenden **Analyse** wird der Begriff der Story definiert und eine Anforderungsanalyse durchgeführt sowie mögliche Anwendungsfälle beschrieben. In der anschließenden **Entwicklung des Prototypen** werden die Anforderungen technisch analysiert und geeignete, die Implementierung vorbereitende Modelle entwickelt. Dies beinhaltet die Festlegung einer geeigneten Architektur, das Entwerfen einer Datenstruktur sowie die Skizzierung der iPhone-App anhand von Mockups. Daraufhin werden in der **Realisierung des Prototypen** die Schritte, Probleme und Lösungen der Umsetzung behandelt. Außerdem wird auf die eingesetzten Implementierungs-Technologien eingegangen. Im Kapitel **Test** werden die Maßnahmen erläutert, die unternommen wurden um eine korrekte Funktionalität zu gewährleisten sowie die verwendeten Bibliotheken und Werkzeuge vorgestellt und die Testinfrastruktur erläutert. Im anschließenden Kapitel **Deployment** werden schließlich die nötigen Schritte beschrieben um das Deployment der einzelnen Komponenten des verteilten Systems durchzuführen. Schlussendlich wird im **Fazit** ein Resümee gezogen und eine Betrachtung der Arbeitsergebnisse durchgeführt sowie überprüft, ob die gesteckten Ziele und Anforderungen erfüllt wurden. Ein abschließender Ausblick zeigt die Erweiterungsmöglichkeiten des Prototypen auf.

Bestimmte Teile der Arbeit sind in Zusammenarbeit mit Daniel Bardutzky (Matrikel-Nr.: xyz) entstanden, der in seiner Bachelorarbeit die android-spezifische Umsetzung der Applikation thematisiert. Überschneidungen werden in der im Anhang befindlichen Tabelle D.1 kenntlich gemacht.

2 Grundlagen

2.1 Webservice

Webservices dienen dazu Daten und Informationen bereitzustellen, die von Anwendungen genutzt werden können. Nach Schill und Springer³ ist ein Webservice ein herstellerübergreifender Dienst, welcher sich durch Plattform- und Programmiersprachenunabhängigkeit auszeichnet und auf Basis standardisierter Protokolle im Internet veröffentlicht wird. Üblicherweise wird dazu HTTP verwendet, möglich sind aber auch andere Webprotokolle, wie beispielsweise SMTP oder FTP.⁴ Im Folgenden werden die gängigen Verfahren zur Implementierung eines Webservices beschrieben.

2.1.1 SOAP

SOAP (Simple Object Access Protocol) ist ein Netzwerkprotokoll, welches den Austausch von Daten zwischen Systemen ermöglicht. Zur Repräsentation der Daten nutzt es XML (Extensible Markup Language) und zur Übertragung der Nachrichten Internet-Protokolle, üblicherweise HTTP. Durch die Repräsentation mittels XML lassen sich auch komplexe Objekte abbilden und übertragen. Laut Schill und Springer⁵ besteht eine SOAP Nachricht aus *Envelope*, einem optionalen *Header* und *Body*. Der *Envelope* stellt den Container der SOAP Nachricht dar, bei dem *Header* handelt es sich um zusätzliche Meta-Informationen und der *Body* enthält die eigentlichen Daten für den Empfänger. SOAP kann auf zwei verschiedene Arten genutzt werden. Entweder über RPC (*Remote Procedure Call*) oder durch dokumentenbasierten Nachrichtenaustausch. Bei *RPC* ruft die SOAP-fähige Anwendung entfernte Methoden des Servers per Web Services auf. Dies ermöglicht es einer Anwendung beliebiger Programmiersprache direkt Methoden anzusprechen und das Ergebnis per SOAP zu erhalten. Beim dokumentenbasierten Nachrichtenaustausch werden strukturierte Informationen an den Server gesendet, von ihm verarbeitet und sofort oder zeitlich versetzt zurück gesendet.⁶

2.1.2 REST

REST (Representational State Transfer) beschreibt viel mehr ein Programmierparadigma als die tatsächliche Umsetzung eines Webservices und wurde im Jahr 2000 durch Roy Thomas Fielding in seiner Dissertation entwickelt. Es stützt sich als Rahmenbedingung auf das Hypertext Transfer Protocol (HTTP) und legt folgende Bedingungen zu Grunde:⁷

1. Client-Server

Kapselung von Client und Server durch Anwendung des Separation-of-Concerns-Prinzips⁸, welches die Trennung von Zuständigkeiten vorschreibt. Dies ermöglicht die getrennte Entwicklung von Client und Server und verbessert die Skalierbarkeit des Servers bzw. der Client fällt schlanker aus.

³ vgl. [SCHAL01] S. 69

⁴ vgl. [SCHAL01] S. 69f

⁵ vgl. [SCHAL01] S. 73f

⁶ vgl. [W3C02]

⁷ vgl. [FIERO01] S. 76ff

⁸ vgl. [FIERO01] S. 78

2. Zustandslosigkeit

Jede Anfrage an den Server beinhaltet alle Informationen um sie zu erfüllen. Serverseitig wird kein Zustand verwaltet (z.B. durch eine Session). Dadurch sind RESTful Web Services einfach skalierbar.

3. Cachefähigkeit

Um die Netzwerklast minimieren zu können, sollen die Antworten des Servers explizit als cachefähig (zwischenspeicherbar) bzw. nicht-cachefähig markiert werden. Dies ermöglicht es dem Client zwischengespeicherte Daten zu nutzen anstatt eine neue Anfrage zu stellen.

4. Einheitliche Schnittstelle

Der Zugriff auf jede Ressource muss über einen einheitlichen Satz von Standardmethoden (Lesen, Schreiben, Erzeugen, Löschen) z.B. durch die HTTP-Methoden (vgl. Tabelle 2.1) möglich sein.

5. Layered System

Geschichteter, modularer Aufbau des Systems erlaubt klare Trennung von Zuständigkeiten und dient Wartung, Wiederverwendbarkeit und Erweiterbarkeit.

6. Code on Demand

Dynamische Erweiterung von Clientfunktionalität zur Laufzeit durch Herunterladen und Ausführen von Code in der Form von Applets und Skripten.

HTTP-Methode	Beschreibung
GET	Fordert die angegebene Ressource an
POST	Anlegen einer Ressource
PUT	Aktualisierung einer Ressource
DELETE	Löscht die Ressource
HEAD	Liefert nur den Antwortkopf einer GET-Anfrage auf die Resource
OPTIONS	Anforderung der vom Server für die angegebene Ressource bereitgestellten Methoden
PATCH	Teilweise Änderung einer Ressource

Tabelle 2.1 - HTTP-Methoden

Quelle: eigene Darstellung

2.2 JSON

JSON (JavaScript Object Notation) ist ein Programmiersprachen unabhängiges, kompaktes Datenaustauschformat in Textform und basiert auf dem Standard ECMA-262 3rd Edition - December 1999.⁹ In einem JSON-Dokument können folgende Typen vorkommen: Objekt, Array, Zahl, Zeichenkette und Boolean. Der Wert null kann jedem Typ zugewiesen werden (siehe Abb. 2.3).

Die Beschreibung eines Objektes wird durch eine öffnende geschweifte Klammer begonnen und durch eine schließende beendet, wobei darin beliebig viele Wertepaare (Name und Wert) enthalten sein können. Der Name ist immer als String dargestellt und wird durch einen Doppelpunkt von dem zugehörigen Wert getrennt. Arrays werden durch ein Paar aus eckigen Klammern beschrieben. In den Klammern können beliebig viele Werte enthalten sein. Zeichenketten werden von doppelten Anführungszeichen eingeschlossen. Unicode und Escape-

⁹ Vgl. [ECMA01]

Sequenzen können enthalten sein. Eine Zahl (siehe Abb. 2.1) kann durch ein negatives Vorzeichen eingeleitet werden, gefolgt von einer Zahl ohne führende Null oder nur einer Null. Die Einleitung der Dezimalstellen erfolgt durch einen Punkt und ist optional. Es kann ein Exponent durch e oder E eingeleitet werden und durch eine Zahl mit positiven, negativen oder gar keinem Vorzeichen beschrieben werden.

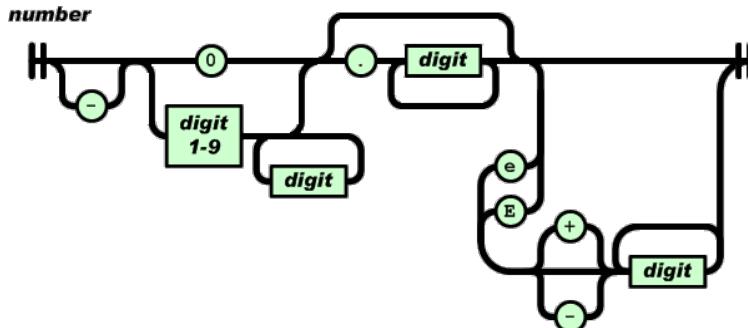


Abbildung 2.1 - Aufbau einer Zahl in JSON
Quelle: The JSON Data Interchange Format¹⁰

```

1   {
2     "key": "Zeichenkette",
3     "objekt": {
4       "objKey": "ObjWert"
5     },
6     "array": [
7       "Element 1",
8       "Element 2"
9     ],
10    "boolean": true,
11    "zahl": -3
12    "Dezimalzahl": -50.6E+33
13    "NULL-Wert" : null
14  }

```

Abbildung 2.3 - Beispielobjekt im JSON-Format
Quelle: eigene Darstellung

2.3 Node.js

Node.js ist eine Umgebung, die es ermöglichen soll, JavaScript außerhalb eines Webbrowsers und insbesondere als Webplattform zu nutzen. Node.js bedient sich statt eines Interpreters Googles V8-Engine zur just-in-time-Kompilierung von JavaScript-Code.

Es wird also kein Byte-Code erzeugt, sondern Maschinencode. Dieser wird zusätzlich zur Laufzeit optimiert. In JavaScript erfolgt der Memberzugriff in den meisten JavaScript-Laufzeitumgebungen mit Hilfe eines Dictionary-lookups. Dadurch wird die Speicheradresse des Klassenmembers ermittelt. Da dies bei jedem einzelnen Zugriff geschieht, ist der Memberzugriff langsamer als in anderen Programmiersprachen, wie Java oder Smalltalk. Denn dort wird für jeden Klassenmember ein Speicheroffset durch den Compiler festgelegt, was einen Lookup überflüssig macht. V8 nutzt statt eines Lookups sogenannte *hidden classes*.¹¹ In einer solchen hidden class werden die Offsetinformationen der Klasse gespeichert. Da in JavaScript während der Laufzeit die Klassenmember dynamisch hinzugefügt werden, wird bei jedem Hinzufügen eine neue hidden class mit der hinzugekommenen Offsetinformation erzeugt und ein Übergang zur aktualisierten hidden class gespeichert.

¹⁰ [ECMA02]

¹¹ vgl. [GOO01]

Neben dem Auslassen der sonst nötigen Lookups, bringt die Nutzung von hidden classes die Möglichkeit der weiteren Optimierung durch inline-caches (IC). Das sind optimierte und den Klassentypen angepasste Maschinencodefragmente, die zur Laufzeit den Maschinencode bei jeweiligen Operationen verbessern.¹²

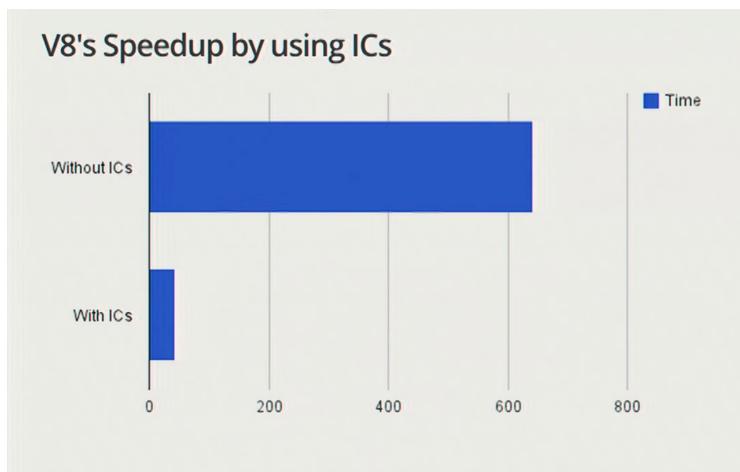


Abbildung 2.4 - Vergleich V8
Quelle: Google I/O 2012 - Breaking the JavaScript Speed Limit with V8

Da JavaScript eine eventgetriebene Sprache ist, baut Node.js auf dieser Grundlage auf und nutzt eine Architektur namens event loop. Im Gegensatz zu anderen Plattformen, wie Apache, welche für jede eingehende Verbindung einen neuen Thread startet¹³, bleibt mit der Nutzung von Node.js dieser Aufwand erspart. Jeder dieser gestarteten Threads – oder bei entsprechender Konfiguration auch Prozesse – arbeitet in der Regel mit synchronen Zugriffen auf CPU-Caches, RAM, Festplatten, oder das Netzwerk. Das führt dazu, dass der Thread eventuell den größten Teil seiner zugeschriebenen CPU-Zyklen damit verbringt auf das Medium zu warten. Die folgende Auflistung soll veranschaulichen, um welche Größenverhältnisse es sich bei diesen Latzenzen handelt.

Medium	Zugriffszeit in CPU-Zyklen
L1-Cache	3 Zyklen
L2-Cache	14 Zyklen
RAM	250 Zyklen
Festplatte	41.000.000 Zyklen
Netzwerk	240.000.000 Zyklen

Tabelle 2.2 -
Quelle: eigene Darstellung, basierend auf Ryan Dahl¹⁴

Durch die Nutzung von Callback-Methoden ist es möglich, die Wartezeit eines einzelnen Threads produktiv zu nutzen, statt ihn im Wartezustand zu halten. Dann kann der Thread andere Anweisungen ausführen und es ist nicht mehr nötig, mit jeder Verbindung einen einzelnen Thread zu eröffnen und zu betreiben. Node.js unterwirft jede Anwendung diesem Paradigma, indem alle langsamen Speicherzugriffe asynchron geschehen.

¹² vgl. [CLIDA01]

¹³ [APA01]

¹⁴ [DAHRY01]

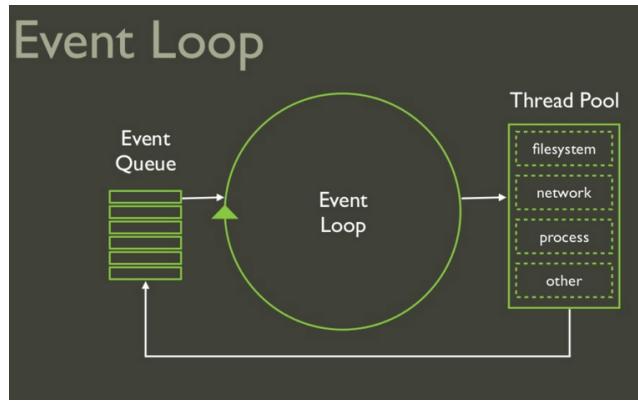


Abbildung 2.5 - Eventloop in Node.js

Quelle: Jeff Kunkle ¹⁵

Die Bearbeitung eingehender Anfragen durch die Anwendung hängt nun nicht mehr von der Erzeugung neuer Threads ab.¹⁶ Entsprechend sparsam ist auch der Speicherbedarf der Anwendung, da durch die Handhabung der Verbindungen in einem einzelnen Thread auch nur dieser eine Thread Stackspeicher reserviert bekommt. Besonders rechenintensive Anwendungen können in dieser Architektur allerdings dazu führen, dass die gesammelten Events und eingehende Anfragen mit extremen Latenzen rechnen müssen. Node.js findet auch bei sehr großen Unternehmen wie Microsoft, PayPal oder Yahoo Verwendung.¹⁷

2.3.1 NPM (Node Package Manager)

NPM ist ein Paketverwaltungssystem für Node.js, welches Softwarepakete aus der öffentlichen NPM-Registry zur Installation bereitstellt, ist ebenfalls vollständig in JavaScript geschrieben und läuft über die Node Plattform. Ein Paket besteht aus einem Verzeichnis mit einer oder mehreren Dateien, sowie Metainformationen. Üblicherweise sind diese Pakete klein und widmen sich gezielt einzelnen Aufgaben.¹⁸ Um beispielsweise das serverseitige Webapplikations Framework Koa innerhalb einer Applikation verwenden zu können, muss es lediglich mit dem Kommando „`npm install koa`“ installiert und anschließend in der Applikation eingebunden werden (siehe Abb. 2.6). Es wird dabei automatisch der Ordner `node_modules` im aktuellen Verzeichnis erstellt, in dem alle Module abgelegt werden.¹⁹

```

1 var koa = require('koa');
2 var app = koa();
3
4 app.use(function *() {
5   this.body = 'Hello World';
6 });
7
8 app.listen(3000);

```

Abbildung 2.6 - Koa Framework über NPM installieren

Quelle: eigene Darstellung

¹⁵ [KUNJE01]¹⁶ vgl. [HUGTO01] S. 3f¹⁷ vlg. [NODIN01]¹⁸ vgl. [NPM01]¹⁹ vgl. [SPRSE01] S. 136 ff.

Eine weitere Möglichkeit Module zu installieren ist es, die Datei package.json anzulegen in der sämtliche Informationen und Modulabhängigkeiten angegeben werden (siehe Abb. 2.7). Durch das Kommando „*npm install*“ werden alle Abhängigkeiten (*dependencies*) in den angegebenen Versionen installiert.

Der Paket Manager ist somit ein wichtiges Instrument bei der Entwicklung einer Node.js Applikation und ermöglicht es ähnlich wie Maven²⁰ bei Java komplexe Abhängigkeiten der Applikation zentral zu verwalten.

```

1  {
2    "name": "FoyerAPI",
3    "version": "0.0.1",
4    "repository": "mschonvogel/foyer-backend",
5    "description": "Public API for Foyer Apps",
6    "keywords": ["foyer", "api"],
7    "scripts": {
8      "test": "make test"
9    },
10   "dependencies": {
11     "koa-router": "3.7.0",
12     "koa": "0.13.0",
13   }
14 }
```

Abbildung 2.7 - Koa Framework über NPM installieren

Quelle: eigene Darstellung

2.3.2 EcmaScript 6

ECMAScript ist die standardisierte Bezeichnung für JavaScript, da es durch die ECMA standardisiert wurde. Es gibt drei bekannte Dialekte von ECMAScript: JavaScript von Oracle, JScript von Microsoft und ActionScript von Adobe. Grundlegend handelt es sich dabei um die gleiche Sprache, die in den jeweiligen Dialekten neue Verhaltensweisen hinzufügen. Der Name „JavaScript“ wird in der Praxis jedoch nicht immer eindeutig verwendet, so dass auch ein anderer Dialekt gemeint sein kann.²¹

Da ES6 ein neuer Standard ist, gibt es derzeit keine durchgehende Unterstützung durch die gängigen Plattformen, welche ECMAScript ausführen.²² Allerdings bietet Node.js bereits einige Features von ES6. So lassen sich beispielsweise seit der Node.js-Version 0.11 Generatorfunktionen aus ECMAScript 6 aktivieren, deren Verwendung in der REST-Schicht (vgl. 5.2.4) ein wichtiger Bestandteil ist.²³

2.4 Skalierbarkeit

Nach Henderson²⁴ wird ein System als skalierbar bezeichnet, wenn es in der Lage ist mit erhöhter Auslastung umzugehen, ohne dass die Performance negativ beeinflusst wird. Im Bereich der Webanwendungen erhält das Kriterium der Skalierbarkeit schnell eine verstärkte Bedeutung, sobald viele Nutzer gleichzeitig auf einen Dienst zugreifen. Henderson charakterisiert ein skalierbares System mit genau drei Punkten²⁵:

1. Das System kann sich erhöhter Nutzung anpassen
2. Das System kann sich an erhöhte Datenmengen (orig.: increased Dataset) anpassen

²⁰ <http://maven.apache.org>, abgerufen am 10.02.2015

²¹ vgl. [FIRMA01]

²² vgl. [KANG01]

²³ vgl. [DAILYJS01]

²⁴ vgl. [HENCA01] S. 202ff

²⁵ vgl. [HENCA01] S. 246

3. Das System ist wartbar

Beispielsweise gilt ein System mit 1000 Nutzern und 1 GB Datenvolumen nicht als skalierend, wenn es seine Antwortzeit nicht beim zehnfachen Nutzeraufkommen und zehnfachen Datenvolumen halten kann.²⁶ Die Leistung eines Systems kann auf zwei verschiedene Arten gesteigert werden:

A. Vertikale Skalierung

Vertikale Skalierung bezeichnet die Steigerung der Leistung eines Systems durch das Hinzufügen von Ressourcen (z.B. Speicherplatz, CPU). Ein an seine Kapazitätsgrenzen gestoßener Webserver kann beispielsweise durch einen leistungsfähigeren ersetzt werden. Dieser würde wiederum wieder ersetzt werden, falls die Kapazität erneut nicht ausreicht. Die Stärke dieser Art der Skalierung liegt im einfachen Softwaredesign, da die Software nicht auf die neue Hardware angepasst werden muss. Der klare Nachteil dieses Verfahrens ist, dass der Skalierbarkeit des Systems durch die Möglichkeiten der Hardware eine Obergrenze gesetzt ist.²⁷

B. Horizontale Skalierung

Das Konzept der horizontalen Skalierung basiert ebenfalls auf dem Hinzufügen weiterer Hardware. Eine Leistungssteigerung des Systems wird bei diesem Ansatz allerdings nicht über Modifikation bestehender Hardware, sondern durch die Ergänzung zusätzlicher Rechner erzielt. Im Zuge des Wachstums würde im Fall eines einzelnen Webservers dem System ein zweiter Server hinzugefügt werden. Die Effizienz dieser Vorgehensweise ist allerdings stark von der Software abhängig, da nicht jede Software in der Lage ist, mit der Hardware zu skalieren.²⁸

2.5 Graph Datenbanken

Ein Graph beschreibt eine mathematische Struktur, welche durch Kanten und Knoten gebildet wird, so dass eine verbundene Struktur entsteht. Eine Graphdatenbank nutzt diese Struktur, um stark vernetzte Informationen effizient zu speichern und abzufragen.

Die Heterogenität der Daten, das Datenvolumen und die Komplexität der Beziehungen zwischen den Daten sind in den letzten Jahren allesamt gestiegen.²⁹ Relationale Datenbanken unterliegen mit ihrem "Ein für alles"-Prinzip den spezialisierten Lösungen für bestimmte Anwendungsfälle. Diese Lösungen werden im Allgemeinen als NoSQL (Not only SQL) bezeichnet. Statt die zu speichernden Daten in Form einer Tabelle, in der jede Zeile ein Datensatz ist, zu präsentieren, nutzt eine Graphdatenbank ein Graphenmodell. Graphdatenbanken eignen sich besser als andere Datenbanksysteme zur Abbildung von sozialen Strukturen, da sich eine soziale Struktur (beispielsweise ein Soziogramm) direkt und ohne weiteren Aufwand als Graph betrachten lässt.

Laut Robinson ist ein Graphdatenbanken Managing System (GDBM) ein DBMS mit Create-, Read-, Update- und Deletemethoden (CRUD), welches ein graphenbasiertes Datenmodell bearbeitet.³⁰

²⁶ vgl. [HENCA01] S. 204

²⁷ vgl. [HENCA01] S. 248ff

²⁸ vgl. [HENCA01] S. 248ff

²⁹ vgl. [HUNMI01] S. 17f

³⁰ vgl. [ROBIA01] S. 5f

Weiterhin heißt es, dass sie im Allgemeinen dem OLTP-Paradigma (Online Transactional Processing) unterworfen sind, was bedeutet, dass sie sich auf die folgenden Anforderungen erfüllen: Verfügbarkeit, Datendurchsatz, Nebenläufigkeit und Wiederherstellbarkeit.³¹

Manche Graphdatenbanken nutzen einen sogenannten native graph storage, welcher auf die Speicherung und Verwaltung von Graphen spezialisiert ist. Das in den Graphdatenbanken populärste und meistgenutzte Graphenmodell ist der Property-Graph.

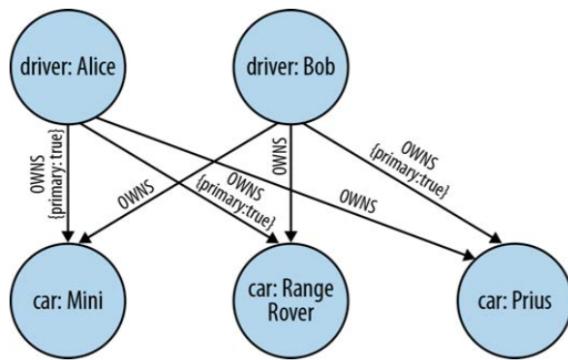


Abbildung 2.6 - Beispiel eines Property-Graphen
Quelle: Ian Robinson³²

Nach Robinson³³ hat ein Property-Graph die folgenden Eigenschaften:

- Er besteht aus Knoten, Beziehungen und Eigenschaften (Key-Value-Paare)
- Knoten enthalten Eigenschaften
- Beziehungen können auch Eigenschaften enthalten
- Beziehungen sind gerichtet und besitzen immer einen Start- und einen Endknoten

Beziehungen sind in diesem Modell gleichwertig zu Knoten anzusehen, da sie als eigene Entität in der Datenbank existieren und nicht erst zu Abfragezeiten generiert werden. Durch die Möglichkeit Eigenschaften aufzunehmen, lässt sich mit Ihnen die gleiche Ausdrucksstärke wie bei Knoten erzielen. Um bei Graphoperatoren den richtigen Pfad für eine Anfrage zu finden, müssen Beziehungen mit einem Typ versehen werden.

2.5.1 Neo4j

Neo4J ist eine quelloffene Graphdatenbank, welche in Java implementiert ist und seit 2007 von dem schwedischen Unternehmen NeoTechnology entwickelt wird.³⁴ Zur Abfrage und Manipulation von Daten aus Neo4j wird die Abfragesprache Cypher verwendet.

Transaktionen in Neo4j erfüllen die zur Verlässlichkeit nötigen ACID-Anforderungen:³⁵

- *atomicity*: Falls ein Teil der Transaktion fehlschlägt, bleibt der Zustand der Datenbank unverändert.
- *consistency*: Jede Transaktion hinterlässt die Datenbank in einem konsistenten Zustand.

³¹ vgl. [ORACLE01]

³² vgl. [ROBIA01] S.198

³³ vgl. [ROBIA01] S. 26

³⁴ vgl. [INFOQ01]

³⁵ vgl. [NEO4J01]

- *isolation*: Während einer Transaktion können die zu verändernden Daten nicht von anderen Operationen genutzt werden. Der Grad der Isolation ist einstellbar.
- *durability*: Das DBMS kann immer das Ergebnis einer erfolgten Transaktion wiedergeben.

Damit vereint Neo4j die Qualitäten aus dem klassischen relationalen Umfeld mit dem Datenmodell der Graphen für verbundene Strukturen.

Es gibt zwei Möglichkeiten Neo4j in eine Anwendung einzubinden. Einerseits die Verwendung als eingebettete Datenbank (in einer JVM³⁶-Umgebung), andererseits der Betrieb als Datenbankserver auf den über eine REST-API zugegriffen werden kann. Der eingebettete Modus ermöglicht den Betrieb der Datenbank im gleichen Prozess wie die zugreifende Anwendung, die Verwendung der Core-API (Framework für das Durchqueren des Graphen) sowie geringere Antwortzeiten, da die Datenbank nicht über das Netzwerk mit der Anwendung kommuniziert. Der Serverbetrieb erlaubt hingegen eine plattformunabhängige Nutzung sowie die verbesserte Möglichkeit der Skalierung, da die Datenbank unabhängig von der zugreifenden Anwendung ist.³⁷

2.5.1.1 Cypher

Die folgende Beschreibung der Sprache basiert auf der Online Dokumentation des Graphdatenbanksystems Neo4j.³⁸ Cypher ist eine deklarative Abfragesprache für Neo4j. Sie wird von NeoTechnology für Neo4J entwickelt und kommt nur dort zum Einsatz.³⁹

Die deklarative Programmierung verzichtet auf die Festlegung einzelner zur Problemlösung nötiger Schritte, wie in einem C++- oder Java-Programm, sondern orientiert sich an der Lösung zu einem Problem. Umgangssprachlich lässt es sich folgendermaßen formulieren: Man frage danach, was man als Ergebnis haben will, statt zu definieren, wie es durch Einzelschritte erreicht werden soll. Cypher ist in Scala implementiert und nutzt die funktionalen Eigenschaften der Sprache.⁴⁰

Abfragen in Cypher bestehen aus Klauseln, welche die Ergebnismenge einschränken.

Die MATCH-Klausel leitet die Beschreibung der zu suchenden Struktur ein. Die runden Klammern enthalten Knotenbezeichner und sind durch festgelegte Beziehungen verknüpft. Die geforderten Beziehungen werden in den Eckigen Klammern nach einem Doppelpunkt angegeben und ihre Richtung ist durch ein Größer- oder Kleinerzeichen (> und <), welches als Pfeil zu deuten ist, angegeben. Die Beispielabfrage liefert Bekanntschaften von Nutzern mit dem Namen ‘Michael’ zurück, die sich auch untereinander kennen.

Die RETURN-Klausel bestimmt, welche an die Bezeichner gebundenen Knoten in der Ergebnismenge liegen sollen. Durch die WHERE-Klausel lassen sich die durch MATCH gefundenen Strukturen noch genauer filtern.

1	MATCH (a {name: 'Michael'}) -[:KNOWS] -> (b) -[:KNOWS] -> (c), (a) -[:KNOWS] -> (c)
2	WHERE b.name = c.name
3	RETURN b, c

Abbildung 2.7 - Cypherabfrage Bekanntschaften von Nutzern
Quelle: eigene Darstellung

Die Ergebnismenge wurde durch die Bedingung eingeschränkt, dass die beiden gemeinsamen Bekanntschaften von a den gleichen Namen haben sollen.

³⁶ Java Virtual Machine, siehe <http://docs.oracle.com/javase/specs/jvms/se8/html/jvms-1.html>, abgerufen am 10.02.2015

³⁷ vgl. [HUNMI01] S. 24

³⁸ vgl. [NEO4J02]

³⁹ vgl. [ROBIA01] S. 27

⁴⁰ vgl. [HUNMI01] S. 42

Anonyme Knoten entstehen, wenn in den Runden Klammern kein Bezeichner angegeben werden. Im folgenden Beispiel werden die direkten Nachbarn anonym behandelt und können somit auch nicht in der Ergebnismenge angegeben werden.

```

1 MATCH (a {name: 'Michael'}) -[:KNOWS]-> () -[:KNOWS]-> (c)
2 RETURN c

```

Abbildung 2.8 - Cypherabfrage Bekanntschaften von Nutzern
Quelle: eigene Darstellung

2.6 Cloud Computing

Cloud Computing beschreibt den Ansatz abstrahierte IT-Ressourcen und Anwendungen auf Abruf und dynamisch mit den Hauptmerkmalen Bedarfsorientierung, Elastizität und Flexibilität über ein Netzwerk zu Verfügung zu stellen.⁴¹ In der Informatik wird das Bild der Wolke verwendet, um formlose Netze mit nur bedingt erkennbaren Strukturen darzustellen. Der Nutzer weiß in der Regel nicht mehr, wo genau sich seine Daten und Anwendungen befinden.

Die folgende Beschreibung basiert auf der Definition des National Institute of Standards and Technology (NIST) welche eine Roadmap zur Standardisierung erarbeitet hat.⁴²

Durch Cloudcomputing soll allgegenwärtiger und einfacher On-Demand-Zugriff auf einen gemeinsam genutzten Pool an IT- Ressourcen (z.B. Netzwerke, Server, Speicher, Anwendungen und Dienste) gewähren. Diese Ressourcen können mit minimalem Aufwand oder Interaktionen mit dem Provider bereitgestellt (orig.: "provisioned and released") werden. Das Modell besteht aus fünf essentiellen Eigenschaften (five essentials), drei Servicemodellen und vier Bereitstellungsmodellen (orig.: "deployment model").

2.6.1 Five essentials

1. **On-Demand Self-Service:** Die Provisionierung der Ressourcen (z. B. Rechenleistung, Storage) läuft automatisiert und ohne (menschliche) Interaktion mit dem Service Provider ab.
2. **Broad network access:** Die Dienste sind mit Standardzugriffsmethoden über das Netzwerk verfügbar und nicht an einen bestimmten Client gebunden.
3. **Resource Pooling:** Der Anbieter hält seine Ressourcen in einem Pool, auf den viele Anwender Zugriff haben (Multi-Tenant Modell). Die Nutzer wissen nicht, wo sich die genutzten Ressourcen örtlich befinden. Dies kann aber vertraglich festgesetzt werden (Region, Land oder Rechenzentrum).
4. **Rapid Elasticity:** Die Dienste können schnell, elastisch und eventuell automatisch bereitgestellt werden. Aus der Sicht des Nutzers erscheinen die genutzten Ressourcen unerschöpflich.
5. **Measured Services:** Die Nutzung kann gemessen und nachvollzogen werden und entsprechend bemessen auch den Cloud-Anwendern zur Verfügung gestellt werden. Die Kosten können durch charge-per-use oder pay-per-use geregelt werden.

⁴¹ vgl. [BAUCH01] S. 1f

⁴² vgl. [NIST01]

2.6.2 Cloud-Servicemodelle

- **Infrastructure as a Service (IaaS)** Bei IaaS wird IT-Infrastruktur (z. B. Rechenleistung, Datenspeicher oder Netze) als Dienst angeboten. Nutzer der Cloud kaufen diese standardisierten virtualisierten Dienste und können auf diesen Plattformen eigene Anwendungen oder Betriebssysteme laufen lassen. Der Nutzer verwaltet nicht die darunter liegende Cloud-Infrastruktur, sondern über das Betriebssystem, Speicher und laufende Anwendungen. Der Zugriff auf die Netzwerkkomponenten kann beschränkt sein (z.B. Hostfirewall).
- **Platform as a Service (PaaS)** Dem Nutzer steht die Möglichkeit bereit, eigene Programme, welche mittels Programmiersprachen, Bibliotheken, Dienste und Werkzeuge, die durch den Dienstleister bereitgestellt werden, entwickelt wurden, in die Cloud zu laden. Der Kunde hat keinen Zugriff auf das Betriebssystem oder die Hardware.
- **Software as a Service (SaaS)** Der Dienstleister gewährt dem Nutzer die Verwendung von Software, die in der Cloud betrieben wird. Die Software ist via Thinclient-Interface (z.B. Webbrowser) oder einer Programmoberfläche erreichbar. Der Kunde hat keinen Zugriff auf die darunterliegende Infrastruktur (Netzwerk, Server, Betriebssystem, Speicher). Ein bekanntes Beispiel ist Google Docs.

2.6.3 Deployment Models

- **private cloud** Die Cloud wird zur ausschließlichen Nutzung durch eine einzelne Organisation bereitgestellt. Besessen, verwaltet und betrieben werden kann sie durch diese Organisation oder durch dritte.
- **community cloud** Die Cloud wird einer bestimmten Interessengemeinschaft bereitgestellt. Die Cloud kann durch eine Organisation betrieben und besessen werden, welche Mitglied in der Gemeinschaft ist.
- **public cloud** Die Cloud wird der Öffentlichkeit bereitgestellt. Besessen, verwaltet und betrieben werden kann sie durch kommerzielle, akademische oder staatliche Organisationen, bzw. Kombinationen aus diesen. Sie liegt auf dem Gelände des Anbieters.
- **hybrid cloud** Mehrere Cloud-Infrastrukturen, welche eigenständig sind, können über standardisierte oder proprietäre Schnittstellen kommunizieren, so dass beispielsweise eine Lastverteilung ermöglicht wird.

2.7 Amazon Web Services

Mit Amazon Web Services (AWS) bietet das Unternehmen Amazon seit 2006 eine Zusammenstellung mehrerer Webservices als eine Infrastrukturplattform in der Cloud an und rechnet die Nutzung nach Verbrauch ab.⁴³ Damit war es das erste Unternehmen, das seine Infrastruktur als Dienstleistung angeboten hat. AWS wird von vielen großen Internetunternehmen, wie Soundcloud, Instagram oder Spotify als Umgebung eingesetzt.⁴⁴ Mittlerweile hat AWS seine Monopolstellung verloren und steht als Cloud-Dienstleister in Konkurrenz mit Google und Microsoft.⁴⁵ AWS bietet SDKs für verschiedene Plattformen, beispielsweise Android, iOS und Node.js.⁴⁶

⁴³ vgl. [AWS01]

⁴⁴ vgl. [AWS02]

⁴⁵ vgl. [GIGAOM01]

⁴⁶ vgl. [AWS03]

2.7.1 Simple Storage Service (S3)

Amazon S3 ist ein Webservice, der theoretisch beliebig große Datenmengen speichern kann. Sogenannte Buckets fungieren bei Amazon S3 zur Organisation der gespeicherten Objekte auf höchster Ebene. Das Konzept der Buckets und Objekte ähnelt dem der Verzeichnisse und Dateien. Buckets lassen sich via HTTP-Requests befüllen und löschen (siehe Abb. 2.9). Jedes Bucket und jedes Objekt innerhalb eines Buckets hat eine eigene Access Control List, mit der Lese- und Schreiboperationen den AWS-Nutzern gewährt werden können. Zudem besteht die Möglichkeit, den Zugriff auf den gesamten Inhalt des Buckets und den Bucket selbst mit einer Bucket-Policy zu verwalten.⁴⁷

```

1 DELETE /testbild.jpg HTTP/1.1
2 User-Agent: dotnet
3 Host: mybucket.s3.amazonaws.com
4 Date: Tue, 15 Jan 2008 21:20:27 +0000
5 x-amz-date: Tue, 15 Jan 2008 21:20:27 +0000
6 Authorization: AWS AKIAIOSFODNN7EXAMPLE:k3nL7gH3+PadhTEVn5EXAMPLE

```

Abbildung 2.9 - HTTP-Request zum Löschen einer Datei

Quelle: eigene Darstellung

2.7.2 Simple Email Service (SES)

Amazon SES ist ein hoch skalierbarer Service zum Versand von Massen-E-Mails und Transaktionsnachrichten und stellt eine effiziente Alternative zum Betrieb eines eigenen Mail-Servers dar. Der Versand kann über die AWS-Konsole, SDK, SMTP oder durch eine HTTPS-Anfrage erfolgen.⁴⁸ Der Empfang von E-Mails ist nicht möglich. Ferner bietet der Service zahlreiche Funktionen und Statistiken, um den E-Mail-Versand auszuwerten und zu kontrollieren. Da SMTP keine Authentifizierungsmethoden bereitstellt, empfiehlt es sich, den Server via SPF oder DKIM zu identifizieren, um eine erfolgreiche Zustellung zu gewährleisten.⁴⁹

2.8 Smartphone-App

Eine App ist ein Computerprogramm, das auf einem Smartphone, bzw. Tablet ausgeführt wird. Es gibt verschiedene Betriebssysteme für Smartphones und damit einhergehend auch eigene Entwicklungswerzeuge, mit denen Apps für das entsprechende Betriebssystem entwickelt werden können.

Apps können über Stores kommerziell vertrieben werden und einem breiten Nutzerfeld zugänglich zu machen. Bekanntestes Beispiel ist der App Store, der von Apple betrieben wird und historisch die erste Vertriebsplattform für mobile Apps war. Die drei größten Store-Betreiber sind Apple, Google und Microsoft.

2.8.1 Unterschied: Nativ-, Hybrid-, Web-App

Native-Apps werden gezielt für bestimmte mobile Betriebssysteme entwickelt. Dabei werden dafür vorgesehene Frameworks und Programmiersprachen verwendet, welche die Kapazitäten des Betriebssystems möglichst optimal ausnutzen, was der Performance der App zugute kommt. Die Portierung einer nativen App von einem Betriebssystem auf ein anderes kann aufgrund der unterschiedlichen Entwicklungsumgebungen

⁴⁷ vgl. [AWS08]

⁴⁸ vgl. [AWS04]

⁴⁹ vgl. [AWS05]

sehr kostspielig sein. Dafür wird dem Entwickler Zugriff auf die Hardware (Gyroskop, GPS, Bluetooth und weitere) erleichtert.

Hybrid-Apps sind eine Zwischenstufe zwischen Web- und Nativ-Anwendungen und stellen eine mögliche Lösung dar, eine Anwendung für mehrere Plattformen gleichzeitig zu entwickeln, indem man sich Webtechnologien (HTML und CSS) zunutze macht. Dies geschieht mit Hilfe von Cross-Development-Frameworks, beispielsweise PhoneGap, welche abstrahierte Schnittstellen für den Zugriff auf die plattformspezifischen Ressourcen bieten.

Web-Apps sind Webseiten, die in mobilen Webbrowsern dargestellt werden und für die Darstellung auf mobilen Geräten optimiert sind. Deshalb erfolgt der Vertrieb nicht über einen Store, wie bei nativen oder hybriden Apps, sondern durch das gewohnte Surfen mit dem Webbrowser. Die Implementierung einer Anwendung als Web-App ist allerdings keine Garantie für identische Darstellung auf allen Geräten, da es eine Vielzahl an mobilen Webbrowsern mit unterschiedlichen Stärken und Schwächen gibt.⁵⁰ Man ist auch in der Lage, mit Technologien wie jQuery Mobile, das Aussehen dem Betriebssystem anzugeleichen, hat jedoch keinen Zugriff auf hardwarenahe Funktionen.

2.9 iOS Software Development Kit

Das Software Development Kit für iOS ist eine Sammlung von zahlreichen Werkzeugen (Tools), Frameworks und Dokumentationen zum Entwickeln von Software für mobile Apple Produkte.

2.9.1 Xcode

Die Entwicklungsumgebung Xcode ist ein wesentlicher Bestandteil des Software Development Kits und nur für MAC OS X verfügbar. Xcode ist primär für die Programmiersprachen Swift und Objective-C unter Verwendung des Cocoa-Frameworks gedacht; die Programmiersprachen C und C++ können aber auch verwendet werden. Ab Xcode 6.0 ist es möglich, durch sogenannte *Playgrounds* bereits während der Eingabe in Xcode zu beurteilen, wie sich das Ergebnis verhält. Dies ermöglicht eine schnelle, iterative Entwicklung von Codestücken (*Rapid Prototyping*) ohne dass ein manuelles Kompilieren und Ausführen des Codes notwendig ist. Das Entwicklungspaket Xcode Tools besteht unter anderem aus diesen Programmen:⁵¹

- Der **Interface Builder** dient zur Erstellung der grafischen Oberflächen und stellt Verbindungen zwischen dieser und dem Code her.
- **Instruments** ist ein Werkzeug zum Analysieren von Applikationen bezüglich Speicherverbrauch, Dateizugriffe oder CPU-Auslastung.
- **Xcode Core** stellt die Basis von Xcode dar und ist eine Ansammlung von notwendigen Bibliotheken und Tools wie etwa die Compiler, Debugger, Kommandozeilenprogramme oder Versionsverwaltungen.
- **Application Loader** erleichtert es dem Entwickler Applikationen im App Store zu veröffentlichen.
- **iPhone Simulator** erlaubt das schnelle Testen der Applikation ohne ein Hardware-Gerät.

⁵⁰ vgl. [QUIRK01]

⁵¹ vgl. [WIKI02]

2.9.2 Frameworks

Die für die iPhone-Entwicklung benötigten Frameworks wurden von Apple anhand der Funktionalität in vier Schichten aufgeteilt und ermöglichen die Nutzung sämtlicher Soft- und Hardwarekomponenten.⁵²



Abbildung 2.10 - Organisation der iOS-Frameworks in vier Schichten

Quelle: eigene Darstellung

Die Schichten bilden eine hierarchische Struktur, in der die übergeordneten Schichten auf die Funktionalitäten der jeweils Vorangegangenen aufbauen. Die Framework Foundation aus der Core-Services-Schicht stellt beispielsweise sämtliche (Objective-C)-Basisklassen für die höher angesiedelten Schichten zur Verfügung.

Im Folgenden sollen die in Abbildung 2.10 dargestellten Schichten kurz beschrieben werden.

2.9.1 Core OS Layer

Die Core OS Schicht bildet die niedrigste Abstraktionsschicht und ist somit die Ebene, die der Hardware am nächsten ist. Sie stellt Funktionen auf niedrigster Ebene zur Verfügung, auf denen die meisten anderen Technologien aufbauen. Das Core Bluetooth Framework ermöglicht beispielsweise die Kommunikation mit Bluetooth Geräten und das System Framework bietet unter anderem Zugriff auf das Dateisystem und das Ansprechen von Netzwerkkomponenten über BSD Sockets. In dieser Ebene befindet sich unter anderem das Speicher- und Threadmanagement sowie die Interprozesskommunikation.⁵³

2.9.2 Core Services

Die Core Service Schicht stellt die Grundlage aller Dienste und Anwendungen, die auf dem Endgerät laufen, dar. Das bereits erwähnte Foundation-Framework zählt zu diesen Diensten. Des Weiteren beinhaltet diese Schicht weitere Frameworks, die Funktionen wie Lokalisierung, iCloud, Socialmedia-Anbindung sowie Netzwerkommunikation zur Verfügung stellen.⁵⁴

⁵² vgl. [APLDEV01]

⁵³ vgl. [APLDEV02]

⁵⁴ vgl. [APLDEV03]

2.9.3 Media

Die Media-Schicht bündelt Frameworks, die zur Implementierung und Verwendung von multimedialen Inhalten erforderlich sind. Das Core Graphics Framework rendert 2D Grafiken, das Framework OpenGL ES ermöglicht 3D-Darstellungen und das Framework Core Animation rendert Bewegungen und Animationen. Um Audio- bzw. Videodaten zu verarbeiten, bietet Apple unter anderem das Framework Core Audio bzw. AV Foundation an.⁵⁵

2.9.4 Cocoa Touch

Die Cocoa-Touch Schicht enthält die zentralen Bestandteile und Frameworks um ereignisgesteuerte Applikationen mit einer grafischen Benutzeroberfläche zu erstellen. Das UIKit-Framework stellt sämtliche, zur Anzeige und Eingabe benötigten grafischen Elemente bereit und unterstützt den Entwickler dabei, Applikationen zu erstellen, welche den Design Vorgaben von Apple entsprechen. Zahlreiche weitere High-Level-Frameworks sind in dieser Schicht gebündelt und ermöglichen beispielsweise die einfache Nutzung der Kartenfunktionalität oder des Adressbuchs.⁵⁶

2.10 Entwurfsmuster

Die, durch das SDK bereitgestellten Frameworks wenden überwiegend objektorientierte Konzepte an.⁵⁷ Selbst systemnahe Frameworks, wie beispielsweise das Core-Foundation-Framework simulieren anhand spezieller C-Funktionen Objektorientierung⁵⁸ und so ist die Adaptierung vieler Entwurfsmuster zum Großteil unvermeidbar und ratsam. In den folgenden Kapiteln werden ausgewählte, die im Laufe dieser Arbeit direkt oder indirekt durch die Verwendung der Frameworks genutzten Entwurfsmuster beschrieben.

2.10.1 Model-View-Controller

Das Entwurfsmuster Model-View-Controller ist bei der Entwicklung von Applikation für iOS von besonderer Bedeutung, denn es ist beim Einsatz des Frameworks UIKit unverzichtbar. Es dient unter anderem der Strukturierung und der Wiederverwendbarkeit und teilt die Präsentation, Verarbeitung und Speicherung von Daten in eine Ansicht (View), eine Programmsteuerungseinheit (Controller) und ein Datenmodell (Model) auf.

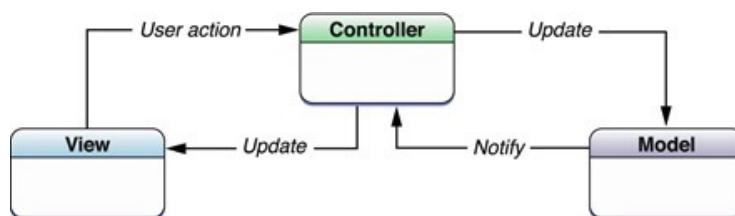


Abbildung 2.9 - Bei der iOS-Entwicklung zum Einsatz kommende Form des MVC-Patterns
Quelle: Apple⁵⁹

⁵⁵ vgl. [APLDEV04]

⁵⁶ vgl. [APLDEV05]

⁵⁷ vgl. [APLDEV06]

⁵⁸ vgl. [APLDEV07]

⁵⁹ vgl. [APLDEV06]

Die View ist hierbei für die Präsentation der Daten als auch die Entgegennahme von Benutzerinteraktionen zuständig. Der *Controller* agiert als sogenannter Vermittler (*Mediator*) und sorgt dafür, dass die Benutzerinteraktionen wirksam werden und delegiert zwischen dem Datenmodell und der zugehörigen View. Da das Pattern *Model-View-Controller* mehrere Entwurfsmuster in sich vereint, zählt es zu den sogenannten *Compound-Patterns*.

Hinsichtlich der Entwicklung einer Applikation für iOS lässt sich das Entwurfsmuster zusammenfassend wie folgt beschreiben: Unter Verwendung des Interface Builders, ein in Xcode integriertes Tool mit dem man grafische Benutzeroberflächen erstellen kann, wird zunächst eine View entworfen. Ein Persistenz-Manager stellt die zugrunde liegenden Daten über entsprechende Schnittstellen zur Verfügung. Im Mittelpunkt steht eine Controllerklasse welche zwischen den bereitgestellten Daten und der Interaktion des Nutzers über die View delegiert.

Die Form des MVC-Patterns, die bei der iOS-Entwicklung zur Anwendung kommt, unterscheidet sich in Hinsicht auf die Verteilung von Zuständigkeiten zu der in den Lehrbüchern dargestellten Form, denn sie lässt keine direkte Kommunikation zwischen Datenmodell und View zu. Dies steigert die Wiederverwendbarkeit beider Komponenten, da sowohl Model als auch View unabhängig voneinander nutzbar sind.

Allerdings besteht auch die Möglichkeit, dass ein Objekt mehrere Rollen aus dem MVC-Pattern in sich vereint. Beispielsweise kann der Controller zusätzlich die Rolle der View übernehmen, in diesem Fall spricht man von einem ViewController.⁶⁰

2.10.2 Observer

Nach Eilebrecht, Karl und Starke, Gernot⁶¹ ermöglicht das Observer-Muster einem oder mehreren Objekten, automatisch auf die Zustandsänderung eines bestimmten Objekts zu reagieren, um den eigenen Zustand anzupassen. Über *Notifications* wird im Cocoa-Framework ein Observer-Mechanismus implementiert. Eine Instanz fügt sich in eine Observer-Liste (über die Klasse NSNotificationCenter) ein. Das Objekt, das benachrichtigen möchte erzeugt eine Instanz von NSNotification und verteilt diese über das NSNotificationCenter. Ein weiterer Mechanismus in dem dieses Entwurfsmuster Anwendung findet, ist das Kay-Value-Observing, in dem eine Benachrichtigung erfolgt, wenn bestimmte Eigenschaften des Objekts geändert werden. Im Gegensatz zu Notifications wird hier keine zentrale Stelle genutzt, sondern die Ereignisse werden direkt am Beobachter übermittelt.⁶²

2.10.3 Delegation

Das Delegation-Pattern stellt eine Adaption des Decorator-Patterns dar und ermöglicht es, Klassenfunktionalitäten ohne Vererbung und ohne Änderung des Codes zu erweitern. Unter Delegation versteht man, dass ein Objekt eine Referenz auf ein anderes Objekt besitzt und in bestimmten Situationen Methoden dieses Objektes aufruft. Das heißt, das sogenannte Host-Objekt delegiert eine Aufgabe an das eingeschlossene Objekt (den Delegate). Viele Klassen innerhalb von Cocoa verwenden Delegates, z.B. UIApplication oder UICollectionView.

⁶⁰ vgl. [STAMA01] S. 69

⁶¹ vgl. [EILKA01] S. 61

⁶² vgl. [SADER01] S. 89

2.10.4 Singleton

Durch die Anwendung des Singleton Entwurfsmusters kann sichergestellt werden, dass von einer Klasse genau ein Objekt existiert und dies über einen zentralen Zugriffspunkt bereitgestellt wird.⁶³ Das Pattern wird von einigen Cocoa-Klassen implementiert, z.B. die Klasse UIApplication aus UIKit.

2.11 ViewController

Wie bereits im Kapitel 2.10.1 erwähnt, existiert auch das Prinzip des Viewcontrollers, also ein Controller, welcher auch View-Aufgaben übernimmt. Eine größere Applikation wird oft in mehrere Masken (auch *Screens* genannt) aufgeteilt, welche wiederum aus einem Satz von View-Objekten zur Darstellung der Daten bestehen. Die von einer Maske beinhalteten Views werden von einem Viewcontroller gehalten, in dessen Verantwortung es liegt, Daten für die einzelnen Views bereitzustellen. Die Basisklasse für alle Viewcontroller in UIKit ist die Klasse UIViewController. Viewcontroller können grundsätzlich in folgende zwei Kategorien aufgeteilt werden:⁶⁴

2.11.1 Contentviewcontroller

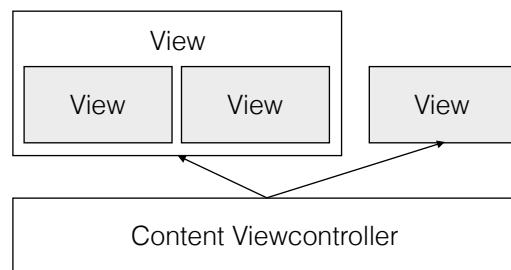


Abbildung 2.10 - Schematische Darstellung eines Contentviewcontroller
Quelle: eigene Darstellung

Hauptaufgabe des Contentviewcontrollers ist die Darstellung von Oberflächenelementen mit Hilfe von Views, welche auch geschachtelt sein können (siehe Abb. 2.10).

2.11.1.1 UICollectionViewController

⁶³ vgl. [GAMER01] S. 157

⁶⁴ vgl [APLDEV16]

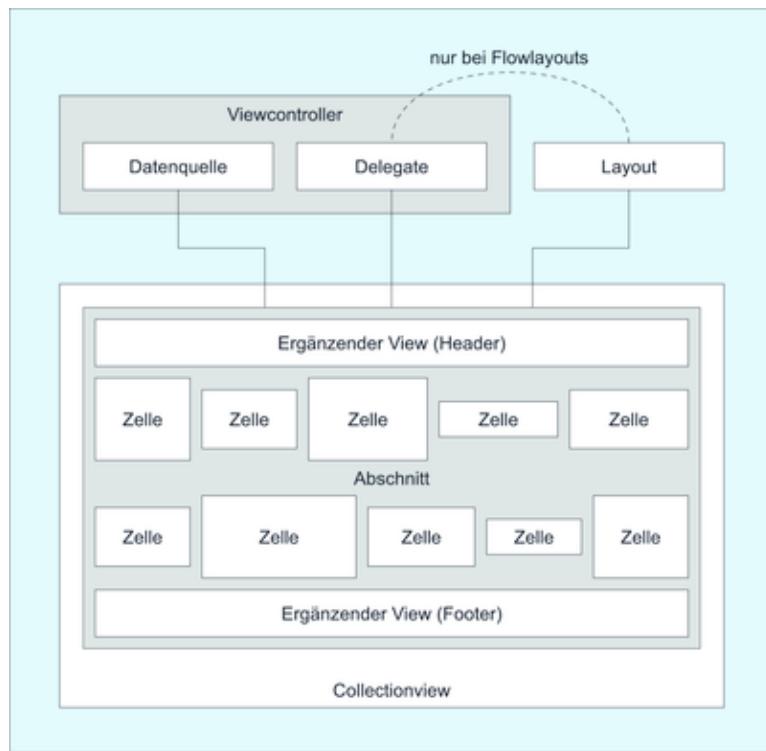


Abbildung 2.11 - Collection Viewcontroller
Quelle: Rodewig, Klaus M. und Wagner, Clemens⁶⁵

Der UICollectionViewController kapselt eine UICollectionView in einem UIViewController und implementiert die Protokolle UICollectionViewDelegate und UICollectionViewDataSource.

Die Klasse UICollectionView bietet die Möglichkeit, simple Raster umzusetzen und stellt eine, unter der Verwendung eines Layouts geordnete Ansammlung von Items dar, welche in Sektionen gefasst wird.

Um Inhalte darzustellen muss die UIViewController die Datenschnittstelle (UICollectionViewDataSource) implementieren, die zwischen dem Datenmodell der Applikation und der UICollectionView vermittelt. Diese Schnittstelle liefert der UICollectionView benötigte Informationen wie die Anzahl der Einträge im Datenmodell oder welche Views bei der Darstellung dieser zu verwenden sind.

Um auf Aspekte bezüglich des Layouts oder der Interaktion des Nutzers mit den einzelnen Zellen Einfluss zu nehmen, dient die optional zu implementierende Schnittstelle UICollectionViewDelegate.

Neben Zellen (UICollectionViewCell) kann eine Collection View zwei weitere Arten von Views beinhalten. Zum einen ergänzende Views (Supplementary View), welche Informationen über eine Sektion anzeigen (z.B. Header / Footer), zum anderen dekorative Views (Decoration View), die rein dekorative Aufgaben (z.B. Hintergrund einer Sektion) übernehmen.

Um die Effizienz der Collection View zu erhöhen, werden Views, die sich durch Scrollen außerhalb des sichtbaren Bereichs befinden, wiederverwendet anstatt gelöscht. Um dieses View-Recycling zu ermöglichen, müssen alle in einer Collection View dargestellten Views von der Klasse UICollectionViewCellReusableView erben.⁶⁶

2.11.1.2 UITableViewController

Der UITableViewController kapselt eine UITableView in einem UIViewController und implementiert die Protokolle UITableViewDelegate und UITableViewDataSource. Das Konzept der UITableView entspricht dem der zuvor

⁶⁵ vgl. [RODKL01]

⁶⁶ vgl. [APLDEV08]

beschriebenen UICollectionView mit der Einschränkung, dass sie für ein Ein-Spalten-Layout konzipiert ist und keine Möglichkeit besteht, mittels eines Layouts Einfluss auf dieses zu nehmen.

2.11.2 ContainerviewController

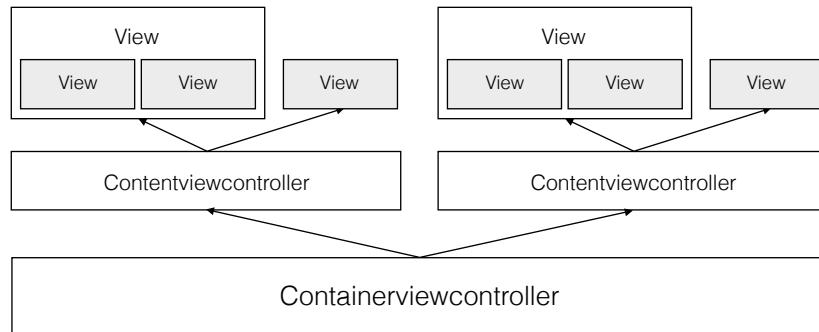


Abbildung 2.12 - ContainerviewController

Quelle: eigene Darstellung

Der ContainerviewController zeigt einen oder mehrere andere ViewController an und verwaltet diese – es entsteht eine ViewController-Hierarchie (siehe Abb. 2.12). Des Weiteren organisiert er den Wechsel zwischen diesen Controllern.

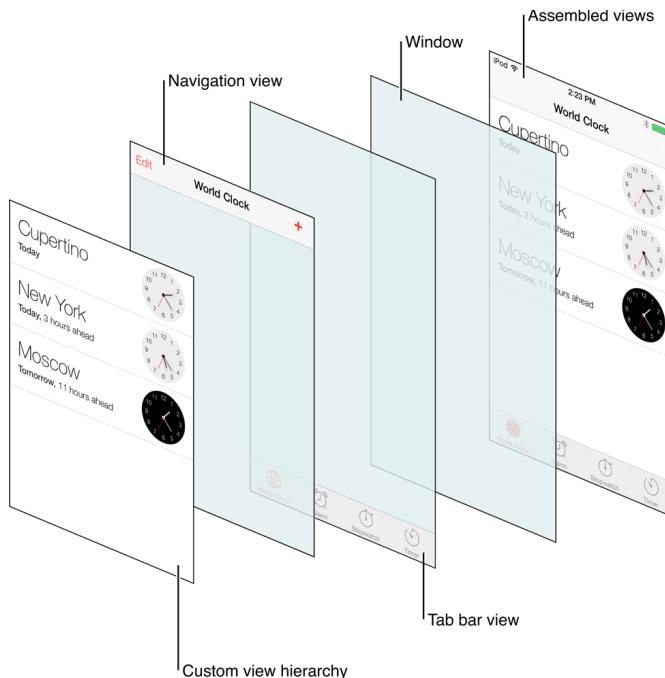


Abbildung 2.13 - View Hierarchie von Container Viewcontrollern

Quelle: Apple⁶⁷

2.11.2.1 UINavigationController

Der UINavigationController ist ein UIViewController, dessen Hauptaufgabe die Verwaltung einer Hierarchie von mehreren Contentviewcontrollern ist, welche er in Form eines internen Stapels (Stack) verwaltet. ViewController

⁶⁷ [APLDEV09]

können oben auf den Stapel gelegt (*push*) oder von der Stapspitze weggenommen werden (*pop*). Oberhalb des Content-Bereichs wird eine Navigationsbar (vgl. Abb. 2.13 “Navigation view”) dargestellt, die ebenfalls durch den UINavigationController verwaltet wird. Sie enthält auf der linken Seite einen Button zum Wechsel in die vorherige Hierarchieebene (*pop*), also zu dem Viewcontroller, der sich unter dem aktuellen Viewcontroller im Stapel befindet. Der typische Anwendungsfall ist der *Drill-Down* in eine tiefer gelegene Detail- bzw. Hierarchieebene („vom Allgemeinen zum Speziellen“).⁶⁸

2.11.2.2 UITabBarController

Der UITabBarController teilt die Applikation in mehrere Bereiche auf, indem dieser mehrere Viewcontroller verwaltet und diese über Reiter am unteren Bildschirmrand in der sogenannten Tabbar zugänglich macht (vgl. Abb. 2.13 “Tab bar view”). Diese Controller werden im Gegensatz zum UINavigationController nicht in Form eines Stapels verwaltet, sondern nebeneinander in Form einer Liste und zeigen nicht einfach detailliertere sondern gänzliche andere Informationen an. Jede View ist durch die Betätigung des entsprechenden Reiters sofort verfügbar. In der Tabbar können bis zu fünf Reiter nebeneinander angezeigt werden. Bei sechs und mehr Reitern verwendet der UITabBarController einen zusätzlichen UINavigationController mit eingebettetem UITableView-Controller, welcher den Zugriff auf die restlichen Viewcontroller erlaubt.⁶⁹

⁶⁸ [APLDEV09]

⁶⁹ vgl. [SILTH01] S.96

3 Analyse

In diesem Kapitel werden die Anforderungen an den Entwurf und die prototypische Entwicklung näher erläutert. Kern der Applikation stellt die so genannte „Story“ dar, die es einführend zu definieren gilt.

3.1 Definition “Story”

Eine Story ist eine von einem Nutzer in einem kreativen Prozess erstellte Kombination von Multimedialen Elementen und Text. Als multimediale Elemente sind sowohl Fotos als auch Videos oder Landkarten denkbar, im Rahmen dieser Arbeit beschränkt sich die Definition einer Story allerdings auf die Kombination von Fotos und Text. Eingeleitet wird jede Story mit einem Coverbild und einer Überschrift, gefolgt von einer vertikalen Aneinanderreichung von (beliebig vielen) Sektionen. Sektionen setzen sich aus optionaler Überschrift und optionalem Fließtext und einem oder mehreren Items (Fotos, Videos etc.) zusammen.

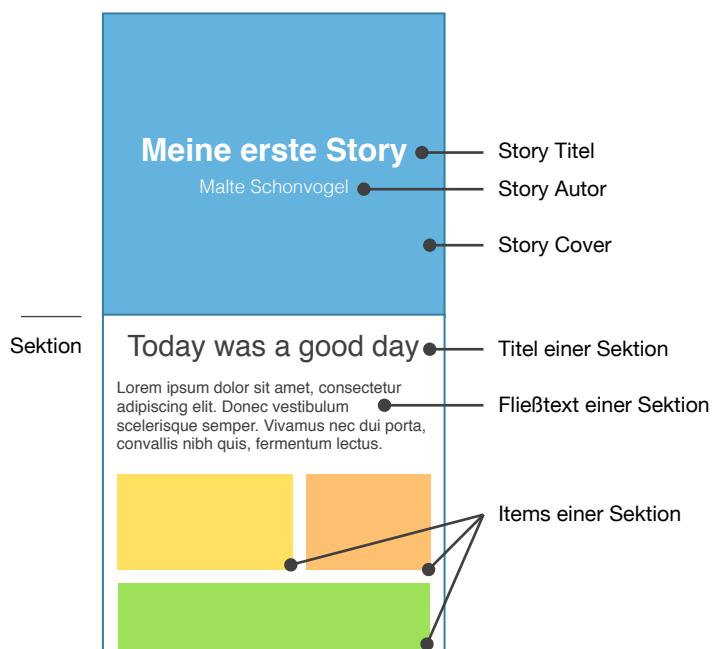


Abbildung 3.1 - Schematische Darstellung einer Story
Quelle: eigene Darstellung

3.2 Funktionale Anforderungen

3.2.1 Muss-Anforderungen

Bei den Muss-Anforderungen der Schnittstelle wird das Augenmerk auf die Erreichung eines minimum viable products (mvp) gelegt, was übersetzt soviel bedeutet wie: „Produkt mit den minimalen Anforderungen und Eigenschaften“. Ziel bei diesem Paradigma ist es, nicht ein Produkt zu entwickeln und perfektionieren von dem man denkt, dass es potentielle Nutzer mögen, sondern so schnell wie möglich ein auslieferbares Produkt zu

erstellen, um es dann mit Hilfe von stetigem Nutzerfeedback weiterzuentwickeln.⁷⁰ Diese Vorgehensweise führt zu geringeren Entwicklungskosten und verwendet Konzepte der agilen Softwareentwicklung, um den Wunsch der Endnutzer genauer zu treffen.

Bei der folgenden Tabelle handelt es sich um eine Auflistung der Muss-Anforderungen der prototypisch zu implementierenden Schnittstelle (REST-API). Sollte eine Muss-Anforderung von einer anderen abhängen, wird diese Abhängigkeit in der Spalte „Abh.“ referenziert.

Nr.	Abh.	Muss-Anforderung
1		Die Middleware muss als ein Webservice in der Programmiersprache Javascript entworfen werden.
2		Über die Schnittstelle muss es möglich sein, ein Nutzerprofil anzulegen (“registrieren”). Als Pflichtangaben sind “E-Mail-Adresse”, “Passwort”, “Username”, “Vorname”, “Nachname” und “Geschlecht” zu realisieren
3	2	Über die Schnittstelle muss es möglich sein, sich mit E-Mail-Adresse und Passwort zu authentifizieren. Die Schnittstelle muss einen Token zurückliefern, mit dem der Nutzer autorisiert ist, Funktionen der Schnittstelle zu nutzen, für die man eingeloggt sein muss.
4	3	Über die Schnittstelle muss es möglich sein, ein Nutzerprofil abzurufen.
5	3	Über die Schnittstelle muss es möglich sein, das eigene Nutzerprofil zu bearbeiten.
6	3	Über die Schnittstelle muss es möglich sein, eine Story zu speichern.
7	3	Über die Schnittstelle muss es möglich sein, eine Story zu bearbeiten.
8	3	Über die Schnittstelle muss es möglich sein, eine Story anhand eines eindeutigen Identifikators abzurufen.
9	3	Über die Schnittstelle muss es möglich sein, mehrere, chronologisch nach Erscheinungsdatum sortierte, Stories abzurufen.

Tabelle 3.1 - Muss-Anforderungen Schnittstelle

Quelle: eigene Darstellung

Bei der folgenden Tabelle handelt es sich um eine Auflistung der Muss-Anforderungen der prototypisch zu implementierenden iPhone Client Applikation (App). Wie in der vorangehenden Tabelle, werden Abhängigkeiten zwischen den Anforderungen mit der Spalte „Abh.“ verdeutlicht.

Nr.	Abh.	Muss-Anforderung
1		Die App muss es dem Nutzer ermöglichen, sich bei erstmaliger Benutzung zu registrieren, also über die Schnittstelle ein neues Nutzerprofil anzulegen.
2	1	Die App muss es dem Nutzer ermöglichen, sich gegenüber der Schnittstelle mit E-Mail-Adresse und Passwort zu authentifizieren.
3	2	Der App muss es möglich sein, die neusten Stories in einer Übersicht anzuzeigen.
4	2	Der App muss es möglich sein, eine Story anzuzeigen.
5	2	Die App muss es dem Nutzer ermöglichen, eine neue Story zu erstellen.
6	2	Die App muss es dem Nutzer ermöglichen, eine von ihm erstellte Story zu bearbeiten.
7	2	Der App muss es möglich sein, Benutzerprofile anzuzeigen.

⁷⁰ vgl. [LEAN01]

Nr.	Abh.	Muss-Anforderung
8	2	Die App muss es dem Nutzer ermöglichen, sein Benutzerprofil zu bearbeiten.

Tabelle 3.2 - Muss-Anforderungen iPhone-App

Quelle: eigene Darstellung

3.2.2 Kann-Anforderungen

Die hier aufgeführten Punkte werden als zusätzliche Anforderungen definiert und sind nach erfolgreicher Umsetzung der Muss-Anforderungen zu realisieren.

Die Tabelle 3.3 listet die Kann-Anforderungen der prototypisch zu implementierenden Schnittstelle auf. Sortiert ist die Tabelle nach Priorität. Ist dabei eine Anforderung von einer anderen abhängig, so wird dies durch die Spalte „Abh.“ erkenntlich.

Nr.	Abh.	Kann-Anforderung
1		Über die Schnittstelle muss es möglich sein, eine Story mit einer "gefällt mir" Angabe zu versehen.
2	1	Über die Schnittstelle muss es möglich sein, die "gefällt mir" Angabe einer Story zu entfernen.
3		Über die Schnittstelle muss es möglich sein, als Nutzer einem anderen Nutzer zu "folgen".
4	3	Über die Schnittstelle muss es möglich sein, als Nutzer einen anderen Nutzer zu "entfolgen".
5		Über die Schnittstelle muss es möglich sein, als Nutzer das eigene Passwort zu ändern.
6		Über die Schnittstelle muss es möglich sein, als Nutzer ein Avatar (Profilfoto) hochzuladen.
7	6	Über die Schnittstelle muss es möglich sein, als Nutzer das eigene Avatar zu löschen.
8		Über die Schnittstelle muss es möglich sein, die Nutzer auszulesen, die einem anderen Nutzer folgen.
9		Über die Schnittstelle muss es möglich sein, die Nutzer auszulesen, denen ein Nutzer folgt.
10		Über die Schnittstelle muss es möglich sein, als Nutzer ein Newsfeed abzurufen.
11		Über die Schnittstelle muss es möglich sein, sich über das soziale Netzwerk Facebook einzuloggen/zu registrieren.
12		Über die Schnittstelle muss es möglich sein, als Nutzer Stories zu kommentieren.

Tabelle 3.3 - Kann-Anforderungen Schnittstelle

Quelle: eigene Darstellung

Bei der folgenden Tabelle handelt es sich um eine Auflistung der Kann-Anforderungen der prototypisch zu implementierenden iPhone-App.

Nr.	Abh.	Kann-Anforderung
1		Die App muss es dem Nutzer ermöglichen, eine Story mit einer "gefällt mir" Angabe zu versehen bzw. diese wieder zurückzunehmen.
2		Die App muss es dem Nutzer ermöglichen, einen anderen Nutzern zu abonnieren bzw. dieses Abonnement wieder aufzulösen.
3		Die App muss es dem Nutzer ermöglichen, sein Passwort zu ändern.

Nr.	Abh.	Kann-Anforderung
4		Die App muss es dem Nutzer ermöglichen, ein Avatar hochzuladen bzw. dieses wieder zu entfernen.
5		Die App muss es dem Nutzer ermöglichen, die Nutzer anzuzeigen, die einem anderen Nutzer folgen.
6		Die App muss es dem Nutzer ermöglichen, die Nutzer anzuzeigen, denen ein Nutzer folgt.
7		Nach Auswahl der zur Story hinzuzufügenden Bilder, muss es die App dem Nutzer ermöglichen, die Bilder zu bearbeiten. Es soll dem Nutzer möglich sein, den Ausschnitt eines Fotos zu bearbeiten ("zuschneiden") und Bildfilter (bspw. schwarz/weiß, sepia) anzuwenden.
8		Die App muss es dem Nutzer ermöglichen, sein Newsfeed aufzurufen.
9		Die App muss es dem Nutzer ermöglichen, sich über das soziale Netzwerk Facebook einzuloggen/zu registrieren.
10		Die App muss es dem Nutzer ermöglichen, Stories zu kommentieren.

Tabelle 3.4 - Kann-Anforderungen iPhone-App

Quelle: eigene Darstellung

3.3 Nichtfunktionale Anforderungen

3.3.1 Zuverlässigkeit

Die Applikation soll etwaiges Fehlverhalten (z.B. fehlerhafte Formulareingaben) seitens des Benutzers tolerieren und keine Abstürze verursachen. Da die Zuverlässigkeit der App jedoch unter anderem über die Verbindung beeinflusst wird, soll dem Nutzer bei Verbindungsabbrüchen die Möglichkeit geboten werden, einen Schnittstellenaufruf wiederholen zu können.

3.3.2 Handhabung/Look and Feel

Die Benutzeroberfläche der Anwendung soll die Human Interface Guidelines⁷¹ von Apple beachten und umsetzen. Des Weiteren soll sie so gestaltet sein, dass diese sowohl mit Geräten, die eine niedrige Auflösung besitzen als auch mit Geräten hoher Auflösung gleichermaßen bedienbar ist. Ferner sollen wiederkehrende grafische Komponenten für eine einheitliche Oberfläche mit klarer Struktur und einer konsistenten Bedienbarkeit sorgen.

3.3.3 Leistung und Effizienz

Bezüglich der Leistung und Effizienz ist in erster Linie die Latenz und Übertragungsgröße entscheidend. Aufgrund der Tatsache, dass die Internetverbindung von mobilen Geräten teilweise sehr eingeschränkt sein kann, müssen Anfragen an Server, sowie Antworten ein möglichst geringes Datenvolumen aufweisen. Um eine Leistungsverbesserung zu erzielen, sollten Daten, die sich selten ändern, während einer Sitzung nur einmalig abgerufen werden. Um eine flüssige und stetige Bedienbarkeit in der Applikation zu gewährleisten, sind performancelastige Aufgaben und Netzwerkanfragen stets asynchron zu bearbeiten.

⁷¹ vgl. [APLDEV10]

3.3.4 Skalierbarkeit

Der Webservice soll so konzipiert werden, dass es mit wenig Aufwand möglich ist zusätzliche Ressourcen hinzuzufügen, um so auf einen möglichen Nutzerzuwachs zu reagieren.

3.3.5 Wartbarkeit

Die Applikation soll so konzipiert und realisiert werden, dass Änderungen einfach durchgeführt werden können. Um die Wartbarkeit zu erhöhen, sind die Standard-Interfacekomponenten gegenüber Eigenentwicklungen zu bevorzugen.

3.3.6 Sicherheit

Die Applikation soll keine systemkritischen Daten speichern, die vom Anwender missbraucht werden könnten. Um die Anmeldung an der Schnittstelle zu beschleunigen, ist von der Schnittstelle bei erstmaliger Anmeldung ein Token zurückzuliefern, den die Applikation bei zukünftigen Anfragen zwecks Authorisierung an die Schnittstelle mitsendet. Sollten Anfragen bei Dritt-Diensten direkt aus der Applikation (z.B. Bilderupload) Zugangsdaten erfordern, so sollte dies über eine signierte URL⁷² umgesetzt werden. Des Weiteren sollte der Datenaustausch mit dem Server ausschließlich über eine SSL gesicherte Verbindung stattfinden. Alle Zugriffe auf persönliche Daten des Benutzers sollen nur mit Login möglich sein.

3.3.7 Distribution

Nachdem die Applikation die Testphase erfolgreich absolviert hat, soll die Anwendung im AppStore veröffentlicht werden. Folglich muss die Applikation die AppStore Review Guidelines⁷³ einhalten.

⁷² URL die eine Signatur mit Authentifizierungsinformationen enthält.

⁷³ vgl. [APLDEV11]

3.4 Anwendungsfälle

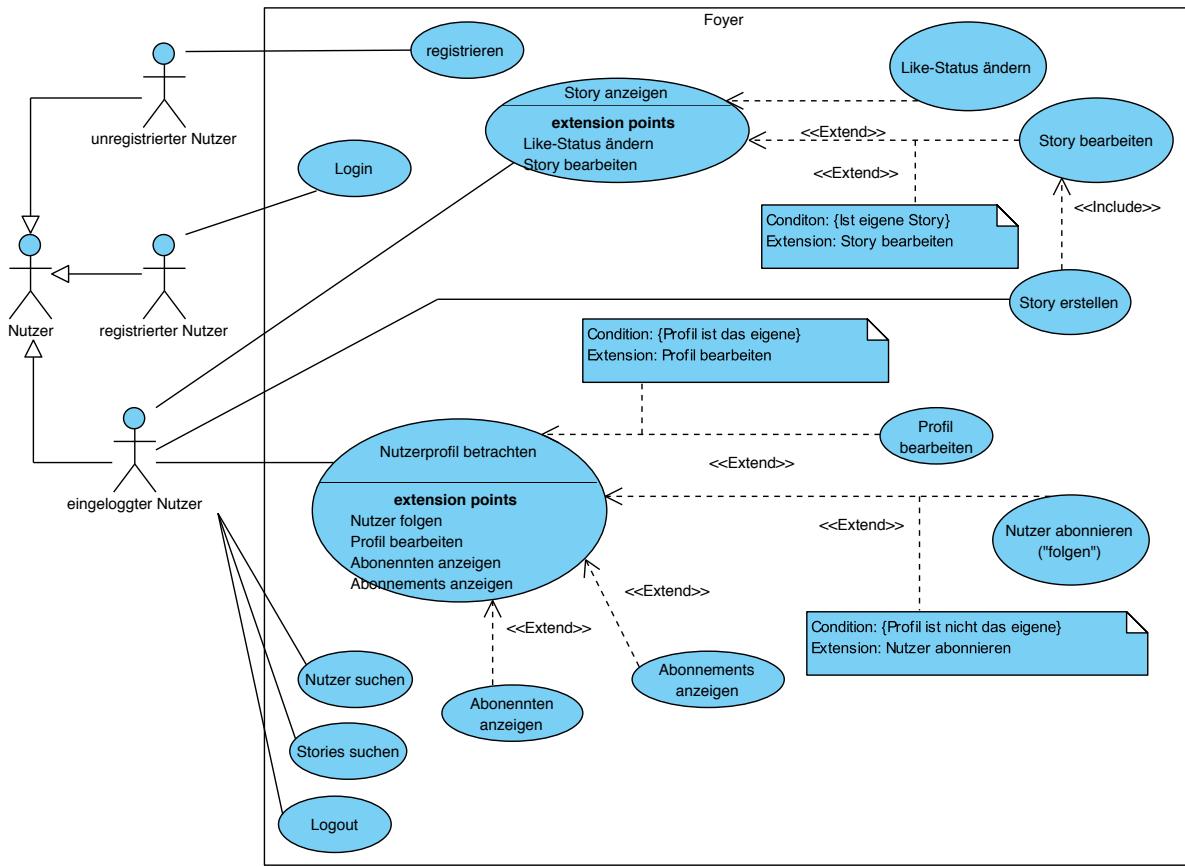


Abbildung 3.2 - Anwendungsfälle der Applikation
Quelle: eigene Darstellung

Das in Abbildung 3.2 dargestellte Anwendungsfalldiagramm zeigt die verschiedenen Aktionen, die ein Benutzer innerhalb der Anwendung durchführen kann. Ein unregistrierter Nutzer hat lediglich die Möglichkeit sich zu registrieren. Ein registrierter Nutzer kann sich nur einloggen und ein eingeloggter Nutzer kann die restlichen Aktionen ausführen. Bei einem eingeloggten Nutzer, der sich ausloggt handelt es sich nur noch um einen registrierten Nutzer.

Kern der Anwendung stellt die Aktion „Story anzeigen“ dar. Die Ausführung ermöglicht die Aktion „Like-Status ändern“. Falls die angezeigte Story dem Nutzer gehört, kann er die Aktion „Story bearbeiten“ ausführen, um eine Bearbeitung zu beginnen. Die Aktion „Story erstellen“ verwendet die Aktion „Story bearbeiten“.

Ein eingeloggter Nutzer kann auch die Aktion „Nutzerprofil betrachten“ ausführen. Falls es das eigene Profil ist, kann es durch die Aktion „Profil bearbeiten“ verändert werden. Falls dies nicht der Fall ist, kann durch die Aktion „Nutzer folgen“ der Follower-Status geändert werden. Die Aktion „Abonnenten anzeigen“ zeigt, welche Nutzer den Profilbesitzer abonniert haben. „Abonnements anzeigen“ zeigt die Nutzer an, die dem Profilbesitzers folgen. Der eingeloggte Nutzer kann „Logout“, „Nutzer suchen“ und „Stories suchen“ ausführen.

3.5 Zielgruppe

Die Zielgruppe der iPhone-Applikation sind alle Nutzer eines Apple iPhone 4 oder höher, die gerne Fotografieren und ihre Fotos mit anderen Menschen teilen möchten.

3.6 Einsatzgebiete

Kernaspekt der Anwendung ist die mobile Anwendbarkeit. Da die Anwendung viele Bilder lädt, ist hierbei darauf zu achten, dass die Größe der Bilder immer so gering wie möglich ausfällt, um das Datenvolumen des Nutzers nicht übermäßig zu belasten. Denkbar wäre hier eine gerätespezifische Auslieferung der Bilder durch den Server. (Geräte mit niedriger aufgelöstem Display bekommen kleinere aufgelöste Bilder). Beim erstellen einer neuen Story sollte die Qualität der hochzuladen gedachten Bilder nicht zu stark minimiert werden, da es sich zum einen um eine einmalige Aktion handelt und zum anderen eine mögliche Web-Version Bilder in höherer Auflösung benötigt. Als zu Veröffentlichen gedachte Inhalte sollen nicht nur private Erlebnisse in Frage kommen. Ebenso soll die Applikation für Anwendungsfälle nutzbar sein, die normalerweise über Blogsoftware bedient werden, z.B. Modeblogs, Kochrezepte, Tutorials.

4 Entwurf des Prototypen

Die Entwicklung des Prototypen erfolgt in mehreren Abschnitten. Es gilt einen Entwurf für die Datenhaltung (Kapitel 4.2) und für die Anwendungsschicht (Kapitel 4.3) sowie einen Prototyp des iPhone Clients zu erstellen (Kapitel 4.4).

4.1 Architektur

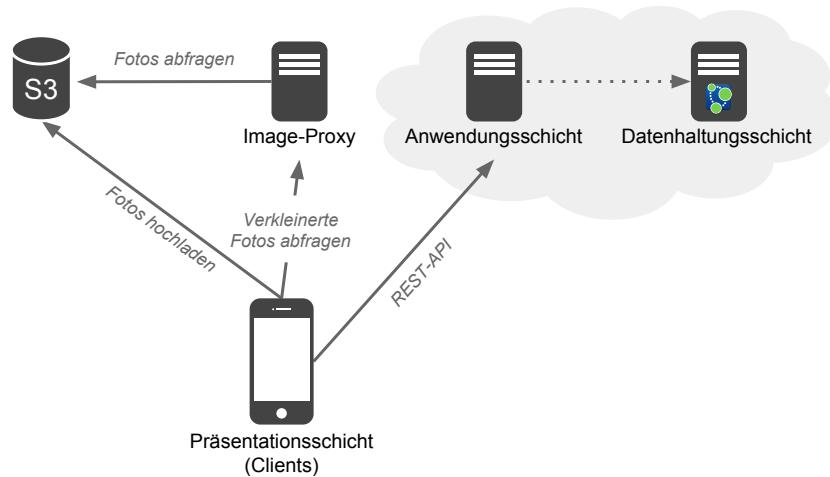


Abbildung 4.1 - Systemarchitektur
Quelle: eigene Darstellung

In den Anforderungen ist festgelegt, dass die Schnittstelle als Webservice realisiert werden muss. Dies lässt sich mit der Tatsache begründen, dass es einem mobilen Endgerät stets möglich sein muss Daten mit der Schnittstelle auszutauschen. Es handelt sich hierbei um eine Client-Server-Kommunikation.

Bezüglich der Architektur wird sich dem Grundkonzept der strengen Drei-Schichten-Architektur (*Three-Tier Architecture*) bedient, welche eine logische Trennung der einzelnen Systemschichten vorsieht und sich aus den folgenden Schichten zusammensetzt: *Präsentationsschicht*, *Anwendungsschicht* und *Datenhaltungsschicht*.⁷⁴ Formal wird zwischen strikter und flexibler Umsetzung unterschieden. Im Gegensatz zur flexiblen Drei-Schichten-Architektur, ist es in der strengen Variante der Präsentationsschicht nicht gestattet, direkt auf die Datenhaltungsschicht zuzugreifen (siehe Abb. 4.1). Diese Trennung ermöglicht eine Reduzierung der Abhängigkeiten innerhalb des Systems und erreicht folglich eine geringere Kopplung bei gleichzeitig höherer Kohäsion der einzelnen Schichten. Laut Coulouris, Dollimore, Kindberg und Blair⁷⁵ wird es durch diese Architektur möglich die Anwendungs- und Datenhaltungsschicht auf unterschiedlichen Servern zu betreiben. Des Weiteren ermöglicht diese physikalische Trennung es die Wartbarkeit des Gesamtsystems zu erhöhen und somit jeder Schicht, entsprechend der gewachsenen Anforderungen, weitere Hardwareressourcen zur Verfügung zu stellen.

⁷⁴ vgl. [SCHAL01] S. 23

⁷⁵ vgl. [COUGE01] S. 69

Es ist jedoch in Betracht zu ziehen, dass durch Weiterleitung und Transformation von Daten zwischen den einzelnen Schichten, die Ausführungsgeschwindigkeit der Applikation reduziert wird.⁷⁶

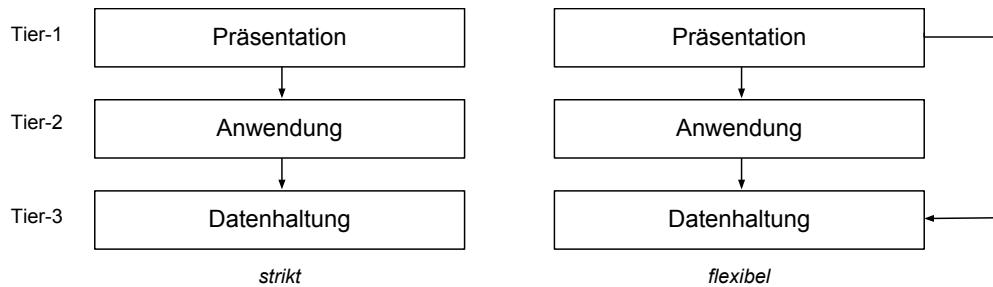


Abbildung 4.2 - Drei-Schichten-Architektur
Quelle: eigene Darstellung, angelehnt an Balzert, Heide⁷⁷

Die prototypisch zu implementierende iPhone-Applikation stellt die Präsentationsschicht innerhalb der Drei-Schichten-Architektur dar und ist für die Darstellung und Entgegennahme der von der Anwendungsschicht bearbeiteten Daten verantwortlich. Ferner interpretiert die Präsentationsschicht Benutzereingaben und generiert fachliche Aufrufe der Anwendungsschicht.

Die Anwendungsschicht beinhaltet die Geschäftslogik und nimmt eine Art Vermittlerrolle zwischen Präsentations- und Datenhaltungsschicht ein. Letztere trägt die Verantwortung für das Speichern und Laden von Daten.

4.2 Datenhaltungsschicht

Zur Persistierung der Daten bietet sich aufgrund der starken Vernetzung und der für Socialmedia typischen Anwendungsfälle („liken“, „folgen“, „abonnieren“) die Nutzung einer Graphdatenbank an.⁷⁸ Das Datenmodell bildet die Struktur der Datenhaltung des Webservices ab.

Der zeitliche Rahmen dieser Arbeit lässt die genauere Evaluierung verschiedener Graphdatenbanken nicht zu. Allerdings wurden diesem Thema schon andere wissenschaftliche Arbeiten gewidmet. Jouili und Vansteenbergh haben die Graphdatenbanken Neo4j, DEX, Titan und OrientDB Belastungstests unterzogen und nennen Neo4J als beste Option: „[...] [Neo4j] outperforms all the other candidates, regardless the workload or the parameters used. Concerning read-only intensive workloads, Neo4j, DEX, Titan-BerkeleyDB and Orient achieved similar performance.“⁷⁹ Sie ist nach einer Messung von db-engines.com die meistbenutzte Graphdatenbank (siehe Abb. 4.3) und garantiert demnach auch den größten Community-Support. Das ist für eine praktische Arbeit mit geringem Zeitfenster ein sehr wichtiges Entscheidungskriterium. Für hohe Verfügbarkeit lässt sich Neo4J, ähnlich wie MySQL, auch im Clustermodus betreiben. Dabei hält jedes Clustermitglied eine gesamte Kopie der Daten, während es einen Master gibt, bei dem die Slaves die aktualisierten Daten im Millisekunden-Takt abfragen können. Dadurch können Leseoperationen durch ein Loadbalancing auf die Slaves verteilt werden.

⁷⁶ vgl. [WIKI04]

⁷⁷ vgl. [BALHE01] S.143

⁷⁸ vgl. [ROBIA01] S. 5f

⁷⁹ [JOUSA01] S. 8

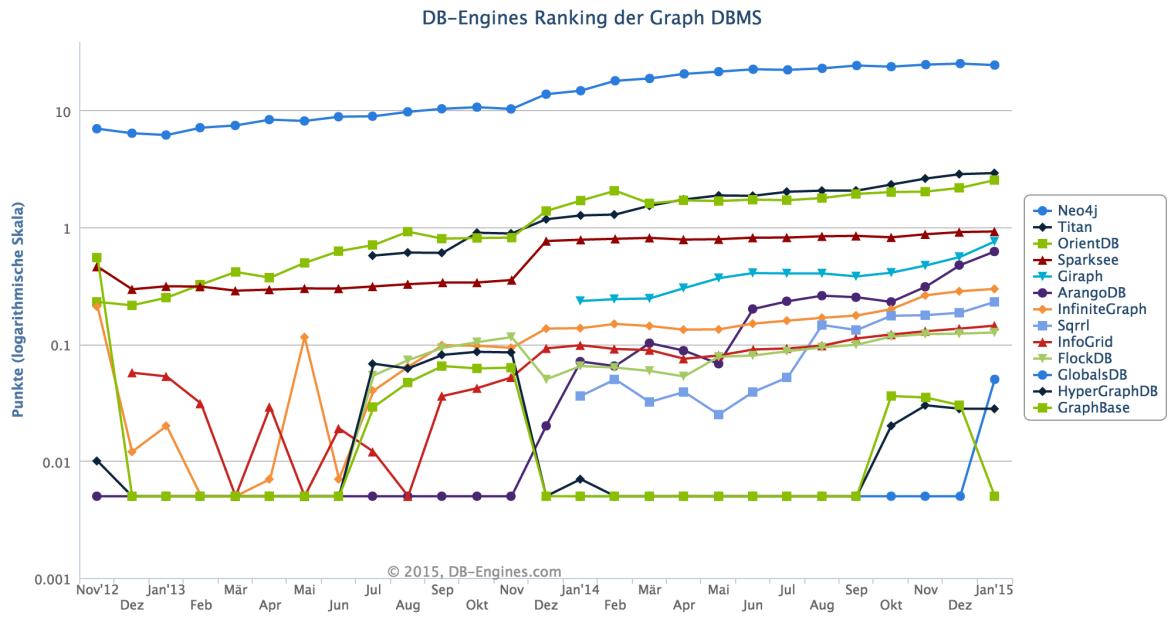


Abbildung 4.3 - logarithmisches Ranking der wichtigsten Graphdatenbanksysteme
Quelle: http://db-engines.com/de/ranking_trend/graph+dbms, abgerufen am 10.01.2015

4.2.1 Entwicklung einer Datenstruktur

Im Bereich der Graphdatenbanken und im Besonderen bei Neo4j wird das Design eines Schemas bzw. einer Struktur durch die Fragestellung an die Datenbasis bestimmt.⁸⁰ Obwohl es in Bezug auf Neo4j keinen festgelegten oder perfekten Weg der Datenmodellierung gibt, sollten bei der Entwicklung einige Richtlinien eingehalten werden, um das Datenmodell möglichst performant und erweiterbar gestalten zu können sowie mögliche Fehlentscheidungen zu vermeiden. Aus diesem Grund wird zunächst auf die sogenannten „Best Practices“ der Datenmodellierung in Bezug auf Neo4j eingegangen.

4.2.1.1 Best Practices der Datenmodellierung

Das Datenmodell ist Abhängig von der Fragestellung. Es sollte sich also nach den Abfragen richten, die an die Datenbasis gerichtet werden. Ähnlich wie bei der Normalisierung von relationalen Datenbanken, sollten auch Knoten so granular und klein wie möglich sein. Es empfiehlt sich also, Eigenschaften von Knoten als eigene Knoten zu modellieren. Das Abfragen von Daten aus dieser granularen Struktur ist im Vergleich zu relationalen Datenbanken schnell. Ferner ermöglicht dies detaillierte Anfragen an die Datenbasis. Beziehungen sollten aussagekräftig und wenig generisch sein. Detaillierte Beziehungsbezeichnungen ermöglichen genauere Anfragen an die Datenbank, da eine Graphenoperation dadurch meist weniger Beziehungen für das Ergebnis evaluieren muss. Sofern für die Beantwortung von Anfragen benötigt, können Indizes innerhalb des Graphen dargestellt werden. So können beispielsweise Datumsangaben als Baumstruktur modelliert werden.⁸¹

4.2.2 Entitäten in Neo4j

In Kapitel 3.1 wurde der Aufbau einer Story erläutert. Dessen einzelnen Komponenten stellen ebenfalls die zu modellierenden Entitäten dar: User, Story, Section und Item. Unter Beachtung der zuvor beschriebenen

⁸⁰ vgl. [BRUGRI] S. 80

⁸¹ vgl. [BRUGRI] S. 80-85

4.2.2.1 User

Ein Nutzer wird in der Entität User unter Verwendung des Labels “User” abgebildet. Die Attribute des Nutzers werden in der folgenden Tabelle aufgeführt und erläutert. Der Beziehungstyp “AUTHOR” referenziert den User als den Ersteller einer Story (Abb. 4.1), der Typ “LIKES” die getätigte “gefällt mir”-Angabe eines Nutzers (Abb. 4.2), und der Typ “FOLLOWS” das Abonnement eines Nutzers (Abb. 4.3). Bei letzterem ist anzumerken, dass die Beziehung in beide Richtungen möglich ist, da Nutzer sich gegenseitig abonnieren können. Die Beziehungstypen “STORIES” und “USERNEXT” stehen in engem Zusammenhang und werden in Abb 4.4 abgebildet. Dieses Beziehungsgeflecht ist genau genommen redundant zu der “AUTHOR” Referenz zwischen User und Story, erhöht aber die Abfrageperformance für den Fall, dass man alle Stories eines Users abfragen möchte. Diese “linked-list” ist ein verbreitetes Pattern in Neo4j.⁸² Die neueste Story ist der Kopf der Liste (Head) welcher über den Beziehungstyp “USERNEXT” die nächste Story referenziert und selbst durch den User über die Referenz “STORIES” angesteuert wird.



Abbildung 4.4 - Entität User mit Beziehungstyp AUTHOR / LIKES
Quelle: eigene Darstellung



Abbildung 4.4 - Entität User mit Beziehungstyp FOLLOWS
Quelle: eigene Darstellung



Abbildung 4.5 - Entität User
Quelle: eigene Darstellung

Attribut	Datentyp	Beschreibung
<u>userName</u>	String	Der Nutzernname des Nutzers und gleichzeitig der Primärschlüssel der Entität
email	String	Die E-Mail Adresse des Nutzers
password	String	Das verschlüsselte Passwort des Nutzers
token	String	Der Zugriffsschlüssel des Nutzers
tokenLifetime	Int	Ein Unix Zeitstempel, der das Ablaufdatum des Tokens darstellt
firstName	String	Der Vorname des Nutzers
lastName	String	Der Nachname des Nutzers
gender	String	Das Geschlecht des Nutzers

⁸² vgl. [ROBIA02]

Attribut	Datentyp	Beschreibung
followersCount	Int	Anzahl der Nutzer, die dem Nutzer abonniert haben
followingCount	Int	Anzahl der Nutzer, die der Nutzer abonniert hat
avatar	String	Dateiname des Nutzeravatars

Tabelle 4.1 - User Entität

Quelle: eigene Darstellung

4.2.2.2 Story

Die Entität Story bildet unter Verwendung des Labels "Story" den eigentlichen Kern der Applikation ab und wird in Tabelle 4.2 aufgelistet und beschrieben. Sie bildet den Knoten des kompletten Story-Graphen und referenziert über den Beziehungstyp "SECTIONS" dazugehörige Sektionen (Abb. 4.2.1). Ähnlich wie schon in 4.2.2.1 beschrieben, referenzieren sich auch alle Stories über den Beziehungstyp "ALLNEXT" zu einer "linked list".

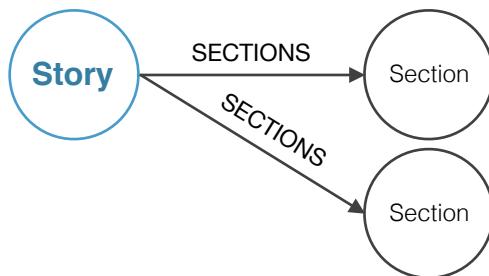


Abbildung 4.6 - Story Entität

Quelle: eigene Darstellung



Abbildung 4.7 - Verkettete Story Entität

Quelle: eigene Darstellung

Attribut	Datentyp	Beschreibung
objectID	String	Eindeutiger Identifikator
title	String	Titel der Story
createdAt	Int	Ein Unix Zeitstempel, der das Erstellungsdatum der Story darstellt
updatedAt	Int	Ein Unix Zeitstempel, der das letzte Bearbeitungsdatum der Story darstellt
likesCount	Int	Anzahl der "gefällt mir"-Angaben, die eine Story erhalten hat

Tabelle 4.2 - Story Entität

Quelle: eigene Darstellung

4.2.2.3 Section

Die Entität Section spiegelt einen Abschnitt (eine Sektion) innerhalb einer Story wieder und referenziert über den Beziehungstyp "ITEMS" dazugehörige Items. Aufgrund der abschätzbaren Menge von Sektionen innerhalb einer Story wird auf den Ansatz der "linked-list" zur Schaffung einer Chronologie verzichtet und stattdessen der Einfachheit halber in der Entität der Integer Wert "order" eingeführt. Die Entität-Section wird in der folgenden Tabelle 4.3 aufgeführt und beschrieben.

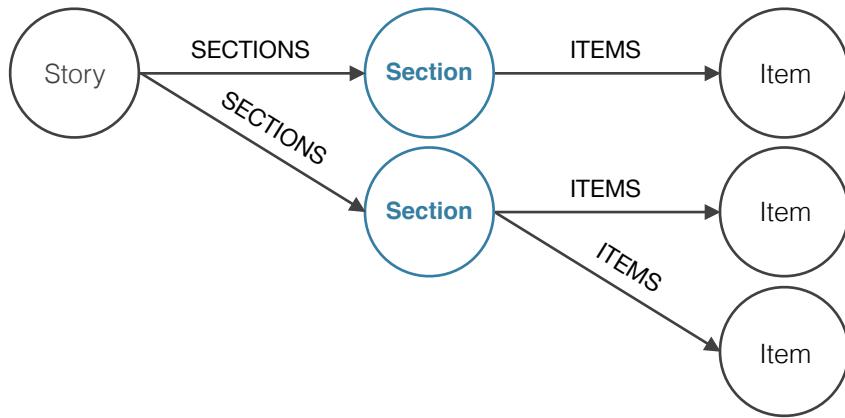


Abbildung 4.8 - Section Entität
Quelle: eigene Darstellung

Attribut	Datentyp	Beschreibung
objectID	String	Eindeutiger Identifikator
title	String	Titel der Sektion
text	String	Freitextfeld der Sektion
type	String	Typ des Layouts der Sektion
order	Int	Position der Sektion innerhalb der Story

Tabelle 4.3 - Section Entität
Quelle: eigene Darstellung

4.2.2.4 Item

Ein Item wird in der Entität Item unter Verwendung des Labels "Item" abgebildet. Die Attribute des Items werden in der folgenden Tabelle 4.4 aufgeführt und erläutert.

Attribut	Datentyp	Beschreibung
objectID	String	Eindeutiger Identifikator
fileName	String	Dateiname der Bildes
width	Int	Breite des Bildes
height	Int	Höhe des Bildes
order	Int	Position des Items innerhalb der Sektion
fillRow	boolean	Gibt an, ob es sich bei diesem Item um ein Zeilenfüllendes Element handelt

Tabelle 4.4 - Item Entität
Quelle: eigene Darstellung

4.2.3 Objektspeicher

Im Objektspeicher werden die vom Anwender hochgeladenen Fotos einer Story persistiert. Um eine Kapselung der Anwendungs- und Datenhaltungsschicht zu erreichen, und um die Möglichkeit der einfachen Austauschbarkeit zu gewährleisten, ist der Objektspeicher eine eigene Komponente in diesem verteilten System (Vgl. Abb. 4.1). Als Objektspeicher wird Amazon S3 (Simple Storage Service) gewählt, da es sich hierbei um

einen hoch skalierbaren und verfügbaren Webservice handelt, der sich zudem durch einfache Bedienbarkeit auszeichnet.⁸³ Ferner ist hervorzuheben, dass sich dieser Webservice schon in vorangegangenen Projekten bewährt hat und somit der Umgang hiermit bekannt ist.

Um den Netzwerk-Overhead zu minimieren, werden Objekte direkt vom Client (der Applikation) in das Applikations-Bucket geladen. Für den Upload von Objekten in ein S3 Bucket wird ein Schlüsselpaar, welches sich aus AccessKey (Public Key) und SecretAccessKey (Privatekey) zusammensetzt, benötigt.⁸⁴ Aufgrund der Tatsache, dass in die Applikation gespeicherte Zugangsdaten ein Sicherheitsrisiko darstellen würden (vgl. Kapitel 3.3.6), wird sich für die Autorisierung von Anfragen der Applikation der Methodik der “Pre-Signed-URL” bedient. Diese ermöglicht es, die Zugangsdaten in der Anwendungsschicht zu speichern und dort unter Verwendung der beiden Schlüssel URLs zu generieren, welche die jeweilige Operation autorisieren.

Amazon nutzt hier das sogenannte Public-Key-Verschlüsselungsverfahren um eine Signatur zu erstellen. Neben dem Schlüsselpaar benötigt man für die Erstellung der URL zudem den Namen des Buckets und des Objekts, die zu verwenden gewünschte HTTP-Methode und die Verfallszeit der URL als UNIX-Zeitschlüssel.⁸⁵ Eine solche URL setzt sich wie folgt zusammen:

```
https://<Bucket>.s3.amazonaws.com/<Objekt>?Expires=<Expires-Timestamp>&AWSAccessKeyId=<AccessKeyId>&Signature=<Signature>
```

Abbildung 4.9 - Amazon S3 signierte URL
Quelle: eigene Darstellung

Die Signatur ist ein unter Verwendung des Privatekeys (SecretAccessKey) erstellter Hashwert (HMAC), welcher eine Kombination von HTTP-Methode, Verfallszeit sowie Bucket und Objekt beinhaltet und mittels Base64 kodiert wird.

4.2.4 Proxyserver

In Abbildung 4.2.1 ist festgelegt, dass der Zugriff des Clients auf die Fotos nicht direkt über den Objektspeicher Amazon S3, sondern über einen *Image Proxy Server* erfolgen soll. Image Proxy Server lassen sich dem Cloud-Servicemodell *Infrastructure as a Service* (siehe 2.6.2) zuordnen und ermöglichen die On-Demand-Bearbeitung von Bildern mittels URL Parametern. Dieses Vorgehen hat zum Vorteil, dass verkleinerte Versionen der hochgeladenen Bilder nicht erstellt und vorgehalten werden müssen und somit ein rechen- und speicherintensiver Teil der Infrastruktur outgesourced werden kann. Außerdem bietet diese Handhabung die Möglichkeit, Bilder für das anfragende Medium zu optimieren, beispielsweise um durch Anpassung der Auflösung Bandbreite zu sparen. Dieser Service wird von zahlreichen Anbietern wie resize.ly⁸⁶, embed.ly⁸⁷, imgix⁸⁸ oder images.weserv.nl⁸⁹ angeboten. Alle genannten Services bieten als Kernfunktionalität das Verkleinern und anschließende Zwischenspeichern von Bildern an. Für dieses Projekt wird der Dienst images.weserv.nl gewählt, da es sich um

⁸³ vgl. [AWS08]

⁸⁴ vgl. [AWS06]

⁸⁵ vlg. [AWS07]

⁸⁶ <http://resize.ly>, abgerufen am 10.02.2015

⁸⁷ <http://embed.ly>, abgerufen am 10.02.2015

⁸⁸ <http://imgix.com>, abgerufen am 10.02.2015

⁸⁹ <http://images.weserv.nl>, abgerufen am 10.02.2015

einen kostenlosen Service handelt und er zudem Teil eines *Content Delivery Networks* (CDN) ist, welches die weltweite Erreichbarkeit der Inhalte beschleunigt.⁹⁰ Die Nutzung des Services geschieht folgendermaßen:⁹¹

- ?url=[Link zum Bild]
- &h= maximale Höhe (optional)
- &w= maximale breite (optional)

`http://images.weserv.nl/?url=www.google.nl/logos/logo.gif&h=30&w=40`

Abbildung 4.10 - Beispiel einer URL zum Erhalt eines Bildes mittels images.weserv.nl
Quelle: eigene Darstellung

4.3 Anwendungsschicht

Die Anwendungsschicht beinhaltet die Geschäfts- und Anwendungslogik des Webservices und interagiert sowohl mit dem Endgerät als auch mit der Datenbank. Der Webservice wird als REST Webservice entworfen. Diese Entscheidung beruht auf der Tatsache, dass das REST-Paradigma Wiederverwendbarkeit, Erweiterbarkeit, Skalierbarkeit und die einfache Integration in die vorhandene Netz-Struktur bietet.⁹² Ferner ist im “[...] WWW bereits ein Großteil der für REST nötigen Infrastruktur (z. B. Web- und Application-Server, HTTP-fähige Clients, HTML- und XML-Parser, Sicherheitsmechanismen) vorhanden, und viele Web-Dienste sind per se REST-konform”.⁹³ Des Weiteren kommt hinzu, dass das REST Paradigma kein bestimmtes Datenaustauschformat vorgibt.

Um die Kommunikation zwischen Datenhaltungsschicht und Anwendungsschicht so einfach wie möglich zu gestalten, wird als applikationsweites Datenaustauschformat JSON genutzt, da dies bereits von Neo4j zu diesem Zweck eingesetzt wird. Aufgrund der Tatsache, dass JSON selbst gültigen JavaScript-Code repräsentiert, kann die Zeichenfolge direkt ausgeführt und somit in ein JavaScript-Objekt überführt werden.

Für die spätere Implementierung des REST-Webservices wird die Plattform Node.js verwendet. Laut Selbstdarstellung⁹⁴ ist Node.js perfekt geeignet für Datenintensive, schnelle und skalierbare Netzwerkanwendungen. Da Node.js auf der von Google entwickelten JavaScript V8 Laufzeitumgebung basiert, also die Sprache JavaScript verwendet wird, ist die Wahl von Node.js in Kombination mit JSON naheliegend.⁹⁵

Um die Modularität und die Möglichkeit der Wartbarkeit zu steigern, wird die Anwendungsschicht in weitere Teile, die REST-, die Service- und die Persistenz-Schicht, untergliedert. Die einzelnen Schichten werden im folgenden näher beschrieben und in Abbildung 4.11 veranschaulicht.

⁹⁰ vgl. [CLOFL01]

⁹¹ vgl. [WESERV01]

⁹² vgl. [DAZMI01]

⁹³ vgl. [WIKI05]

⁹⁴ vgl. [NODEJS01]

⁹⁵ vgl. [WIKI06]



Abbildung 4.11 - Gliederung der Anwendungsschicht

Quelle: eigene Darstellung

Die REST-Schicht stellt die Schnittstelle des Webservices dar. Sie nimmt Anfragen entgegen und sendet dem Anfragesteller nach einer Validierung eine entsprechende Antwort zurück. Sofern die Anfrage alle zur Verarbeitung nötigen Informationen enthält, gibt sie diese in die Service-Schicht weiter, welche diese weitererarbeitet. Die Service-Schicht beinhaltet die eigentliche Anwendungslogik. Von hier aus werden beispielsweise die Daten einer Story aufbereitet und mit dem Aufruf einer Methode aus der Persistenz-Schicht in der Datenbank gespeichert. Durch die Persistenz-Schicht wird der Zugriff auf die Datenhaltung abgebildet. Sie stellt beispielsweise Methoden zum Erzeugen, Lesen, Aktualisieren und Löschen von Daten zur Verfügung.

Da es sich bei JavaScript um eine klassenlose Scriptsprache handelt, dementsprechend Klassen im Sinne von objektorientierten Sprachen wie Java oder C# nicht umsetzbar sind, werden die einzelnen Schichten und deren Funktionalitäten in Modulen gekapselt.

4.3.1 REST-Schicht

Die REST-Schnittstelle stellt einen zentralen Punkt der Arbeit dar, und ist die Grundvoraussetzung für den Einsatz der mobilen Anwendung, denn jegliche Informationen und Daten welche innerhalb der mobilen Applikation dargestellt und verarbeitet werden, müssen über diese Schnittstelle bezogen werden.

Für die REST-Schicht ist die Verwendung des Frameworks Koa.js⁹⁶ vorgesehen. Es handelt sich hierbei um ein extrem lichtgewichtiges Framework, welches auf einige neuen Funktionen von EcmaScript – der Basis von JavaScript⁹⁷ – in Version 6 aufsetzt, die es unter anderem ermöglichen, das Schreiben von asynchronen Code zu vereinfachen und das für JavaScript typische Callback-Problem zu vermeiden.⁹⁸ Die Dokumentation von Koa bezeichnet Koa darüber hinaus eher als Erweiterung des Node.js http-Moduls als ein vollwertiges Framework für Webanwendungen. Dies liegt in der Tatsache begründet, dass Koa sehr modular gestaltet ist und nur die Funktionen enthält, die für einen HTTP-Server relevant sind. Es verzichtet in der Standardversion auf jegliche Middleware-Module, bietet dem Entwickler also die Möglichkeit, sich das Framework individuell nach den jeweiligen Bedürfnissen zusammenzustellen.⁹⁹

⁹⁶ <http://koajs.com>, abgerufen am 10.02.2015

⁹⁷ vgl. [WIKI07]

⁹⁸ vgl. [KOA01]

⁹⁹ vgl. [KOA02]

Koa ist das Nachfolger-Framework von Express.js, welches, laut Messungen der Standart-Paketverwaltung für Node.js npm¹⁰⁰, mit 1,5 Millionen Downloads monatlich¹⁰¹ das beliebteste Framework ist.

Aus den im 3. Kapitel definierten Anforderungen können die benötigten Funktionalitäten abgeleitet werden, die durch die REST-Schnittstelle bereitgestellt werden müssen.

Diese Anforderungen lassen sich in drei Ressourcen einteilen und werden in den Tabellen 4.5, 4.6 und 4.7 beschrieben. Geschweifte Klammern stellen innerhalb der einzelnen Methoden Platzhalter dar, welche in der Applikation durch „echte“ Daten ersetzt werden.

4.3.1.1 Autorisierung

Zur Verwendung der Schnittstellen ist es erforderlich Autorisierungsdaten im Header der Anfrage mit zu senden. Eine jede Anfrage muss das Feld “api-token” enthalten, welches genutzt wird, um eine unautorisierte Nutzung der API zu verhindern. Um eine Anfrage an die Ressourcen User und Stories zu stellen, ist es zudem obligatorisch das Feld “user-token” zu setzen. Dies ermöglicht es der Service-Schicht, den aktuellen Nutzer zu identifizieren.

4.3.1.2 REST-URI-Design

REST APIs nutzen URIs, um Ressourcen zu adressieren. Nach Mark Massé¹⁰² ist eine URI zu bevorzugen, welche leicht zu verstehen ist:

```
http://api.example.restapi.org/france/paris/louvre/leonardo-da-vinci/mona-lisa
```

Abbildung 4.12 - Beispiel einer leicht zu verstehenden URI
Quelle: eigene Darstellung

Eine URI, deren Bedeutungen nicht erkennbar wäre, würden sich beispielsweise folgendermaßen lesen:

```
http://api.example.restapi.org/68dd0-a9d3-11e0-9f1c-0800200c9a66
```

Abbildung 4.13 - Beispiel einer URI, dessen Bedeutung nicht erkennbar ist
Quelle: eigene Darstellung

Die URIs sollten das Ressourcenmodell selbsterklärend vermitteln. Folgende Designregeln des URI-Designs gibt es nach Messé zu beachten:

- Der Slash (/) soll genutzt werden, um die Hierarchien deutlich machen
- URIs sollten keinen Slash am Ende haben
- Zur Verbesserung der Lesbarkeit sollten Bindestriche (-) genutzt werden
- Unterstriche sollten nicht in URIs benutzt werden
- Kleinbuchstaben sind in URI-Pfaden zu bevorzugen
- Dateierweiterungen (.json, .xml usw) sollten in der URL nicht enthalten sein
- Namen von Collections sollten im Plural stehen

¹⁰⁰ vgl. [NPM02]

¹⁰¹ vgl. [NPM03]

¹⁰² vgl. [MASMA01] S. 11ff

4.3.1.3 Ressourcen

4.3.1.3.1 Account

Die Ressource “account” dient dazu, Anwender am System anzumelden oder nicht registrierte Nutzer in das System aufzunehmen.

URI	Methode	Beschreibung
/account/register	POST	Fügt einen neuen Anwender zum System hinzu
/account/login	POST	Meldet einen Nutzer am System an Parameter: email*, password*
/account/forgotpassword	POST	Sendet dem Nutzer eine E-Mail zum zurücksetzen des Passworts Es muss die E-Mail Adresse übergeben werden Parameter: email *

Tabelle 4.5 - Account Resource
Quelle: eigene Darstellung

4.3.1.3.2 User

Über die Ressource “user”, lassen sich jegliche Aktionen steuern, die den Nutzer betreffen.

URI	Methode	Beschreibung
/user	GET	Gibt den aktuellen Nutzer zurück
/user/{userName}	GET	Liefert einen bestimmten Nutzerdatensatz zum übergebenen Benutzernamen zurück
/user	PATCH	Bearbeitet den aktuellen Nutzer Parameter: email, firstName, lastName, userName, gender
/user/password	PATCH	Bearbeitet das Password des aktuellen Nutzers Parameter: currentPassword*, newPassword*
/user/avatar	DELETE	Löscht das Avatar des aktuellen Nutzers
/user/{userName}/follow	PUT	Abboniert den Nutzer zum übergebenen Benutzernamen
/user/{userName}/follow	DELETE	Löscht das Abonnement zum Nutzer des übergebenen Benutzernamens
/user/{userName}/followers	GET	Liefert ein Array mit Nutzern zurück, die dem Nutzer des übergebenen Benutzernamens folgen
/user/{userName}/following	GET	Liefert ein Array mit Nutzern zurück, denen der Nutzer des übergebenen Benutzernamens folgt

Tabelle 4.6 - User Ressource
Quelle: eigene Darstellung

4.3.1.3.3 Stories

Die Ressource “Stories” ermöglicht es, Stories zu konsumieren, zu erstellen und zu bearbeiten.

URI	Methode	Beschreibung
/stories?{lastObjectId}&{amountPerPage}	GET	Liefert ein Array mit Stories zurück, sortiert nach Erscheinungsdatum
/stories/{storyId}	GET	Liefert ein Story-Objekt zurück
/stories	POST	Bearbeitet eine existierende (wenn die objectId gesetzt ist) oder erstellt eine neue Story
/stories/{storyId}/like	PUT	Speichert die „gefällt mir“-Angabe des aktuellen Nutzers
/stories/{storyId}/like	DELETE	Löscht die „gefällt mir“-Angabe des aktuellen Nutzers

Tabelle 4.7 - Story Ressource
Quelle: eigene Darstellung

4.3.2 Service-Schicht

Die Service-Schicht beinhaltet die eigentliche Anwendungslogik der Anwendungsschicht und agiert als Vermittler zwischen REST- und Persistenzschicht indem sie die, über die REST-Schicht angebotenen Funktionen kapselt und die weitergegebenen Anfragen verarbeitet.

4.3.3 Persistenz-Schicht

Die Persistenz-Schicht stellt Methoden zur Kommunikation der Service-Schicht mit der eigentlichen Datenhaltung bereit. Durch die Einführung dieser Schicht, soll die Austauschbarkeit der Datenbank ermöglicht werden. Die Schicht gliedert sich in zwei Module, welche die Funktionen der zwei wichtigsten Entitäten, Story und User, bereitstellen. Für die Kommunikation mit dem Datenbankserver ist die Verwendung des Frameworks „seraph“¹⁰³ vorgesehen. Mit Hilfe dieses Frameworks lassen sich unter anderem Abfragen in Neo4js nativer Abfragesprache Cypher formulieren und zudem Batchoperationen (Stapelverarbeitung) durchführen.

4.3.4 Ausfallsicherheit und Skalierbarkeit

Um sicherzustellen, dass die Node.js-Applikation auch nach einem Neustart des eigentlichen Servers oder nach einem schwerwiegenden Fehler wieder korrekt hochfährt, wird sich dem Node.js-Programm PM2 bedient. Es bietet zusätzlich die Option sogenannte „Watcher“ einzurichten. Dieser beobachtet den Projektordner der Applikation und führt bei Dateiänderungen automatisch einen Neustart der Applikation durch.

Des Weiteren ermöglicht es PM2, die Node.js-Applikation entsprechend der Anzahl der CPU-Kerne des Servers in mehreren Prozessen zu starten („Cluster“, siehe Abb. 4.14). Dies hat zum Vorteil, dass ohne Veränderung am Programmcode die Hardware optimal ausgenutzt werden kann. PM2 agiert dann als *LoadBalancer* und verteilt die HTTP-, TCP oder UDP-Anfragen gleichmäßig auf alle gestarteten Instanzen („Worker“). Um bei einem Neustart der Applikation eine Nichterreichbarkeit („Downtime“) auszuschließen, startet PM2 jeden Arbeiter nacheinander neu und verteilt in der Zwischenzeit die Last auf die anderen Arbeiter.¹⁰⁴

¹⁰³ vgl. [SERAPH01]

¹⁰⁴ vgl. [PM201]

Malte:oyer-backend Malte\$ pm2 list									
App name	id	mode	PID	status	restarted	uptime	memory	watching	
oyer-api	0	cluster	1412	online	0	5s	28.762 MB	activated	
oyer-api	1	cluster	1413	online	0	5s	29.398 MB	activated	
oyer-api	2	cluster	1414	online	0	5s	29.383 MB	activated	
oyer-api	3	cluster	1415	online	0	5s	29.219 MB	activated	
oyer-api	4	cluster	1416	online	0	5s	29.133 MB	activated	
oyer-api	5	cluster	1417	online	0	4s	29.242 MB	activated	
oyer-api	6	cluster	1418	online	0	4s	29.047 MB	activated	
oyer-api	7	cluster	1419	online	0	4s	29.461 MB	activated	
git-hook	8	fork	1420	online	0	4s	17.598 MB	activated	

Abbildung 4.14 - pm2 Cluster
Quelle: eigene Darstellung

4.4 Präsentationsschicht - iPhone App

In den folgenden Kapiteln soll die prototypische iPhone-Applikation entwickelt werden. Die Umsetzung soll in der Programmiersprache Swift erfolgen. Swift ist im Jahr 2014 erschienen und ergänzt die bisherige Programmiersprache für Apple Software, Objective-C.

Der Grund für die Wahl von Swift als Programmiersprache für den Prototypen ist zum einen der Tatsache geschuldet, dass Swift Standards implementiert, die schon seit einiger Zeit in anderen Programmiersprachen vorhanden sind und der 30 Jahre alten Sprache Objective-C fehlen. Dazu zählen beispielsweise Tupel, *generische Programmierung* („generics“), verschachtelte Funktionen („nested functions“), Typinferenz, Namensräume („Namespaces“) oder funktionale Programmiermuster.¹⁰⁵ Zum anderen ist Swift laut Apple bis zu 2,6x schneller als Objective-C¹⁰⁶ und wird als ein Nachfolger der Sprachen C und Objective-C beschrieben („Swift is a successor to the C and Objective-C languages“¹⁰⁷). Ein weiterer Vorteil ist, dass es die Programmierumgebung Xcode ermöglicht, Klassen sowohl in Swift als auch Objective-C zu programmieren und diese miteinander zu verwenden. Somit lässt sich einer der größten Vorteile von Objective-C nämlich seine Verbreitung (der überwiegende Teil an Bibliotheken und Frameworks ist in dieser Sprache umgesetzt) auch in Swift nutzen.

4.4.1 Frameworks und Bibliotheken

Neben den, im SDK enthaltenen Frameworks sollen folgende die Umsetzung erleichtern und die Usability verbessern:

4.4.1.1 Haneke

Haneke ist eine in Swift geschriebene Cache Bibliothek, die es beispielsweise ermöglicht Bilder zwischenspeichern. Zum einen stellt sie einen In-Memory Cache bereit, welcher auf NSCache basiert, zum anderen einen LRU (Least Recently Used) Festspeicher Cache.¹⁰⁸

¹⁰⁵ vgl. [APLSWI02]

¹⁰⁶ vgl. [APLSWI01]

¹⁰⁷ [APLSWI02]

¹⁰⁸ vgl. [HANEKE01]

4.4.1.2 Alamofire

Alamofire ist eine in Swift programmierte HTTP-Netzwerk Bibliothek, welche auf die System-Bibliothek NSURLSession aufsetzt und den Umgang mit Netzwerkressourcen vereinfacht.¹⁰⁹

4.4.1.3 SwiftForms

SwiftForms ist eine in Swift geschriebene Bibliothek, die es erlaubt Formulare mit verhältnismäßig wenig Code zu definieren. Ferner ist sie sehr flexibel und ermöglicht die problemlose Erweiterung und Anpassung der Benutzeroberfläche.¹¹⁰

4.4.1.4 FloatLabelFields

FloatLabelFields ist die Swift Implementation des UX Patterns mit dem Namen “Float Label Pattern”, welches ein Usability-Problem von Formularen in kleinen Bildschirmen behebt. Aufgrund des Platzmangels ist es üblich, Bezeichner von Eingabefeldern innerhalb des Eingabefelds zu positionieren (Placeholder). Handelt es sich um ein vorausgefülltes Formular oder hat der Nutzer schon etwas eingegeben, so werden diese durch die Eingabe ersetzt und der Nutzer kann die jeweiligen Eingaben nicht mehr zuordnen. Das Pattern löst dieses Problem, indem es den Bezeichner bei Texteingabe über das Formularfeld bewegt.¹¹¹

4.4.1.5 ACScrollNavBar

Die in Swift geschriebenen Klassenerweiterungen für UINavigationBar und UINavigationController sorgen dafür, dass die Titelleiste am oberen Rand des Bildschirms der Scrollbewegung folgt und somit mehr Platz für den eigentlichen Inhalt vorhanden ist. Dieses Verhalten wurde von Apple beispielsweise in Safari implementiert.

4.4.1.6 GKPopoverController

Diese in Objective-C geschriebenen Klassen ahmen die Klasse UIPopoverController nach, welche von Apple nur für das iPad vorgesehen wurden.¹¹²

4.4.1.7 NHBalancedFlowLayout

In Objective-C geschriebenes Layout für UICollectionView, das durch Anwendung des *Partition Problem* Algorithmus¹¹³ ein ausgeglichenes Raster umsetzt, welches dafür sorgt, dass die einzelnen Zellen der Collection View die Bildschirmfläche optimal ausnutzen.¹¹⁴

4.4.2 Views

Die verschiedenen Oberflächen innerhalb der Anwendung werden Views (zu deutsch “Sichten”) genannt. Jeder View wird hierbei eine spezielle Funktion zugeordnet, die es dem Anwender ermöglicht mit der Applikation zu interagieren. In den folgenden Kapiteln sollen die einzelnen Views, die in der Applikation zum Einsatz kommen,

¹⁰⁹ vgl. [ALAM01]

¹¹⁰ vgl. [SWIFO01]

¹¹¹ vgl. [FLOLA01]

¹¹² vgl. [GKPOP01]

¹¹³ vgl. [SKIST01]

¹¹⁴ vgl. [DEHNJ01]

beschrieben und deren jeweilige Aufgabe erläutert werden. Ferner wird beschrieben, wie die Umsetzung der Views mit erfolgen kann und welche Systemkomponenten sich hierfür eignen.

In den folgenden Kapiteln werden sowohl die Views aus 3.2.1 (Muss-Anforderungen) als auch aus 3.2.2 (Kann-Anforderungen) anhand von Mockups beschrieben. Für alle dargestellten Views, außer den in Kapitel 4.4.2.2 beschriebenen, muss der Anwender der Applikation eingeloggt sein.

4.4.2.1 App-Übergreifendes

Zentrales Navigationselement innerhalb der Applikation ist die sogenannte TabBar, welche fix am unteren Rand des Bildschirms positioniert ist und dem Nutzer die Möglichkeit gibt, zwischen mehreren Views zu wechseln. Innerhalb der Applikation wird sie verwendet, um dem Nutzer schnellen Zugriff auf das Newsfeed, die Entdeckungsfunktion, den Editor, das eigene Nutzerprofil sowie die Einstellungen zu geben. (siehe Abb. 4.16.1 unten). Für die Implementierung dieser Komponente ist der in 2.11.2.2 beschriebene *UITabBarController* vorgesehen. Er stellt den Ursprungs-Viewcontroller der Applikation dar und instanziert die Viewcontroller, die in der Tab Bar angezeigt sein sollen. Ferner werden alle diese Viewcontroller in UINavigationController gekapselt, um dem Nutzer über ein vertrautes UI-Element stets die Möglichkeit zu geben, sich in der Hierarchie der Viewcontroller zu bewegen. Am oberen Bildschirmrand befindet sich in den meisten modellierten Views die sogenannte UINavigationBar und beinhaltet standardmäßig einen Titel und zwei Buttons, um es dem Anwender zu ermöglichen, sich durch die View-Hierarchie zu navigieren. Hierbei handelt es sich ebenfalls um eine Standardkomponente.

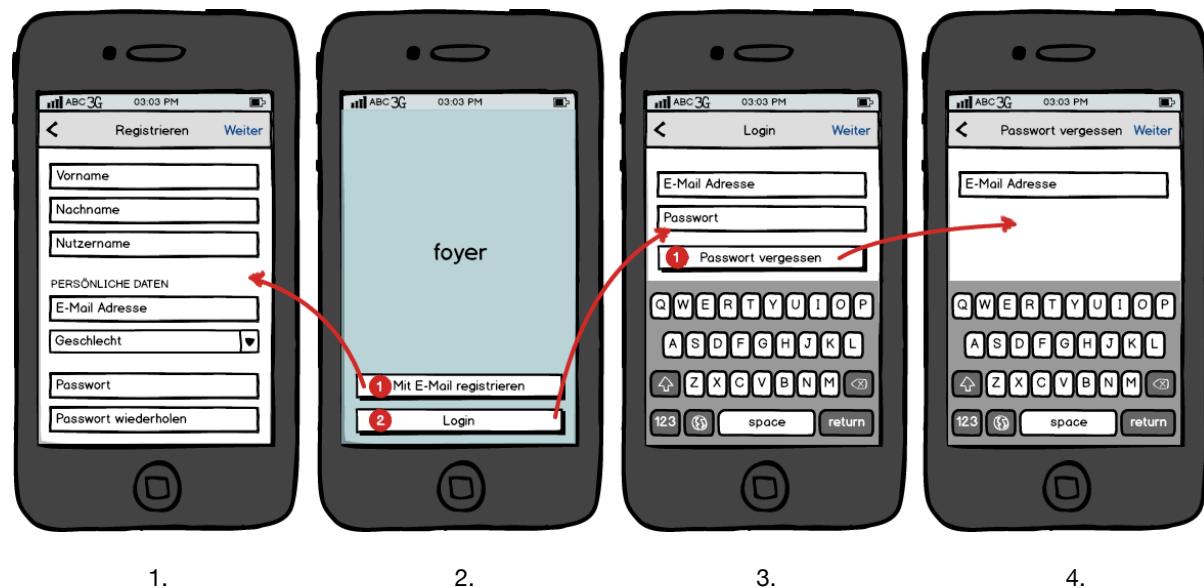


Abbildung 4.15 - Mockups ausgeloggter Nutzer
Quelle: eigene Darstellung

4.4.2.2 Ausgeloggter Nutzer - Registrierung, Login, Passwort vergessen

Die in Abb. 4.15.1 - 4.15.4 dargestellten Views sind für Anwender relevant, die noch kein Benutzerkonto besitzen oder nicht eingeloggt sind. Ist dies der Fall, wird die View des Startbildschirms (Abb. 4.15.2) angezeigt. Von hier aus hat der Anwender die Möglichkeit, durch das Berühren des Buttons [1] zur Registrierungsview (Abb. 4.15.1) zu gelangen. Das Formular in dieser View umfasst Nutzerdaten, die für den späteren Login in der Anwendung und für das Nutzerprofil relevant sind. Bei allen Angaben (Vorname, Nachname, Nutzernname, E-

Mail Adresse Geschlecht und Passwort) handelt es sich um Pflichtangaben. Ferner sorgt der Button [2] in Abb. 4.15.2. dafür, dass dem Anwender die in Abb. 4.15.3. illustrierte View dargestellt wird und es ihm ermöglicht, sich mit Anmeldeinformationen, bestehend aus E-Mail Adresse und Passwort, einzuloggen. Durch Berühren des Buttons [1] in Abb. 4.15.3 wird dem Nutzer die View aus Abb. 4.15.4 präsentiert und er hat die Möglichkeit unter Angabe der E-Mail Adresse sein Passwort zurückzusetzen. Die Daten aus den abgebildeten Views mit Formularen (Abb. 4.15 1, 3, 4) werden, nach Berühren des jeweiligen “Senden-Buttons”, zunächst clientseitig auf Ihre Validität überprüft. Sind alle Daten vollständig und valide, werden die Daten an den Server gesendet und der jeweilige Schritt ist abgeschlossen.

Umsetzung

Für die Implementierung der in 4.15.1, 4.15.3 und 4.15.4 abgebildeten Formulare ist die Nutzung der in Kapitel 4.4.1.3 eingeführten Bibliothek SwiftForms in Verbindung mit der in Kapitel 4.4.1.4 beschriebenen Bibliothek FloatLabelFields vorgesehen. Die Umsetzung des Startbildschirms (Abb. 4.15.2) erfolgt mittels eines regulären UIViewControllers in Verbindung mit zwei UIButtonns.

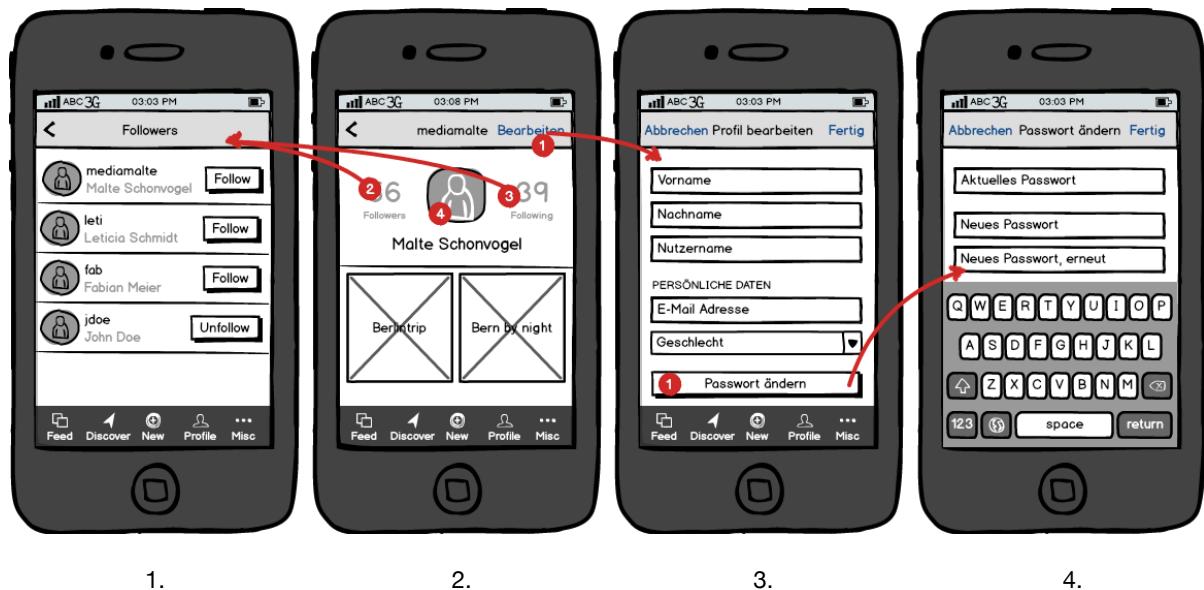


Abbildung 4.16 - Mockups ausgeloggter Nutzer
Quelle: eigene Darstellung

4.4.2.3 Nutzerprofil

Das Nutzerprofil ist neben der Story und dem Feed ein wesentliches Element der Applikation. Dies liegt unter anderem darin begründet, dass man über Nutzerprofile weitere Nutzerprofile und deren Inhalte entdecken kann und sich der Nutzer selbst über sein Profil den anderen Nutzern präsentiert.

Beim Nutzerprofil handelt es sich nicht um eine View, im Sinn der zuvor erwähnten Views, sondern vielmehr um eine Komponente, welche an verschiedenen Stellen der Applikation dynamisch eingebunden werden kann.

Der Aufbau dieser Komponente ist in zwei Bereiche unterteilt (Vgl. Abb 4.16.2). Der erste Bereich des Nutzerprofils beinhaltet als zentrales Element das Profilbild (Abb 4.16.2 [4]). Die Anzahl der Abonnenten (Abb 4.16.2 [2]) und die Anzahl der Nutzer, die der Profilnutzer abonniert hat (Abb 4.16.2 [3]), befindet sich links bzw. rechts vom Profilbild und führen durch Berühren auf die in Abb. 4.16.1 dargestellte View, welche die Nutzer mit Avatar, Nutzernamen, Vor- und Nachnamen und einem Button zum Folgen auflistet. Ferner beinhaltet dieser Bereich den Vor- und Nachnamen des Nutzers.

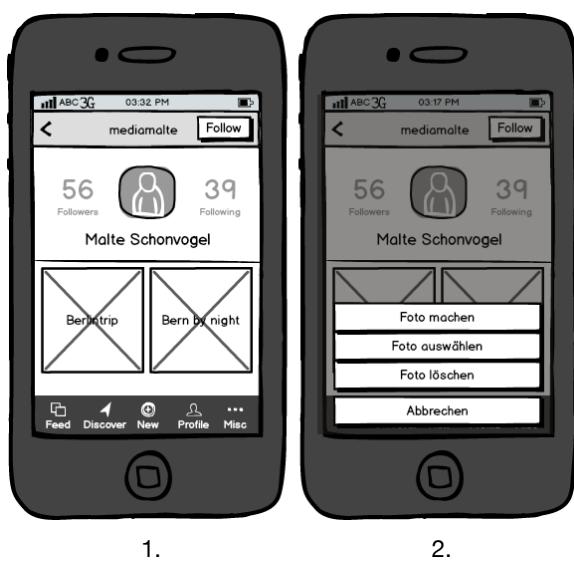


Abbildung 4.17 - Mockups ausgeloggter Nutzer
Quelle: eigene Darstellung

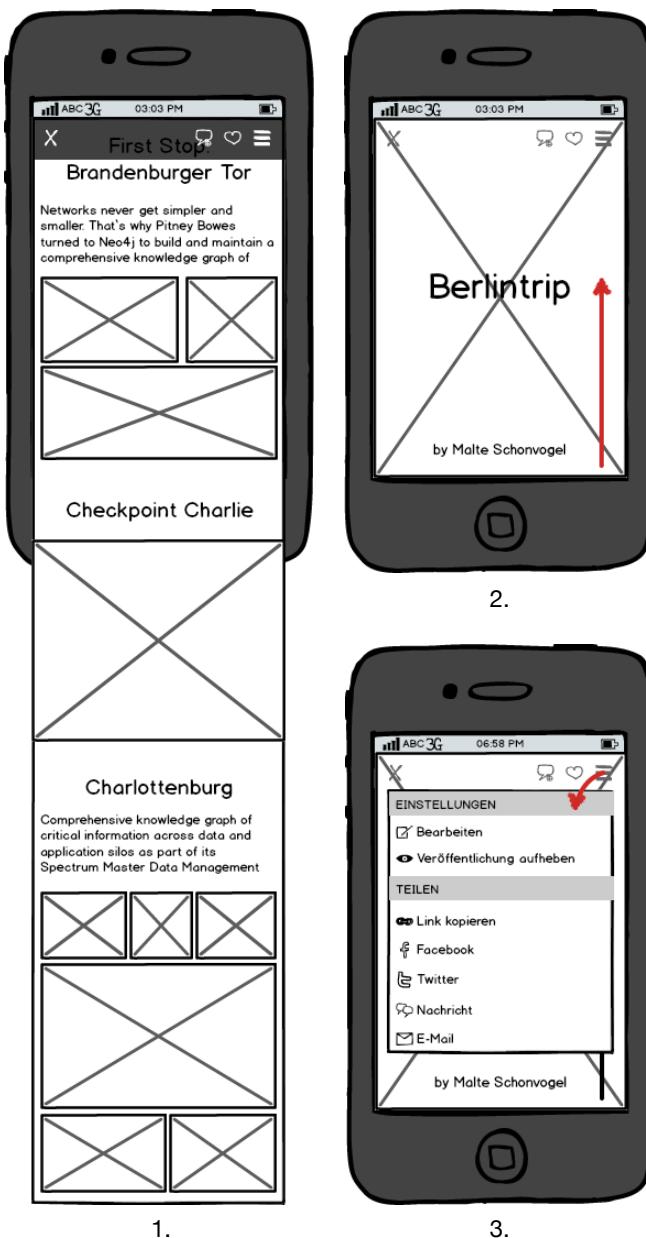


Abbildung 4.18 - Story Ansicht
Quelle: eigene Darstellung

Der zweite Bereich setzt sich aus den Stories zusammen, die der Profilnutzer schon veröffentlicht hat. In Abb. 4.16.2 wird das Nutzerprofil des aktuell eingelogten Nutzers dargestellt. Im Unterschied zu einem Profil eines anderen Nutzers, hat das "eigene" Profil statt des "Abonnieren"-Buttons (vgl. Abb. 4.17.1) einen Button zum bearbeiten des Profils (Abb. 4.16.2 [1]). Berührt der Anwender diesen, so wird die in Abb. 4.16.3 dargestellte View präsentiert. Hierbei handelt es sich um ein Formular anhand dessen der Nutzer seine zuvor getätigten Angaben ändern kann. Über den Button [1] in Abb. 4.16.3 gelangt der Nutzer zu der, in Abb. 4.16.4 dargestellten View, die es ihm, unter Angabe des aktuellen Passworts erlaubt, ein neues Passwort zu wählen. Berührt der Nutzer das Profilbild, so öffnet sich ein Kontextmenü (Vgl. Abb. 4.17.2), welches ihm erlaubt, mit der Kamera ein neues Foto zu machen, ein bestehendes Foto aus der Fotobibliothek des Geräts zu wählen oder das aktuell gesetzte Foto zu löschen. Die ersten zwei genannten Optionen rufen jeweils Standardsystemkomponenten zum Zugriff auf Kamera bzw. Fotobibliothek auf.

Umsetzung

Für die Implementierung der in Abbildung 4.16.1 dargestellten View ist aufgrund der Tabellendarstellung ein UITableViewController (vgl. Kapitel 2.11.1.2) vorgesehen. Die einzelnen Zellen der Tabelle sollen durch die Klasse ListUsersCellView dargestellt werden, welche von UITableViewCell erbt und die benötigten Komponenten enthält.

Für die View des Nutzerprofils (Vgl. Abb. 4.16.2) bietet sich die Verwendung eines UICollectionViewControllers (siehe Kapitel 2.11.1.1) an, um selbst bei vielen darzustellenden Stories ein performante Benutzeroberfläche umzusetzen. Der erste Bereich des Profils wird als Header View der ersten (und einzigen) Sektion der Collection View durch die Klasse ProfileHeaderView umgesetzt (Vgl. Abb. 2.11). Die Umsetzung der in Abb. 4.16.3 und 4.16.4 skizzierten Ansichten erfolgt analog zu den Formularen aus 4.4.2.2. Für die



Abbildung 4.19 - Story Ansicht
Quelle: eigene Darstellung

Umsetzung des in Abb. 4.17.6 dargestellten Kontextmenüs wird sich der Klasse `UIActionSheet` bedient, welche laut des `UIKit User Interface Catalog` für die Präsentation von mehreren Buttons, die alternative Auswahlmöglichkeiten bieten, konzipiert ist. Berührt der Nutzer den Button "Foto machen" oder "Foto auswählen", so wird die Standardkomponente des Systems zur Auswahl von Fotos, `UIImagePickerController`, dargestellt.

4.4.2.4 Story

Die Storyview ist das zentrale Element der Applikation und ist, wie das Nutzerprofil, eine Komponente, die an verschiedenen Stellen der App dynamisch eingebunden werden kann. Der Aufbau der Komponente lässt sich in drei Teile unterteilen. Der erste Bereich stellt das Storycover dar (Vgl. Abb. 4.19.2 & Def. Story) welches unabhängig von der Größe des Bildschirms stets bildschirmfüllend sein soll. Zentriert in der Mitte befindet sich der Titel der Story und am unteren Rand der Name des Autors. Berührt der Nutzer den Namen, wird die Profilview (Vgl. Abb. 4.18.5) angezeigt.

Den zweiten Bereich innerhalb der Komponente, stellt die am oberen Bildschirmrand fixierte Leiste (Abb. 4.19.1 [1]) dar und ermöglicht es dem Nutzer, die Komponente zu schließen, die Story zu kommentieren, sie mit einer "gefällt mir"-Angabe zu versehen und ein Kontextmenü zu öffnen, über das weitere Aktionen möglich sind (Vgl. Abb. 4.18.3). Dem Autor der Story ist es möglich, die Story zu bearbeiten und die Veröffentlichung aufzuheben. Andere Anwender können über dieses Menü die Story über verschiedene Kanäle teilen. Scrollt der Nutzer in Richtung des in Abb. 4.18.2 dargestellten Pfeils, so wird der dritte Bereich, der sich aus variabel vielen Absätzen (Sektionen) zusammensetzen kann, sichtbar. Jede Sektion besteht aus einem Titel, einem Fließtext und einem oder mehreren Fotos, wobei Titel und Fließtext für den Autor bei der Erstellung der Story optional sind (Vgl. Abb. 4.18.1). Möchte der Autor eine neue Sektion hinzufügen, so berührt er den Button [2] in Abb. 4.19.1. Ihm bietet nun ein Auswahlmenü, dargestellt in Abb. 4.19.1 [4], die Wahl zwischen zwei verschiedenen Layouts. Zum einen, ein Layout, das es ermöglicht, mehrere Fotos zu einer Sektion hinzuzufügen, die durch einen Lay-

outalgorithmus in einem flexiblen Raster angeordnet werden. Zum anderen ein Layout, welches die Auswahl eines Fotos zulässt und dies in Vollbreite des jeweiligen Displays anzeigt.

Hat sich der Autor für eine Layoutvariante entschieden, so öffnet sich die in Abb. 4.19.2 dargestellte View, die es dem Anwender ermöglicht, Bilder aus der Fotobibliothek des Geräts zu wählen. Sie stellt die Übersicht der Fotobibliothek dar und listet die einzelnen Fotoalben auf. Wählt der Nutzer ein Album, so wird ihm die in Abb. 4.19.3 dargestellte View präsentiert, in der Fotos durch einen Tap ausgewählt werden können. Die Leiste am unteren Bildschirmrand zeigt die Anzahl der ausgewählten und maximal auswählbaren Fotos an. Über den Button [1] in Abb. 4.19.3 gelangt der Nutzer zu der in Abb. 4.19.4 dargestellten View, in der es dem Nutzer ermöglicht wird, die zuvor ausgewählten Fotos zu bearbeiten. Diese View ist in mehrere Teilbereiche gegliedert. Der erste Bereich [1] beinhaltet die vom Nutzer ausgewählten Fotos. Das letzte Element innerhalb dieses Bereichs stellt dabei immer ein Button dar, der es ermöglicht, weitere Fotos auszuwählen. Der zweite Bereich [2] innerhalb der View bietet dem Anwender die Möglichkeit, ein Werkzeug auszuwählen, mit dem das aktuell in Teilbereich [1] ausgewählte und in Teilbereich drei [3] vergrößert dargestellte Foto, bearbeitet werden kann.

Für die Bearbeitung von Fotos sind zwei Werkzeuge vorgesehen (Vgl. Anforderung 7, Tabelle 3.4). Zum einen ein Werkzeug, mit dem es möglich ist, das ausgewählte Foto zuzuschneiden, zum anderen ein Filterwerkzeug, welches es dem Nutzer ermöglicht zwischen mehreren Filtern zu wählen und anzuwenden. Ist ein Werkzeug aktiv, so werden die Buttons in der Leiste (Vgl. Abb. 4.19.4. [1]) ausgeblendet und werkzeugspezifische Buttons eingeblendet (Vgl. Abb. 4.20.1 [1]).

Über den Button [4] in Abb. 4.19.4 kann der Bearbeitungs- und Auswahlprozess schließlich beendet, und die gewählten Fotos einer neuen Sektion in der Story hinzugefügt werden. Ist die Story im Bearbeitungsmodus, wurde also bei einer bestehenden Story im Kontextmenü der Punkt "Bearbeiten" gewählt, oder wird eine neue Story erstellt, so erscheinen über jeder Sektion eine Leiste mit Buttons (Vgl. Abb. 4.19.1, [5-7]), über die der Nutzer Aktionen ausführen kann. Die Buttons [5] ermöglichen es, die Sektionen zu sortieren, der Button [6] löscht die betreffende Sektion und der Button [7] öffnet den Bildauswahl dialog (Vgl. Abb. 4.19.2) und ermöglicht es dem Nutzer, weitere Bilder hinzuzufügen. Ferner schaltet ein aktiver Bearbeitungsmodus zahlreiche Gesten frei, die dem Nutzer zusätzliche Funktionen bieten und in Tabelle 4.7 beschrieben werden.

Die Wahl der Größe der einzelnen Fotozellen (*/items*) erfolgt über einen Algorithmus, welcher die Maße so berechnet, um die Bildschirmfläche optimal zu nutzen und so ein ausgeglichenes Raster ermöglicht. Der Nutzer hat, wie in Tabelle 4.7 beschrieben, die Möglichkeit auf den Algorithmus Einfluss zu nehmen.

Geste	Aktion
Wischgeste über eine Fotozelle (Item)	Ein Kontextmenü öffnet sich und bietet die Möglichkeit das Foto zu entfernen (Vgl. Abb. 4.20.2).
Doppel Tab auf eine Fotozelle (Item)	Die Fotozelle wird auf die Vollbreite des Rasters skaliert, die anderen Zellen passen sich dementsprechend an (Vgl. Abb. 4.19.1 [9]). Eine erneute Durchführung der Geste hebt diese Aktion wieder auf.
Langer Tab auf eine Fotozelle (Item)	Die Zelle folgt dem Finger des Nutzers und lässt sich so umpositionieren (Drag'n'Drop).
Bei aktiver Tastatur: Tab außerhalb eines Textfeldes	Beendet die Texteingabe und blendet die Tastatur aus.

Tabelle 4.7 - Mögliche Gesten in der Storyview
Quelle: eigene Darstellung

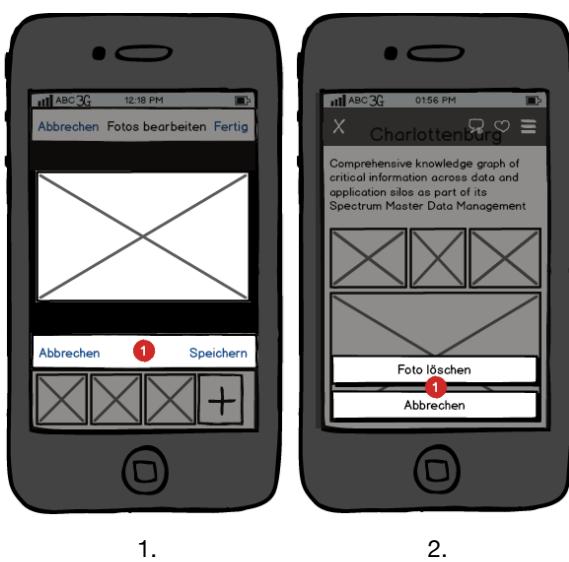


Abbildung 4.20 - Bild bearbeiten, Zelle löschen
Quelle: eigene Darstellung

Aufgrund der Tatsache, dass eine Story mitunter mehrere Dutzend Bilder beinhalten kann, bietet es sich an, als Grundlage für diese Komponente den *UICollectionView-Controller* zu verwenden. Zum einen wird durch dessen View-Recycling Mechanismus eine performante Benutzeroberfläche begünstigt und zum anderen ist er durch die Möglichkeit der Nutzung anwendungsspezifischer Layouts sehr anpassbar. Ferner hat die *UICollectionView* durch die Aufteilung des Layouts in *Sections* und *Items* (vgl. Abb. 2.11) die gleiche Struktur wie eine Story (vgl. Kapitel 3.1). Für die Umsetzung des ausgeglichenen Rasters wird sich dem Layout *NHBalancedFlowLayout* (vgl. Kapitel 4.4.1.7) bedient. Um die Anforderungen der Abbildung 4.19.1 an das Layout umzusetzen, muss es um einige Funktionen erweitert werden. Einerseits muss das Layout die Möglichkeit bieten, mehrere Ergänzende

Views darzustellen. Diese werden benötigt, um die Leiste mit Buttons über einer Sektion (*StorySectionHeaderView* - vgl. Abb. 4.19.1 [5-7]), die Eingabefelder für Sektionstitel und Fließtext (*StorySectionTextView*) sowie einen *Footer* als Platzhalter zwischen den Sektionen einzubinden. Ferner ist der im Layout genutzte Algorithmus so zu erweitern, dass nach Nutzereingabe einzelne Zellen die Vollbreite des Bildschirms einnehmen. Um die Darstellung des in Abbildung 4.18.2 dargestellten Covers (*StoryCoverView*) zu ermöglichen, also einer bildschirmfüllenden View, welche sich über der *UICollectionView* befindet, ist dem Layout der *UICollectionView* ein Offset hinzuzufügen, welches der Höhe des Bildschirms entspricht.

Für das in 4.19.1 [4] angezeigte Auswahlmenü ist die Komponente *GKPopoverController* (vgl. Kapitel 4.4.1.6) in Verbindung mit einem *UITableViewcontroller* für die Darstellung der einzelnen Optionen vorgesehen. Die Umsetzung der einzelnen Optionen zum Teilen der Story erfolgt durch die Nutzung von Systembibliotheken. Für die Optionen „Facebook“ und „Twitter“ wird die Klasse *SLComposeViewController* des Social Frameworks verwendet, für die Optionen „Nachricht“ und „E-Mail“ die Klasse *MFMessage- bzw. MFMailViewController* des MessageUI Frameworks.

Da die Systemkomponente zur Auswahl von Bildern aus der Fotogalerie in iOS, *UIImagePickerController*, keine Mehrfachauswahl unterstützt,¹¹⁵ muss diese Komponente nachimplementiert werden (vgl. Abb. 4.19.2-4.19.4). Als Datenquelle für diese Implementierung dient die Schnittstelle zur Fotogalerie, *ALAssetsLibrary*.¹¹⁶

Für die Umsetzung der Fotoalbenübersicht (*AssetPickerController* / *IndexViewController* - vgl. Abb. 4.19.2) ist, wie im Original, ein *UITableViewcontroller* und für die View der Albumansicht (*AssetsPickerController* / *AlbumViewController* - vgl. Abb. 4.19.3) ein *UICollectionViewController* vorgesehen. Um das Bearbeiten der Bilder zu ermöglichen ist zudem eine weitere Komponente (*EditViewController* - vgl. Abb. 4.19.4) nötig. Diese lässt sich unter Verwendung von drei Elementen realisieren: Einer *UICollectionView*, welche das zu bearbeitende Bild anzeigt, einer *UIToolbar*, die dem Nutzer Zugriff auf die Werkzeuge gibt und einer weiteren *UICollectionView*, die alle zuvor ausgewählten Bilder darstellt.

¹¹⁵ vgl. [APLDEV15]

¹¹⁶ vgl. [APLDEV16]

Wählt der Nutzer ein Werkzeug, so wird es über eine Fabrikklasse erzeugt und fügt der View des zu bearbeitenden Bildes entsprechende Komponenten hinzu, welche für das gewählte Werkzeug benötigt werden. Jedes Werkzeug implementiert das Protokoll *ImageEditorTool* und bietet somit eine einheitliche Schnittstelle.

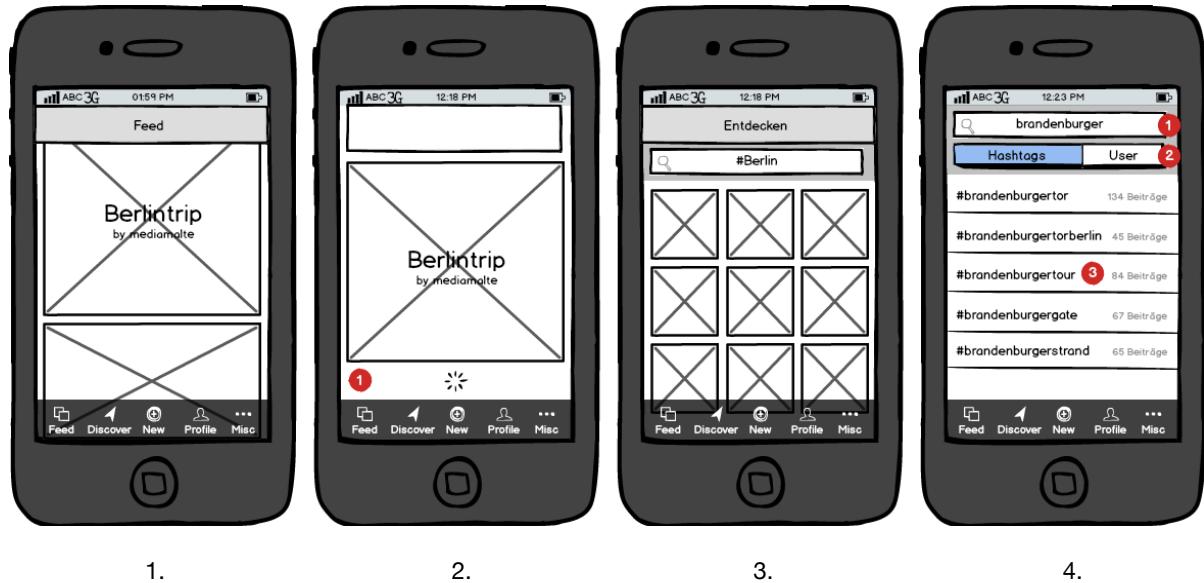


Abbildung 4.21 - „Feed“- und „Entdecken“-Ansicht
Quelle: eigene Darstellung

4.4.2.5 Feed

Die View „Feed“ ist die Einstiegsview für eingeloggte Anwender. Sie stellt chronologisch sortiert die Stories der User zusammen, denen der eingeloggte Anwender folgt. Die einzelnen Zellen (Vgl. Abb. 4.21.1) bestehen aus dem Storycover, dem Titel und dem Namen des Autors. Gelangt der Nutzer durch Scrollen an das untere Ende der Collection View, so soll diese automatisch neuen Inhalt nachladen sowie eine View anzeigen, die den Nutzer darüber informiert (vgl. Abb. 4.21.2 [1]). Bei Berührung einer Zelle öffnet sich die Komplettansicht (Vgl. Abb. 4.18.1) der jeweiligen Story.

Umsetzung

Die Umsetzung erfolgt mittels eines UICollectionViewControllers. Die Abmessung der Zellen orientiert sich an der Breite des Bildschirms. Um das automatische Nachladen weiterer Stories umzusetzen, soll der View-Recycling Mechanismus der CollectionView in Verbindung mit einer Footer View genutzt werden. Nähert sich die Footer View dem sichtbaren Bereich, so wird diese wiederverwendet, zeigt eine UIActivityIndicatorView an und löst den Aufruf der Schnittstelle aus.

4.4.2.6 Entdecken

Die View „Entdecken“ (Vgl. Abb. 4.21.3) besteht zum einen aus einer Suchmaske und zum anderen aus den zu dem Suchwort passenden Stories. Berührt der Anwender das Suchfeld, so wird die gesamte Suchmaske (vgl. Abb. 4.21.4), bestehend aus Suchfeld [1] und einer TabBar [2] eingeblendet und nimmt die Fläche der UINavigationBar ein. Über die TabBar hat der Nutzer die Möglichkeit festzulegen, ob er nach Hashtags oder Nutzern sucht. Außerdem werden bei einer aktiven Suchmaske alle Stories ausgeblendet und an dieser Stelle dem Suchwort ähnliche Vorschläge präsentiert [3]. Wurde keine Suche getätig, werden standardmäßig Stories chro-

nologisch angezeigt. Berührt der Anwender eine der Zellen, so öffnet sich die Komplettansicht (Vgl. Abb. 4.18.1) der jeweiligen Story.

Umsetzung

Die Umsetzung erfolgt anhand eines UITableViewControllers in Verbindung mit einer UISearchBar, einem UISearchDisplayController und einer UICollectionView welche standardmäßig den UITableViewController überlappt. Betätigt der Nutzer die Suche, so wird die UICollectionView ausgeblendet und in der UITableView Suchwortvorschläge angezeigt. Die Suchmaske bestehend aus Eingabefeld und TabBar lässt sich mit der UISearchBar umsetzen. Die Erweiterung der Suchmaske und die Verdrängung der UINavigationBar erfolgt durch den UISearchDisplayController.

4.4.3 Datenhaltung

Bei der Datenhaltung auf dem Endgerät kann zwischen sitzungsbasierter und persistenter Datenhaltung unterschieden werden. Sitzungsbasierte Daten bezeichnen jene Daten, welche die Applikation über die unterschiedlichen Schnittstellen erhält und in Objekten kapselt. Es ist nicht sinnvoll diese Daten persistent auf dem Gerät zu speichern, da sich diese Daten auf dem Server ändern können und sie somit nur für die aktuelle Sitzung gültig sind.

Unter persistenten Daten sind jene zu verstehen, die dauerhaft auf Gerät gespeichert werden sollen. Es handelt sich hier in der Regel um anwendungsspezifische Daten, wie Anwendungseinstellungen oder Daten die den fortlaufenden Login, auch nach Neustart, gewährleisten sollen.

4.4.3.1 Persistente Datenhaltung

Um Daten unabhängig von der Laufzeit des Systems zu speichern, stellt Apple mit *Core-Data* eine sehr mächtige Bibliothek bereit.¹¹⁷ Da es sich aber bei den Daten, die persistent auf dem Gerät gespeichert werden sollen, lediglich um den Datensatz des aktuell eingeloggten Nutzers handelt, wird an dieser Stelle NSUserDefaults verwendet. UserDefaults ermöglicht Key/Value Pair Speicherungen von Objekten, die dem NSCoder Protokoll entsprechen.¹¹⁸

4.4.3.2 Sitzungsbasierte Datenhaltung

Da sich die, über die REST-Schnittstelle angeforderten Daten jederzeit ändern können, werden diese Daten während der Sitzung in Objekten in der Klasse PersistenceManager gekapselt und über die Klasse FoyerApi bereitgestellt. Das folgende Diagramm gibt eine Übersicht über den Aufbau und Inhalt der Klassen:

¹¹⁷ vgl. [APLDEV13]

¹¹⁸ vgl. [APLDEV13]

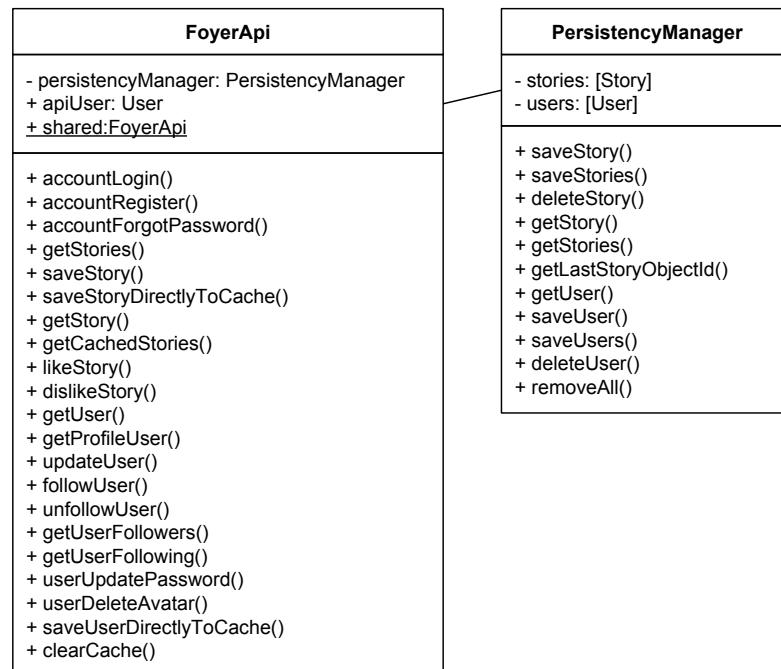


Abbildung 4.22 - Klassendiagramm FoyerApi

Quelle: eigene Darstellung

5 Realisierung des Prototypen

Auf Grundlage des in Kapitel 4 getätigten Entwurfs, erfolgte die prototypische Implementierung des Webservices und der iPhone Applikation. Dabei kamen verschiedene Werkzeuge und Bibliotheken zum Einsatz, die in diesem Kapitel erläutert werden. Ferner wird in diesem Kapitel auf ausgewählte Aspekte der Implementierung des Webservices und der iPhone Applikation eingegangen.

5.1 Werkzeuge & Entwicklungsumgebung

Für die Implementierung des Webservices und der iPhone App wurden verschiedene Werkzeuge verwendet. So wurde für die Implementierung des Webservices in der Programmiersprache JavaScript der Editor Sublime Text¹¹⁹ genutzt. Der Prozessmanager pm2 sorgt unter anderem dafür, dass bei einer Änderung des Codes des Webservices der Node.js Server automatisch neu gestartet wird und bietet zusätzlich eine komfortable Logging-Funktion.¹²⁰ Das Programm DHC ist ein Plugin für den Browser Google Chrome und ermöglicht es, HTTP / REST Anfragen an den Server zu stellen. Das Programm git dient der Versionskontrolle und ermöglicht es, die Entwicklung mehrerer Entwickler zu koordinieren, Quelltexte zentral auf einem Server zu verwalten und eine Historie über alle Veränderungen zu führen. Dies erlaubt es, Veränderungen nachvollziehen und auch auf alle älteren Zustände zurückgreifen zu können. Als Hosting-Dienst für das git Repository kommt GitHub¹²¹ zum Einsatz.

Für die Entwicklung der prototypischen iPhone App wurde die Entwicklungsumgebung Xcode in der Version 6.1.1 verwendet. Die Applikation wurde in der Programmiersprache Swift und teilweise in Objective-C programmiert.

Als mobile Endgeräte kamen ein iPhone 4 mit iOS 7.1 und ein iPhone 6 mit iOS 8.1 zum Einsatz. Die Kommunikation der Endgeräte zum Webservice wurde sowohl per WLAN als auch über das Mobilfunknetz realisiert. Bei der Implementierung der Datenhaltungsschicht (Kapitel 5.3) wird Neo4j in der Version 2.1.5 in der Community Edition eingesetzt.

5.2 Anwendungsschicht

Zunächst wird die Projektstruktur und die verwendeten Frameworks dargelegt und schließlich auf die Implementierung der einzelnen Schichten eingegangen.

5.2.3 Projektstruktur

Das Projekt wurde anhand von Ordnern und Node.js-Modulen in einzelne Komponenten unterteilt, um nach Funktionalitäten zu trennen und um Übersichtlichkeit zu gewährleisten. Der Aufbau des Webservices gliedert sich wie folgt (Ordner sind fettgedruckt):

¹¹⁹ <http://www.sublime.com>, abgerufen am 10.02.2015

¹²⁰ vgl. [PM201]

¹²¹ <http://www.github.com>, abgerufen am 10.02.2015

- **rest**
- **service**
- **persistence**
- config.js
- index.js
- **lib**
- **node_modules**
- **testing**
- Makefile
- package.json

Bei den Ordnern **rest**, **service** und **persistence** handelt es sich um die in Kapitel 4.3 beschriebenen Schichten der Anwendungsschicht. Der Ordner **lib** beinhaltet ein Modul, um die Routen der REST-Schicht dem Server hinzuzufügen. Im Ordner **testing** befinden sich Komponenten um die Applikation testbar zu machen und der Ordner **node_modules** ist ein automatisch vom Node.js-Paketmanager (NPM) erstelltes Verzeichnis, welches alle Abhängigkeiten beinhaltet. Die Datei config.js beinhaltet alle relevanten Konfigurationsdaten wie beispielsweise die Adresse des Datenbankservers oder das Schlüsselpaar für Amazon S3. Das Makefile dient ebenfalls der Testbarkeit der Anwendung und die Datei package.json ist eine vom Paketmanager konsumierte Datei, die die Abhängigkeiten der Applikation beinhaltet. Schlussendlich stellt die Datei index.js den Webserver dar.

5.2.4 Verwendete Frameworks und Bibliotheken

Name	Version	Funktion
koa	0.13.0	Web Framework
koa-response-time	1.0.2	Middleware für Koa.js, setzt den X-Response-Time HTTP-Response Header
koa-compress	1.0.8	Middleware für Koa.js, gzip Kompression der Antwort des Servers. Besonders bei Austauschformaten nützlich, da viel Redundanz enthalten ist.
koa-logger	1.2.2	Middleware für Koa.js, loggt Anfragen an den Server.
koa-validate	0.2.5	Middleware für Koa.js, validiert die Anfrage bzw. die Parameter der Anfrage aufgrund von vorgegebenen Regeln.
koa-body	0.4.0	Middleware für Koa.js, parst den Inhalt einer Anfrage.
koa-router	3.7.0	Middleware für Koa.js, ermöglicht es Routen hinzuzufügen.
seraph	0.9.10	Datenbank-Abstraktion für Neo4j
amazon-s3-url-signer	0.0.7	Modul um URLs zu privaten Amazon S3-Ressourcen zu signieren
bcrypt-nodejs	0.0.3	Node.js implementation der bcrypt Hashfunktion
shortid	2.1.3	Generator für nicht-sequentielle und eindeutige IDs.
shelljs	0.3.0	Modal das Unix shell Kommandos in Node.js ermöglicht.
node-getopt	0.2.3	Parser für Kommandozeilenargumente
mocha	2.0.1	Einfaches Test-Framework
should	4.4.1	Behavior Driven Development (BDD) Test-Framework
supertest	0.15.0	Framework zum Testen von HTTP-Servern

Tabelle 5.1 - Abhängigkeiten der Anwendungsschicht

Quelle: eigene Darstellung

5.2.5 REST-Schicht

Die REST-Schicht gliedert sich in die Ressourcen der Schnittstelle *account*, *default*, *stories* und *user*. Jede Ressource setzt sich aus einer Konfigurationsdatei (*config.json*), einem Node.js-Modul (*index.js*) und einer Datei zum Testen der Ressource (*test.js*) zusammen.

Die Konfigurationsdatei beinhaltet die unterschiedlichen Routen der Ressource. Die nachfolgende Abbildung zeigt eine beispielhafte Konfigurationsdatei:

```

1  {
2      "name": "Stories",
3      "description": "Routes for stories resource",
4      "routes": [
5          {
6              "method": "GET",
7              "path": "/stories/:objectId",
8              "public": false,
9              "action": "getStoryAction"
10         },
11         [...]
12     ]
13 }
```

Abbildung 5.1 - Konfigurationsdatei einer Ressource in der REST-Schicht
Quelle: eigene Darstellung

Die Hauptaufgabe der in Abbildung 5.1 dargestellten Konfigurationsdatei einer Ressource ist die Bereitstellung der Routen. Ein Routen-Objekt (Z.5-10) beinhaltet vier Schlüssel-Wert-Paare. Mit dem Schlüssel *method* wird die HTTP-Methode der Route festgelegt und mit dem Schlüssel *path* der Pfad, unter der die jeweilige Funktionalität aufrufbar sein soll. Der Pfad kann zudem Platzhalter für zu übergebene Parameter enthalten. Diese werden mit einem Doppelpunkt angeführt ("*:objectId*") und sind in der angegebenen Aktion (*action*) unter diesem Namen verfügbar. Der Schlüssel *public* stellt einen booleschen Wert dar, welcher festlegt, ob die Route auch für dem System unbekannte Nutzer aufrufbar ist. Ist dieser Wert *false*, so wird vor Aufruf der, in der Route angegebenen *action*, die Methode *auth* (Vgl. Abb. 5.2) aufgerufen.

```

1  exports.auth = function* (next)
2  {
3      var token = this.request.header['user-token'],
4          user = null;
5
6      if(token !== void 0) {
7          var userDbModel = require('../..../persistence/user');
8          user = yield userDbModel.getUserByToken(token);
9      }
10     if(user != null) {
11         this.user = user;
12         yield next;
13     } else {
14         this.status = 401;
15     }
16 }
```

Abbildung 5.2 - Middleware zur Autorisierung eines Requests mittels user-token
Quelle: eigene Darstellung

Diese Methode überprüft, ob im Anfrage-Objekt (*this.request*, Z.3, Abb. 5.2) der Header *user-token* gesetzt wurde. Ist dies der Fall, so wird der Nutzer über die Persistenz-Schicht abgefragt. Falls zu dem übergebenen Token ein Nutzer existiert, so wird zudem überprüft, ob dieser noch gültig ist, die ebenfalls in der Datenbank gespeicherte Lebenszeit des Tokens also noch nicht abgelaufen ist. Existiert ein Nutzer und ist der Token gültig, so wird das Objekt des Nutzers der Instanz hinzugefügt (Vgl. Z.11, Abb. 5.2) und die nächste Methode

aufgerufen (Vgl. Z.12, Abb. 5.2). Falls nicht, so wird die Anfrage mit dem Status-Code 401 als unautorisiert beendet.

Die Deklaration einer Funktion durch `function*` erzeugt eine Generatorfunktion.¹²² Bei Aufruf dieser Generatorfunktion wird der Code allerdings noch nicht ausgeführt, sondern ein Iteratorobjekt erzeugt. Erst bei Aufruf dessen `.next()`-Funktion wird die Generatorfunktion ausgeführt. Allerdings hält diese bei dem nächsten `yield`-Aufruf in der Generatorfunktion an und liefert ein Objekt mit dem Wert des Ausdrucks rechts neben dem `yield`-Befehl als Feld zurück. Erst, wenn der nächste `next()`-Aufruf von außen stattfindet, wird die Generatorfunktion weiter ausgeführt und hält gegebenenfalls beim nächsten `yield`-Befehl. Die Methode `next()` liefert ein Objekt zurück, welches den aktuellen `yield`-Wert enthält und ein boolsches Feld, das vermittelt, ob die Generatorfunktion komplett abgearbeitet wurde. Das `yield`-Schlüsselwort kann nur in einer Generatorfunktion benutzt werden. Dies lässt sich nutzen, um auf die Verwendung von Callbackmethoden im Kontext asynchroner Programmierung zu verzichten, da das Warten bereits in der Natur der Generatorfunktion liegt und asynchron ausgeführter Code sich dadurch wie synchroner Code schreiben lässt.¹²³ Diese Funktionalität wird durch das Konzept der Koroutinen beschrieben.

```

1 exports.getStoriesAction = function* (next)
2 {
3     this.checkQuery('lastStoryObjectId').optional()
4     this.checkQuery('amountPerPage').optional().isNumeric("Only numbers
5 accepted").toInt()
6     // x === void 0 is short for typeof(x) === 'undefined'
7     if(this.errors === void 0) {
8         var service = require('../service/stories/get');
9         var data = yield service.getStories(
10             this.user.id,
11             this.query.amountPerPage,
12             this.query.lastStoryObjectId
13         );
14
15         if(data.status > 199 && data.status < 300) {
16             this.body = data.response;
17         } else {
18             this.body = {
19                 message: data.message,
20                 status: data.status
21             };
22         }
23         this.status = data.status;
24     } else {
25         this.status = 422;
26         this.body = {
27             message: "There are some errors:",
28             errors: this.errors,
29             status: 422
30         }
31     }
32 }
```

Abbildung 5.3 - Exemplarische REST-Methode aus der Ressource „Stories“
Quelle: eigene Darstellung

Die in Abbildung 5.3 dargestellte Methode aus der Ressource Stories ermöglicht es, Stories sortiert nach Erscheinungsdatum abrufen. Über einen optionalen Query String lassen sich zwei Parameter, `lastStoryObjectId` und `amountPerPage`, setzen (Vgl. Tabelle 4.6), welche mit Hilfe der Bibliothek koa-validate (Vgl. Tabelle 5.1) validiert werden (Z.3ff). Treten bei der Validierung Fehler auf, so werden diese durch die Bibliothek in der Instanzvariable `errors` gespeichert und die Anfrage mit dem HTTP-Statuscode 422 unter Angabe der Fehler im

¹²² vgl. [MDN]

¹²³ vgl. [STRLO01]

Nachrichtenrumpf (Body) beantwortet (Z. 24-31). Sind keine Fehler vorhanden, so wird das betreffende Service-Modul referenziert und die entsprechende Methode im Service aufgerufen (Z.8f).

Die Service-Schicht gibt stets ein Objekt zurück, welches aus *response*, *message* und *status* besteht. Bei dem Attribut *response* handelt es sich um das eigentliche Ergebnis, das Attribut *message* enthält im Fall eines Fehlers eine Fehlerbeschreibung und das Attribut *status* enthält einen HTTP-Statuscode. Dies ermöglicht es, den Anwender der Schnittstelle auch aus der Service-Schicht heraus genau über die mögliche Fehlerursache zu informieren.

Befindet sich das Attribut *status* in einem akzeptablen Bereich(200-299), so wird das Ergebnis der Abfrage (*response*) in den Body geschrieben, andernfalls enthält der Body die Fehlerbeschreibung unter Angabe des Statuscodes.

Die durch den Query String übergebenen Parameter *lastObjectId* und *amountPerPage* ermöglichen die Umsetzung einer Paginierung. Um die nächste „Seite“ zu erhalten, muss der Nutzer der Schnittstelle die *objectId* der zuletzt abgefragten Story sowie die gewünschte Anzahl Stories übergeben. Wird kein Wert für *amountPerPage* übergeben, so werden standardmäßig 20 Einträge zurückgegeben.

5.2.6 Service-Schicht

Entsprechend der REST-Schicht ist die Service-Schicht anhand der Ressourcen *account*, *default*, *stories* und *user* organisiert. Nach Funktionalitäten in verschiedenen Modulen gekapselt, stellen die einzelnen Ressourcen Methoden bereit, um die Anfragen der REST-Schicht zu verarbeiten. Beispielsweise werden HTTP-Statuscodes gemäß der Geschäftslogik gesetzt und entsprechende Fehlermeldungen erzeugt. Diese Nachrichten können dann über den HTTP-Protokollstack zur Auswertung weitergereicht und auf der Clientseite ausgewertet werden. Die in Abbildung 5.5 abgebildete Service-Methode *getUser* der Ressource *user* stellt exemplarisch die Rolle der Service-Schicht als eine Vermittlerschicht zwischen REST- und Persistenz-Schicht dar. Sie kapselt damit die Zugriffe auf die Persistenz-Schicht (vgl. Z. 9f). Wie bereits in Kapitel 5.2.5 beschrieben liefert jede Methode der Service-Schicht ein Objekt zurück, welches aus *response*, *status* und *message* besteht (Z. 3-7).

```

1 exports.getUser = function* (currentUserId, userName)
2 {
3     var data = {
4         response: {},
5         status: 200,
6         message: null
7     };
8
9     var userDbModel = require('../..../persistence/user');
10    var response = yield userDbModel.getUserProfileByUserName(
11        currentUserId, userName);
12
13    if(Object.keys(response).length === 0) {
14        data.status = 404;
15        data.message = "User could not be found.";
16    } else {
17        data.response = response;
18    }
19    return data;
20 }

```

Abbildung 5.5 - Exemplarische Service-Methode aus der Ressource ,User'
Quelle: eigene Darstellung

5.2.7 Persistenz-Schicht

Die Persistenz-Schicht setzt sich aus zwei Modulen, *user* und *story*, zusammen, die den Zugriff auf die Datenhaltungsschicht (siehe Kapitel 4.2) kapseln. Diese stellen Methoden bereit, um die jeweiligen CRUD-Operationen durchzuführen. Dies wird anhand der folgenden Abbildung 5.6 beispielhaft erläutert.

```

1 exports.followUser = function* (userId, userName)
2 {
3     var async = require('thunkify');
4     var db = require('seraph')(_config.db.content.host);
5
6     var cypher = [
7         'MATCH (a:User), (b:User)',
8         'WHERE id(a) = {userId} AND b.userName = {userName}',
9         'MERGE (a)-[r:FOLLOWS]->(b)',
10        'ON CREATE SET',
11        '    a.followingCount = coalesce(a.followingCount, 0)+1,' ,
12        '    b.followersCount = coalesce(b.followersCount, 0)+1',
13        'RETURN r'
14    ].join(' ')
15
16    return yield async(db.query)(cypher, {
17        'userId': userId,
18        'userName': userName
19    });
20 }

```

Abbildung 5.5 - Exemplarische Persistenz-Methode aus der Entität ,User' zum abonnieren eines Nutzers
Quelle: eigene Darstellung

Die Datenbank wird in allen Methoden direkt über die Abfragesprache Cypher angesteuert. Dies lässt sich mit der Tatsache begründen, dass die verwendete Bibliothek (und viele andere Bibliotheken für Neo4j ebenfalls) nicht alle Operationen von Cypher in ihrer API gekapselt hat. Ferner ermöglicht die Nutzung der nativen Abfragesprache der Datenbank den einfachen Austausch der genutzten Bibliothek.

Ziel des Queries ist es, eine Beziehung mit dem Typ *FOLLOWERS* zwischen Nutzer (a) und Nutzer (b) zu erstellen, falls diese noch nicht existiert. Sofern sie erstellt wird, sollen zudem Zähler inkrementiert werden und falls nicht, so soll keine Aktion getätigter werden.

Der Query startet mit der Klausel *MATCH*, die es erlaubt Muster zu definieren, nach denen die Datenbank durchsucht wird (Vgl. Z.7). Die darauf folgende *WHERE* Klausel spezifiziert die Suche indem sie für jeden der Knoten eine eindeutige Identifizierungsmöglichkeit angibt (Vgl. Z.8). Zum einen die Datenbankinterne ID des

Knotens (a), zum anderen das Attribut `userName` des Knotens (b) welches als *UNIQUE* indiziert ist. Die Bezeichner in geschweiften Klammern (Z.8) stellen Platzhalter für einzusetzende Werte dar. Sofern beide Knoten vorhanden sind und diese noch nicht in der vorgeschriebenen Beziehung miteinander stehen, wird durch die Klausel *MERGE* die Beziehung unter Verwendung des Typs hergestellt (Z. 9). Ferner werden die jeweiligen Attribute, welche die Anzahl der Abonnenten bzw. Abonnierten Nutzer speichern inkrementiert (Z. 11f). Schließlich wird die Beziehung zurückgegeben.

Zeile 16ff. stellt den Aufruf der asynchronen Methode `query` des zuvor (Z. 4) erstellten db-Objekts dar und übergibt neben dem Query, als zweiten Parameter ein Objekt, welches die Platzhalter innerhalb des Queries definiert.

5.3 Datenhaltungs-Schicht

Die Komponenten der Datenhaltungs-Schicht realisieren die persistente Speicherung der zur Realisierung des Systems benötigten und in Kapitel 4.2.2 beschriebenen Daten. Der Zugriff der Anwendungsschicht auf die physische Datenbank Neo4j erfolgt über die REST-Schnittstelle.

5.3.1 Neo4j

5.3.1.1 Indizes und Labels

Um die Konsistenz der Daten sicherzustellen und die Geschwindigkeit der Abfragen zu erhöhen, kommt der Erstellung von Indizes eine zentrale Bedeutung zu. Anhand von Labels (optionale Bezeichner für Knoten) ist es möglich Attribute mit einem Index zu versehen bzw. Restriktionen (z.B. Eindeutigkeit) für das Datenmodell einzuführen. In Abbildung 5.7 werden so die Anforderungen aus Tabelle 4.1 und 4.2 umgesetzt.

```

1 CREATE CONSTRAINT ON (a:User) ASSERT a.email IS UNIQUE
2 CREATE CONSTRAINT ON (b:User) ASSERT b.userName IS UNIQUE
3 CREATE CONSTRAINT ON (c:User) ASSERT c.token IS UNIQUE
4 CREATE CONSTRAINT ON (d:Story) ASSERT d.objectId IS UNIQUE

```

Abbildung 5.7 - Erstellung der Indizes

Quelle: eigene Darstellung

5.3.1.2 Linked List

In Abbildung 5.8 ist ein einfacher Cypher Query dargestellt, welcher die Sortierfunktion von Neo4j und ein Limit nutzt, um die 20 neuesten Stories zu erhalten. Allerdings muss die Datenbank in dieser Variante erst alle Knoten sortieren, bevor die Limitierung greift. Folglich wird der Query bei steigender Storyzahl immer langsamer.

Wie bereits in Kapitel 4.2.2.1 angedeutet, ist die Möglichkeit eigene Datenstrukturen innerhalb des Graphs zu erstellen, eine mächtige Funktion einer Graph Datenbank. Durch Implementierung und Nutzung einer verketteten Liste (*linked list*) kann das gleiche Ergebnis mit weniger Overhead erzielt werden. Bei der Erstellung eines Story-Knoten wird dieser einfach an die verkettete Liste angehängt (Vgl. Abb. 5.10). Die Datenbank muss dann lediglich vom Ursprungsknoten dem Pfad *STORIES* bis zum 20. Knoten folgen, um das sortierte Ergebnis zu erhalten (Vgl. Abb. 5.9).

```

1 MATCH (story:Story)
2 RETURN story
3 ORDER BY story.createdAt DESC
4 LIMIT 20

```

Abbildung 5.8 - Abfrage der 20 neusten Stories ohne linked list

Quelle: eigene Darstellung

```

1 MATCH (root)-[:STORIES]->()-[:ALLNEXT*0..19]->(story:Story)
2 WHERE ID(root) = 0
3 RETURN collect(story)

```

Abbildung 5.9 - Abfrage der 20 neusten Stories anhand einer linked list

Quelle: eigene Darstellung

Die folgende Abbildung 5.10 fügt ein neues Element an die erste Stelle der linked-list, in der sich die Stories befinden:

```

1 MATCH (root), (newHead:Story {objectId: {objectId} }) WHERE id(root) = 0
2
3 CREATE UNIQUE (root)-[:STORIES]->(newHead)
4
5 WITH root, newHead
6 MATCH (newHead)<-[ :STORIES]-(root)-[oldRel1:STORIES]->(oldHead)
7 DELETE oldRel1
8 CREATE (newHead)-[:ALLNEXT]->(oldHead)

```

Abbildung 5.10 - Abfrage der 20 neusten Stories anhand einer linked list

Quelle: eigene Darstellung

Die erste Zeile weist dem Bezeichner *root* den Knoten mit dem Wert 0 als id zu. Der Wert von *objectId* des Knotens mit dem Bezeichner *newHead* wird parametrisch an den Query übergeben und steht an der Stelle *{objectId}*.

Die dritte Zeile erzeugt eine gerichtete Kante von *root* nach *newHead*. Danach existieren zwei erste Listenelemente. In Zeile sechs wird deshalb die alte Kante von *root* zum älteren ersten Listenelement mit dem Bezeichner *oldRel1* zugewiesen und in Zeile sieben gelöscht. Die Kette wird in der letzten Zeile durch eine Kante zwischen *newHead* und *oldHead* wieder zusammengefügt.

5.3.2 Objektspeicher

Die Einrichtung des Objektspeichers S3 erfolgt über eine Weboberfläche. Dort ist ein Name und eine Region (z.B. Irland, Tokyo, Frankfurt) für das zu erstellende Bucket festzulegen (siehe Abb. D.3).

Um nur bestimmten Anfragen den Zugriff auf die Objekte eines Buckets zu erlauben, gibt es bei Amazon S3 die so genannte Bucket Policy. Um dem, in Kapitel 4.2 beschriebenen Image-Proxy-Server den lesenden Zugriff (Abb. 5.11 Z. 7) zu gewähren, wird über eine Regel festgelegt, dass nur Anfragen der Zugriff erlaubt ist, dessen HTTP-User-Agent-Header die URL des genutzten Proxy-Servers beinhaltet. Der schreibende bzw. löschen Zugriff auf die Resource erfolgt, wie in Kapitel 4.2.3 beschrieben über zuvor serverseitig signierte URLs.

```

1 {
2     "Sid": "Allow images.wesurf.nl useragent",
3     "Effect": "Allow",
4     "Principal": {
5         "AWS": "*"
6     },
7     "Action": "s3:GetObject",
8     "Resource": "arn:aws:s3:::static.foyer.co/*",
9     "Condition": {
10         "StringLike": {
11             "aws:UserAgent": "*http://images.weserv.nl*"
12         }
13     }
14 }

```

Abbildung 5.11 - Amazon AWS S3 Bucket Policy

Quelle: eigene Darstellung

5.4 Präsentationsschicht

Der Ausgangspunkt einer jeden iOS-Applikation ist die Klasse *AppDelegate*, welche ein Delegateobjekt der Klasse *UIApplication* ist und über applikationsweite Statusänderungen (Applikation startet, geht in den Ruhe-modus, wird beendet, etc.) informiert wird. Sie ist eine Instanzvariable des Singletons *UIApplication* und somit kann in allen Klassen auf sie zugegriffen werden.

Die Views dieser Applikation werden programmatisch und nicht über den in Kapitel 2.9.1 erwähnten *Interface-builder* erstellt. Zum einen ist dies auf Gewohnheit zurückzuführen, zum anderen sprechen nachfolgende Gründe für dieses Verfahren:

- Treten Fehler in Views auf, so sind diese bei programmatisch erstellten Views durch die Entwicklerkonsole leichter zu identifizieren.
- Programmatisch erstellte Views sind performanter
- Arbeiten mehr als ein Entwickler an der Interfacedatei, so sind Konflikte durch Versionskontrollprogramme nur sehr zeitaufwändig zu beheben.
- Besteht dadurch die Möglichkeit größeren Einfluss auf das Erscheinungsbild zu nehmen, da der Interfacebuilder nur eine eingeschränkte Funktionalität bietet.

5.4.1 Projektstruktur

Um die Übersichtlichkeit zu gewährleisten werden die Dateien innerhalb des Xcode Projekts in mehreren Ordern nach Zusammengehörigkeit organisiert:

Ordner	Inhalt / Funktion
Api	Klassen zur Kommunikation mit der Schnittstelle
Models	Die Datenmodellklassen <i>Story</i> , <i>Section</i> , <i>Item</i> und <i>User</i>
Controller	Sämtliche ViewController der Applikation sowie deren View Klassen
Extensions	Klassenerweiterungen
AssetPicker	Der nachimplementierte ImagePickerController
Dependencies	Bibliotheken und Frameworks die das Projekt verwendet. Da Xcode kein Werkzeug für Dependency-Management bereitstellt und das verbreitete Dependency-Management-Tool „Cocoa Pods“ noch nicht für Swift verfügbar ist, werden die Abhängigkeit über Git-Submodule eingefügt und auf dem neusten Stand gehalten.
Images.xcassets	Der Asset-Katalog dient der Verwaltung von jeglichen Bildern, die Teil der Benutzeroberfläche sind.
Decoration	Views die einen rein dekorativen Zweck haben
Supporting Files	Ergänzende Dateien, wie Sprachdateien zur Internationalisierung

Tabelle 5.2 - Projektstruktur des Xcode Projekts

Quelle: eigene Darstellung

5.4.2 Kommunikation mit der Schnittstelle

Die Kommunikation mit der REST-API erfolgt in der Applikation über die Klasse *FoyerApi*, welche nach dem Singleton Entwurfsmuster umgesetzt ist und in der *AppDelegate* in einer Instanzvariable gespeichert wird, wodurch die Schnittstelle in jeder Klasse aufrufbar ist.

In der Programmiersprache Swift wird ein Singleton wie folgt konstruiert (vgl. Abb. 5.12):

```

1 class FoyerApi: NSObject
2 {
3     class var shared:FoyerApi {
4         struct Static {
5             static let instance = FoyerApi()
6         }
7         return Static.instance
8     }
9     private override init() {
10    {
11        super.init()
12    }
13 }
```

Abbildung 5.12 - Singleton Entwurfsmuster in Swift

Quelle: eigene Darstellung

Die Klassenvariable¹²⁴ *shared* (Z. 3) beinhaltet die Struktur (Z. 4) *Static*, welche die Instanz des Singletons als eine statische Konstante definiert. Der statische Teil dieses Statements sorgt dafür, dass das erstellte Objekt nur einmal existiert. Die Kapselung in dem *struct* ist notwendig, da im Gegensatz zu Strukturen, in Referenztypen keine statischen Variablen erlaubt sind. Durch das Zugriffsattribut *private* vor der *init*-Methode wird zudem verhindert, dass das Objekt ohne Verwendung der Singleton-Logik erzeugt werden kann.

Die Klasse referenziert über die Instanzvariable *apiUser* den aktuell eingeloggten Anwender der Applikation und liest ihn bei Applikationsstart, wie in Kapitel 4.4.3.1 beschrieben, aus den *NSUserDefaults* aus. Durch Nutzung eines *Property-Observers*¹²⁵ hält die Instanzvariable zudem die in *NSUserDefaults* gespeicherten Nutzerdaten stets auf dem neusten Stand. (vgl. Abb. 5.13)

```

1 var apiUser:User? {
2     didSet {
3         let ud = NSUserDefaults.standardUserDefaults()
4         ud.setObject(apiUser.serialize(), forKey: "apiUser")
5         ud.synchronize()
6     }
7 }
```

Abbildung 5.13 - Property-Observer

Quelle: eigene Darstellung

Die Klasse *PersistencyManager*, welche in der Klasse *FoyerApi* instanziert und als private Instanzvariable gehalten wird, hält die bereits über die Schnittstelle erhaltenen Nutzer und Stories und stellt Methoden bereit um diese Abzufragen oder zu löschen.

Jede Aktion der in Kapitel 4.3.1.3 beschriebenen API-Resourcen hat in der Klasse *FoyerApi* eine eigene Methode (vgl. Abb. 4.22). Beispielhaft ist eine solche Methode in Abbildung 5.13 dargestellt und soll im Folgenden beschrieben werden:

¹²⁴ Variable, die man aufrufen kann, ohne die Klasse instanzieren zu müssen

¹²⁵ vgl. [APLDEV16]

```

1 func getStories(refresh:Bool = false, completion: (([Story]?, NSError?) -> Void)) {
2 {
3     var lastStoryId = refresh ? nil : persistencyManager.getLastStoryObjectId()
4     Alamofire.manager.request(Router.StoriesRead( lastStoryId )).responseCollection() {
5         (_, _, stories:[Story]?, error) in
6         if refresh && error == nil {
7             self.persistencyManager.removeAll()
8         }
9         self.persistencyManager.saveStories(stories)
10        completion(stories, error)
11    }
12 }

```

Abbildung 5.14- Methode aus der Klasse *FoyerApi*
Quelle: eigene Darstellung

Jede Methode der Klasse *FoyerApi*, die mit dem der Schnittstelle kommuniziert wird asynchron ausgeführt und liefert das Ergebnis über eine Callback-Funktion zurück (Z.1). Der Parameter *refresh* gibt an, ob der Datenbestand des *PersistencyManagers* bei einer erfolgreichen Anfrage zurückgesetzt werden soll. In der Variable *lastStoryId* wird die ID der ältesten Story, die im *PersistencyManager* gehalten wird, gespeichert (Z.3) und so eine absatzweise Abfrage der Schnittstelle ermöglicht (Paginierung - vgl. Kapitel 5.2.5). Die eigentliche Abfrage der Schnittstelle findet ab Zeile vier statt. Hier wird sich der in Kapitel 4.4.1.2 eingeführten Netzwerk-Bibliothek *Alamofire* bedient, welche die Nutzung des Systemframeworks *NSURLSession* vereinfacht, zur Umsetzung von Best Practices verleitet und zudem die Nutzung von *Response Serialization* ermöglicht.

Der in Zeile vier genutzte und in Abbildung 5.15 dargestellte *Router* soll ein Beispiel für die Umsetzung einer Best Practice sein. Durch die Anwendung des *URLRequestConvertible* Protokolls lassen sich URL-Requests erstellen und ein typsicheres Routing umsetzen. In der Klasse wird das *Request*-Objekt sukzessive definiert. Die möglichen Routen sind in Zeile 9-12 definiert und legen zudem fest, welcher Parametertyp erwartet wird. In Zeile 15-22 wird anhand der gesetzten Route die zu nutzende HTTP-Methode gesetzt und ab Zeile 24 wird das eigentliche Request-Objekt erstellt. In den Zeilen 26-35 wird der Pfad der Route und eventuelle Parameter in einem Tupel gespeichert und in den folgenden Zeilen die zuvor gesetzten Variablen dem Request-Objekt hinzugefügt. Ferner werden dem Request-Objekt in den Zeilen 41-46 die von der REST-API zur Autorisierung benötigten HTTP-Header hinzugefügt (vgl. Kapitel 5.2.5). In den Zeilen 47-52 werden die Parameter von festgelegten Routen (*StoriesRead*) in einen Query-String formatiert und zurückgegeben.

Durch die Methode *responseCollection* (Abb. 5.14, Z. 4) wird ein zuvor definierter, generischer (Swift Generics) *Response Serializer* aufgerufen der automatisch und typsicher das *Response*-Objekt der aufgerufenen REST-Methode in ein Objekt angegebenen Typs (vgl. Abb. 5.14, Z. 5) serialisiert. Um dies zu ermöglichen implementieren die einzelnen Modellklassen eine *init*-Methode, welche als Parameter ein JSON-Dictionary¹²⁶ empfängt und die jeweiligen Instanzvariablen mit den zugehörigen Werten aus dem Dictionary befüllt.

Ferner prüft der *Response Serializer* anhand des HTTP-Statuscodes auf eventuelle Fehler und überführt ein mögliches *Error*-Objekt der REST-API in ein Cocoa *NSError* Objekt.

In Zeile 9 werden die von der Schnittstelle zurückgegebenen und durch den *Response Serializer* in serialisierten Story Objekte schlussendlichpersistiert und in Zeile 10 über den Aufruf der Callback-Funktion zurückgegeben.

¹²⁶ Assoziatives Array

```

1 enum Router: URLRequestConvertible
2 {
3     static let baseURLString = kApiBaseUrl
4     static var apiToken: String?
5     var userToken: String? {
6         return FoyerApi.shared.apiUser?.token
7     }
8
9     // — ROUTES (gekürzt)
10    case StoriesRead(String?)
11    case StoryLike(String)
12    case UserRead(String)
13
14    // — HTTP-METHOD (gekürzt)
15    var method: Method {
16        switch self {
17            case .StoriesRead, .UserRead:
18                return .GET
19            case .StoryLike:
20                return .PUT
21        }
22    }
23    // MARK: URLRequestConvertible
24    var URLRequest: NSURLRequest
25    {
26        let (path: String, parameters: [String: AnyObject]?) = {
27            switch self {
28                case .StoriesRead(let lastStoryId):
29                    return ("/stories", lastStoryId != nil ? ["lastObjectId":lastStoryId!] : nil)
30                case .StoryLike(let storyId):
31                    return ("/stories/\\"(storyId)/like", nil)
32                case .UserRead(let userName):
33                    return ("/user/\\"(userName)", nil)
34            }
35        }()
36
37        let URL = NSURL(string: Router.baseURLString)!
38        let request = NSMutableURLRequest(URL: URL.URLByAppendingPathComponent(path))
39        request = method.rawValue
40
41        if let token = Router.apiToken {
42            request.setValue(token, forHTTPHeaderField: "api-token")
43        }
44        if let token = userToken {
45            request.setValue(token, forHTTPHeaderField: "user-token")
46        }
47        switch self {
48            case .StoriesRead:
49                return ParameterEncoding.URL.encode(request, parameters: parameters).0
50            default:
51                return request
52        }
53    }
54}

```

Abbildung 5.15 - Router-Klasse innerhalb der Klasse FoyerApi

Quelle: eigene Darstellung

5.4.3 Umsetzung des User-Interfaces

In den folgenden Kapiteln soll die praktische Umsetzung des User-Interfaces anhand von zwei zentralen Elementen der Applikation in Ausschnitten beschrieben werden: dem Feed- und dem StoryViewController.

Wie einleitend in Kapitel 5.4 erwähnt, ist die Klasse *AppDelegate* der Ausgangspunkt einer jeden iOS-Applikation. Hier wird das Anwendungsfenster initialisiert und der Ursprungs Viewcontroller gesetzt. Letzterer wird in dieser Applikation durch die Klasse *RootViewController* repräsentiert, erbt von der Klasse *UITabBarController* und ist somit, wie in Kapitel 4.4.2.1 beschrieben, das zentrale Navigationselement der Anwendung. Ferner prüft er, ob der aktuell eingeloggte Nutzer gesetzt ist (siehe 5.4.2) und stellt, falls nicht, den *StartViewController* dar, welcher es dem Nutzer ermöglicht, sich einzuloggen oder zu registrieren (vgl. Kapitel 4.4.2.2).

5.4.3.1 FeedViewController

Für Nutzer, die bereits eingeloggt sind, ist der FeedViewController der Startpunkt der Applikation und sorgt bei erstmaliger Anzeige dafür, dass der Datenbestand der Applikation aktualisiert wird. Anhand von Ausschnitten des FeedViewControllers wird im Folgenden die Implementierung des UICollectionViewControllers erläutert.

Die in Abbildung 5.16 dargestellten Methoden des *FeedViewControllers* stellen die zur Anzeige einer UICollectionView zu implementierenden Kernmethoden der UICollectionView Datenquelle (*UICollectionViewDataSource*) dar, definieren die Struktur der CollectionView und stellen die Inhalte bereit.

```

1 override func numberOfSectionsInCollectionView(cv: UICollectionView) -> Int
2 {
3     return 1
4 }
5
6 override func collectionView(cv: UICollectionView, numberOfItemsInSection section:
7 Int) -> Int
8 {
9     return data.count
10 }
11
12 override func collectionView(collectionView: UICollectionView, cellForItemAtIndexPath
13 indexPath: NSIndexPath) -> UICollectionViewCell
14 {
15     let story = data[indexPath.item]
16     let cell = collectionView.dequeueReusableCellWithReuseIdentifier(kReuseIdentifier,
17 forIndexPath: indexPath) as FeedCell
18     cell.bind(story)
19
20     CircleActivityIndicator.setOnView(cell, size:kActivityIndicatorSize, animated: true)
21     UIImage.cachedImageWithItem(story.cover) {
22         (image: UIImage?, error: NSError?) in
23         CircleActivityIndicator.hideFromView(cell, animated: true)
24         if error == nil && image != nil {
25             if let cell = collectionView.cellForItemAtIndexPath(indexPath) as? FeedCell {
26                 cell.imageView.image = image
27             }
28         }
29     }
30     return cell
31 }
```

Abbildung 5.16 - Kernmethoden der Datenquelle

Quelle: eigene Darstellung

Durch die Methode *numberOfSectionsInCollectionView* (Z. 1-4) wird festgelegt, in wie viele Abschnitte die Collection View unterteilt wird. Die Methode *numberOfItemsInSection* (Z. 6-10) gibt die Anzahl der Zellen (Items) pro Abschnitt (Section) zurück, was in diesem Fall der Anzahl der Zellen der gesamten Collection View entspricht. Die Methode *cellForItemAtIndexPath* (Z. 12-31) gibt eine Zelle an die aufrufende Methode zurück. Der Parameter *indexPath* der Methode beschreibt den Pfad durch einen Datenbaum zu einem bestimmten Knoten, also zu einem bestimmten Abschnitt und einer bestimmten Zelle.¹²⁷ Der erste Schritt in dieser Methode ist die Abfrage des Datenobjekts um die zur aktuellen Zelle passenden Story zu erhalten (Z. 15). In Zeile 16 wird der Konstante *cell* eine wiederverwendete Zelle zugewiesen (vgl. Kapitel 2.11.1.1) und mit dem Story Objekt mittels der Methode *bind* (vgl. Abb. 5.17 Z. 43-39) befüllt. Der *ReuseIdentifier* bietet die Möglichkeit in der Collection View verschiedene Arten von Zellen zu verwenden. Da in dieser Collection View alle Zellen dem gleichen Typ entsprechen, wird hier eine zuvor definierte Konstante verwendet (*kReuseIdentifier*, Z. 16). In den Zeilen 20-29 wird der Zelle das Story Cover als Hintergrundbild hinzugefügt. Da dies über eine Callback Methode und somit asynchron erfolgt, wird dem Anwender durch eine hinzugefügte *ActivityIndicatorView*¹²⁸ signalisiert, dass

¹²⁷ vgl. [APLDEV17]

¹²⁸ Ein *UIActivityIndicatorView* signalisiert dem Nutzer, dass ein Vorgang in Bearbeitung ist

Inhalt geladen wird (Z. 20). Die Methode `cachedImageWithItem` (Z. 21) wurde über eine *Extension*¹²⁹ der Klasse `UIImage` hinzugefügt und erhält ein Objekt vom Typ `Item` anhand dessen das Bild entweder unter Verwendung der Bibliothek *Haneke* (siehe Kapitel 4.4.1.1) aus dem Cache oder über eine URL geladen wird. Ist dies erfolgt, so wird der `ActivityIndicator` entfernt (Z. 23) und das Bild der entsprechenden View der Zelle zugewiesen. Von besonderer Bedeutung ist es an dieser Stelle, die Zelle in der Callback Methode mittels des Aufrufs der Methode `cellForItemAtIndexPath` am `collectionView` Objekt erneut abzufragen. Sofern sich die angeforderte Zelle im sichtbaren Bereich des Bildschirms befindet, gibt die Methode die Zelle zurück, andernfalls „nil“¹³⁰. Diese Vorgehensweise ist dem View-Recycling geschuldet, da sonst durch weiteres Scrollen des Anwenders die Zelle nach Abschluss des Ladevorgangs schon längst aus dem Sichtbereich des Anwenders gelangt sein könnte und somit das nun fertig geladene Bild einer falschen Zelle zugeordnet werden würde.

```

1 class FeedCell: ParallaxTableViewCell {
2     let kPaddingCell: CGFloat = 10.0
3     lazy var title: UILabel = {
4         let label = UILabel()
5         label.numberOfLines = 0
6         label.lineBreakMode = NSLineBreakMode.ByWordWrapping
7         label.textAlignment = NSTextAlignment.Center
8         label.setTranslatesAutoresizingMaskIntoConstraints(false)
9         label.font = UIFont.boldSystemFontOfSize(28.0)
10        label.textColor = UIColor.whiteColor()
11        label.preferredMaxLayoutWidth = self.bounds.width-2*self.kPaddingCell
12        return label
13    }()
14    [...]
15
16    override init(frame: CGRect) {
17        super.init(frame: frame)
18        addSubview(title)
19        addSubview(author)
20        addConstraints()
21    }
22
23    func addConstraints() {
24        addConstraint(NSLayoutConstraint(
25            item: title,
26            attribute:NSLayoutAttribute.centerY,
27            relatedBy:NSLayoutRelation.equal,
28            toItem:self,
29            attribute:NSLayoutAttribute.centerY,
30            multiplier:1.0,
31            constant:0.0
32        ))
33        [...]
34        addConstraints(NSLayoutConstraint.constraintsWithVisualFormat(
35            "V:[title]-padding-[author]",
36            options: nil,
37            metrics: ["padding":kPaddingCell],
38            views: ["author":author, "title": title]
39        ))
40    }
41
42    func bind(story: Story?) {
43        imageView.image = nil
44        if let _story = story {
45            title.text = _story.title
46            author.text = _story.author?.userName
47        }
48    }
49}

```

Abbildung 5.17 - Zelle der `UICollectionView` des `FeedViewControllers`
Quelle: eigene Darstellung

¹²⁹ Extensions fügen einem Typ neue Funktionalitäten hinzu, können aber keine bestehende überschreiben

¹³⁰ „nil“ entspricht einem nicht-existenten Wert

Die in Abbildung 5.17 (verkürzt) dargestellte Klasse stellt die in der CollectionView des *FeedViewControllers* verwendete Zelle dar und steht exemplarisch für einen Großteil der restlichen Views, da viele nach dem gleichen Schema aufgebaut sind.

Der erste Teil der Klasse (Z. 3-14) legt die beinhalteten Subviews fest. Dies erfolgt anhand von Lazy Stored Properties, dessen initiale Werte erst berechnet werden, wenn Sie das erste mal verwendet werden.¹³¹ In Zeile drei wird der Titel einer Story als UILabel umgesetzt und legt in den folgenden Zeilen das Erscheinungsbild fest. Durch setzen des Attributes *numberOfLines* auf den Wert 0 und *preferredMaxLayoutWidth* wird erreicht, dass die Höhe des Labels automatisch abhängig vom Inhalt berechnet wird.

Im nächsten Teil der Klasse werden die zuvor definierten Views durch die Methode *addSubview* der *contentView* der Klasse hinzugefügt. Durch Aufruf der Methode *addConstraints*, welche in den Zeilen 23-41 definiert ist, werden die zuvor der View hinzugefügten *subviews* durch *NSLayoutConstraint*s (Restriktionen) positioniert. *NSLayoutConstraint*s sind Teil von Apples Autolayoutsystem und legen Größen, Abstände und Beziehungen von Views fest. Dies ermöglicht es, Restriktionen zu definieren, die eine vielseitige und dynamische Oberfläche erstellen, welche sich automatisch an Änderungen, wie beispielsweise der Bildschirmgröße anpasst.

Die in Zeile 24 hinzugefügte Restriktion positioniert die View „title“ (Z.25, *item*) in Relation zur View der Zelle (Z. 28, *toItem*). In den Zeilen 26, 27 und 29 wird festgelegt, dass die Y-Koordinate des Mittelpunkts der „title“ View dem der Zelle entsprechen soll, der Titel der Story also vertikal zentriert ausgerichtet wird. Über die Klassenmethode *constraintsWithVisualFormat* (Z. 34) lassen sich relativ bequem gleich mehrere Restriktionen auf einmal erstellen. Die Anordnung der einzelnen Views untereinander wird hier über eine ASCII-art Zeichenkette (vgl. Z. 35) festgelegt.

Die in Abbildung 5.18 dargestellte Delegate-Methode des *FeedViewControllers* wird aufgerufen, sobald der Anwender eine Zelle berührt. Dann wird der *StoryViewController* mit der entsprechenden Story initialisiert (Z. 3) und durch den *UINavigationController* präsentiert (Z. 4).

```

1 override func collectionView(collectionView: UICollectionView, didSelectItemAtIndexPath:
2     Path indexPath: NSIndexPath) {
3     let vc = StoryViewController(story: &data[indexPath.item])
4     navigationController?.pushViewController(vc, animated: true)
5 }
```

Abbildung 5.18 - Delegate-Methode des *FeedViewControllers*
Quelle: eigene Darstellung

5.4.3.2 StoryViewController

Der *StoryViewController* basiert ebenfalls auf der Klasse *UICollectionViewController* und implementiert, wie in Kapitel 4.4.2.4 festgelegt, die Layoutklasse *NHBalancedFlowLayout*. Da diese in der Programmiersprache Objective-C umgesetzt ist, die Applikation jedoch in Swift programmiert wird, gilt es zunächst einen „Bridging-Header“ zu erstellen. Dieser ermöglicht die Koexistenz von Objective-C und Swift Code innerhalb eines Projekts. Um eine Objective-C Klasse in Swift zu verwenden, muss die Headerdatei der Objective-C Klasse dem Bridging-Header anhand einer Importanweisung hinzugefügt werden. Umgekehrt ist es ebenfalls möglich Swift-Code in Objective-C Klassen zu verwenden.

Während des Layoutprozesses ruft die Collection View am Layoutobjekt bestimmte Methoden auf, die es zu erweitern galt, um die im Entwurf festgelegten Anforderungen (z.B. Titel- und Freitexteingabefeld, Storycover, Werkzeugeiste, etc.) zu erfüllen (vgl. Kapitel 4.4.2.4 - Umsetzung):

¹³¹ vgl. [APLDEV18]

1. **prepareLayout**

In der `prepareLayout` Methode werden die nötigen Berechnungen angestellt, um die Positionen von Zellen und den weiteren Views innerhalb des Layouts im Voraus festzulegen (siehe Abb. 5.19).

2. **collectionViewContentSize**

Basierend auf den zuvor getätigten Berechnungen, gibt diese Methode die Größe der benötigten Fläche (Breite und Höhe als `CGSize`-Struktur) zurück, um die gesamte Collection View darzustellen. Diese Größe wird von der Collection View verwendet um die Scroll View entsprechend zu konfigurieren.

3. **layoutAttributesForElementsInRect:**

Diese Methode wird basierend auf der aktuellen Scroll-Position von der Collection View aufgerufen, um die Layoutattribute für die Zellen und Views anzufordern, die sich in einem bestimmten Rechteck (Bildschirmausschnitt) befinden.

```

1 - (void)prepareLayout
2 {
3     [super prepareLayout];
4     [...]
5     // Offset for header image
6     if(self.storyCoverHeight){
7         contentSize.height = self.storyCoverHeight;
8     }
9
10    for (int section = 0; section < [self.collectionView numberOfSections]; section++) {
11        [...]
12        CGSize sectionSize = CGSizeMakeZero;
13        // --- set header size
14        CGSize headerSize = [self referenceSizeForHeaderInSection:section];
15        CGRect headerFrame = CGRectMake(0, contentSize.height, CGRectGetWidth(self.collectionView.bounds), headerSize.height);
16        [headerFrames addObject:[NSValue valueWithCGRect:headerFrame]];
17        // --- set header end
18
19        // --- set title size
20        CGSize titleSize = [self referenceSizeForTitleInSection:section];
21        CGRect titleFrame = CGRectMake(0, contentSize.height + headerSize.height, CGRectGetWidth(self.collectionView.bounds), titleSize.height);
22        [titleFrames addObject:[NSValue valueWithCGRect:titleFrame]];
23        // --- set title end
24
25        // --- set sectionSize
26        [...]
27        contentSize = CGSizeMake(sectionSize.width, contentSize.height + headerSize.height
28        + titleSize.height + textSize.height + sectionSize.height + footerSize.height);
29    }
30}
31
32 }
```

Abbildung 5.19 - `prepareLayout` Methode des `UICollectionViewLayouts`

Quelle: eigene Darstellung

Die, in Abbildung 5.19 verkürzt dargestellte Methode `prepareLayout` sowie ihre Helper-Methoden stellen die eigentliche Logik des Layouts dar und berechnen dieses im Voraus. Zu diesem Zweck wird durch die Sektionen der Collection View interiert (Z. 10) und die Abmessungen der einzelnen Views über zahlreiche Delegate-Methoden abgefragt (Z. 14, 22) sowie sukzessive addiert.

Beispielsweise wird in Zeile 22 über eine Helper-Methode die Größe der View abgefragt, welche das Eingabefeld für den Titel einer Sektion beinhaltet. Diese Helfermethode überprüft, ob die Collection View die optionale Delegate-Methode `referenceSizeForTitleInSection` implementiert. Falls ja, so wird diese aufgerufen, falls nicht wird ein zuvor gesetzter Standardwert zurückgegeben. Innerhalb des `CollectionViewControllers` berechnet diese Methode dann die benötigte Fläche (`CGSize`) anhand der Länge des jeweiligen Titels (Abb. 5.20 Z. 11) und gibt diese zurück. Dadurch dass Schriftgröße sowie Zeilenabstand für die Berechnung von Bedeutung sind, initial-

isiert die Methode die Text-View (Abb. 5.20, Z. 4) und lässt diese die Größe berechnen (Abb. 5.20, Z. 11). Um die Positionen der View sowie deren Abmessungen zu speichern, wird sich der Struktur `CGRect` bedient, welche aus einem Ursprung (`CGPoint`, x und y) sowie einer Größe (`CGSize`, Breite und Höhe) besteht und durch die Funktion `CGRectMake` (Z. 22) anhand von vier Parametern in der Reihenfolge: x, y, Breite und Höhe erstellt wird. Das Rechteck zur Positionierung der Titel View startet am linken Bildschirmrand (x = 0), befindet sich unter dem zuvor definierten Header (Gesamthöhe der letzten Interaktionen, addiert mit Höhe des Headers), ist so breit wie das Fenster und nutzt die zuvor berechnete Höhe.

```

1 func collectionView(collectionView: UICollectionView!, layout collectionViewLayout:
2 UICollectionViewLayout!, referenceSizeForTitleInSection section: Int) -> CGSize
3 {
4     let view = StorySectionTextView(frame: CGRectMake(0, 0, collectionView.bounds.width,
5 10.0))
6     view.bind(BalancedLayoutTitleIdentifier, text: story.sections[section].title, index-
7 Path: NSIndexPath(forItem: 0, inSection: section))
8     if editmodeEnabled {
9         view.editModeEnabled()
10    }
11    return view.calcSize(story.sections[section].title ?? "")
12 }
```

Abbildung 5.20 - Delegate-Methode `referenceSizeForTitleInSection` innerhalb der Collection View

Quelle: eigene Darstellung

Die Aufgabe der einleitend beschriebenen Methoden `collectionViewContentSize` und `layoutAttributesForElementsInRect` besteht lediglich darin, die durch die Methode `prepareLayout` angestellten Berechnungen auszugeben bzw. nach Bereich zu filtern und dann auszugeben. Um dies durchzuführen, wird in der Funktion `layoutAttributesForElementsInRect:` (siehe Abb. 5.21) durch alle Sektionen der Collection View iteriert (Z. 5) und überprüft, ob sich die jeweilige View mit dem parametrisch übergebenen Rechteck schneidet (Z. 13f). Ist dies der Fall, so wird dies im Feld `layoutAttributes` festgehalten (Z. 15) und schließlich durch die `return`-Anweisung zurückgegeben (Z. 19).

```

1 - (NSArray *)layoutAttributesForElementsInRect:(CGRect)rect {
2     NSMutableArray *layoutAttributes = [NSMutableArray array];
3     NSInteger numberOfSections = [self.collectionView numberOfSections];
4
5     for (NSInteger section = 0; section < numberOfSections; section++) {
6         NSIndexPath *sectionIndexPath = [NSIndexPath indexPathForItem:0
7             inSection:section];
8
9         UICollectionViewLayoutAttributes *headerAttributes = [self
10 layoutAttributesForSupplementaryViewOfKind:UICollectionViewElementKindSectionHeader
11 atIndexPath:sectionIndexPath];
12
13         if (! CGSizeEqualToString(headerAttributes.frame.size, CGSizeZero) &&
14             CGRectIntersectsRect(headerAttributes.frame, rect)) {
15             [layoutAttributes addObject:headerAttributes];
16         }
17         [...]
18     }
19     return layoutAttributes;
20 }
```

Abbildung 5.21 - `layoutAttributesForElementsInRect` Methode des `UICollectionViewLayouts`

Quelle: eigene Darstellung

5.4.4 Internationalisierung

Die Internationalisierung, gerne auch mit dem Numeronym **i18n** (die 18 steht für die Anzahl der Buchstaben zwischen i und n im englischen Wort „Internationalization“) abgekürzt, befasst sich unter anderem mit der Übersetzung der Texte der Anwendung.¹³² An diversen Stellen im Quellcode wurden Zeichenketten fest verankert. Um die Wartbarkeit zu gewährleisten und damit eine Internationalisierung der Applikation möglich ist, wurden Sprachschlüssel verwendet und die entsprechenden Übersetzungen in Sprachdateien ausgelagert. Hierzu wurde in der Xcode-Projektdatei eine weitere Localisation angelegt und die entsprechenden *.strings Sprachdatein verknüpft. Um im Quellcode auf die richtige Variante des Sprachschlüssels zuzugreifen wird die Funktion *NSLocalizedString* verwendet. Sie erhält als ersten Parameter den Schlüssel und als zweiten einen Kommentar. (vgl. Abb. 5.15):

```
1 NSLocalizedString("story.edit.button.addsection", comment:"")
```

Abbildung 5.18 - Kennzeichnung einer zu übersetzenden Zeichenkette
Quelle: eigene Darstellung

4.4.5 Probleme und Schwierigkeiten

Die im Kapitel 5.4.3.2 (StoryViewController) beschriebene Layout-Klasse *NHBalancedFlowLayout* wurde zu Beginn der Arbeit in Swift umgesetzt (~400 LoC, *BalancedLayout.swift*) und zeigt alle Stories korrekt an. Die letztendlich verwendete und in Objective-C programmierte Klasse sollte ursprünglich nur als Beispiel und Vorlage dienen. Allerdings fiel durch intensives Testen auf, dass durch einen häufigen Wechsel zwischen Eingabefeldern bedingt, die Applikation abstürzt. Durch das in Xcode integrierte Profiling-Werkzeug „Instruments“ konnte ein ZombieObjekt identifiziert werden.

Zombieobjekte sind Objekte, die, obwohl sie bereits deallokiert wurden, weiter Nachrichten empfangen und dadurch die Applikation zum Absturz bringen. Um diese Objekte zu identifizieren bietet Xcode die Möglichkeit eine Build-Einstellung namens *NSZombies* zu setzen, welche dafür sorgt, dass Objekte nicht freigegeben werden, wenn ihr Referenzzähler auf 0 fällt. Stattdessen werden diese in Objekte vom Typ *NSZombie* umgewandelt, welche jeden Aufruf einer Instanz von ihr zu protokollieren, da dies bedeutet, dass der Code versucht, eine Methode eines nicht mehr existenten Objekts aufzurufen.

Die Nutzung dieser Methode lieferte allerdings immer unterschiedliche Zombiobjekte aus internen Cocoa Klassen. Auch in einem Swift-Meetup¹³³ konnten andere Entwickler das eigentliche Problem nicht rekonstruieren. Es besteht die Möglichkeit, dass es sich um ein Swift-spezifisches Problem handelt.

Ein weiterer Fehler, welcher zum Absturz der Applikation mit einer unverständlichen Fehlerausgabe führte ist der Aufruf der Methode *insertItemsAtIndexPaths* an einer leeren *Collection View*. Diese Methode wird aufgerufen um der Collection View mitzuteilen, dass dem Datenobjekt eine neue Sektion hinzugefügt wurde. Der Fehler lässt sich durch eine Prüfung der Anzahl der Sektionen (Abb. 5.23, Z. 2) vermeiden. Ist die einzufügende Sektion die Erste, so wird anhand der Methode *reloadData* die Collection View erneut geladen (Z. 10). Andernfalls wird die Sektion mittels der Methode *performBatchUpdates* unter Verwendung einer Animation eingefügt.

¹³² vgl. [KOEDI] S. 256

¹³³ Meetup mit dem Titel „swift.berlin #6“ vom 26. Januar 2015 bei der wimdu GmbH

```
1 [...]  
2 if self.story.sections.count > 1 {  
3     self.collectionView!.performBatchUpdates({  
4         self.collectionView!.insertSections(NSIndexSet(index: sectionId))  
5     }) { (_) in  
6         self.collectionView!.scrollToItem(atIndexPath(NSIndexPath(forItem: 0, inSection:  
7             sectionId), atScrollPosition:.Top, animated: true)  
8     }  
9 } else {  
10    self.collectionView!.reloadData()  
11    self.collectionView!.scrollToItem(atIndexPath(NSIndexPath(forItem: 0, inSection:  
12        sectionId), atScrollPosition:.Top, animated: true)  
13 }  
14 [...]
```

Abbildung 5.23 - Fehler bei Aufruf von `insertItemsAtIndexPaths` an leerer Collection View

Quelle: eigene Darstellung

6 Tests

Tests sind während und besonders gegen Ende der Entwicklung einer Software unerlässlich, um Fehler aufzudecken und die Qualität des Endproduktes zu steigern. In diesem Kapitel werden die Maßnahmen erläutert, die unternommen wurden, um eine korrekte Funktionalität zu gewährleisten. Es werden die zum Testen verwendeten Bibliotheken und Werkzeuge vorgestellt sowie die Testinfrastruktur erläutert.

6.1 Webservice

Um die Qualität des Webservices sicherzustellen, wurde die Priorität auf Integrationstests gelegt. Integrationstests bieten die Möglichkeit, Entwurfskomponenten und insbesondere deren Schnittstellen sowie ihre Zusammenspiel zu betrachten. Im Fall des zuvor entwickelten und implementierten Webservice geht es um das Zusammenspiel der einzelnen Schichten (siehe Kapitel 4.3). Zu diesem Zweck werden eine Reihe von aufeinander abgestimmten Einzeltest durchgeführt.

Prinzipiell enthält Node.js alle Werkzeuge um als Testumgebung zu fungieren. Allerdings empfiehlt es sich Test-Frameworks einzusetzen, da diese üblicherweise viel Komfort bieten und sich um zahlreiche Aspekte der Testausführung und -analyse kümmern. Es wird sich zahlreichen Test-Frameworks bedient, um eine Testinfrastruktur zu schaffen, die nachfolgend erläutert werden soll.

Die Basis der Testinfrastruktur bildet das Test-Framework mocha (siehe Tabelle 5.1), das einfach gehalten und besonders für das Testen von asynchronen Code geeignet ist. Mocha lässt sich von der Kommandozeile ausführen und wird zur einfacheren Handhabung über ein Makefile¹³⁴ gestartet („make test“), da üblicherweise viele Parameter übergeben werden (siehe Abb. 6.1). Mocha verfolgt den Ansatz des sogenannten *Behavior Driven Developments* (BDD). Kerngedanke des BDD ist es, dass nicht nur Entwickler Tests formulieren können, sondern diese ebenfalls von Personen erstellt werden können, die nicht an der Entwicklung beteiligt waren. Mit BDD werden anders als in Unit Tests nicht nur atomare Einheiten im Quellcode, sondern das Verhalten bestimmter Komponenten der Applikation getestet.

Außerdem wird das Framework *should.js* verwendet, welches eine, zum Stil des BDD passende Syntax (siehe Abb. 6.1) einführt, um zu überprüfen, ob eine Aussage über den Zustand eines Objektes korrekt ist.

```

1 require('should');
2 var user = {
3   name: 'Malte',
4   age: 25,
5   likes: ['Hamburger', 'Pizza', 'Sushi']
6 };
7
8 user.should.have.property('name', 'Malte');
9 user.age.should.be.exactly(25).and.be.a.Number;
10 user.should.have.property('likes').with.lengthOf(3);

```

Abbildung 6.1 - Testen eines Objektes mit *should.js*
Quelle: eigene Darstellung

Um während der Tests auf die Datenbank zuzugreifen, wird sich dem Modul *neo4j-test* bedient. Dieses beinhaltet unter anderem einen Neo4j-Server, welcher durch die Nutzung eines anderen Ports mit anderen Neo4j-

¹³⁴ In Makefiles werden normalerweise die Schritte und Abhängigkeiten zusammengefasst, die zum Kompilieren eines Programms erforderlich sind

Servern koexistieren kann. Um die Datenbank zwecks Testbarkeit mit Daten zu füllen, wurde ein Script geschrieben, welches Stories aus einer JSON-Datei einliest, daraus Neo4j Statements erstellt und diese in einer temporären Datei speichert (testing/testdata.js). Ferner wurde ein Script aus dem neo4j-test Modul, welches den Server startet bzw. stoppt, um Funktionalitäten erweitert, die es ermöglichen, den Server beim Start mit den Testdaten aus der temporären Datei zu befüllen (vgl. Abb. 6.1 Z. 7, Parameter -t oder --test) und die Datenbank nach dem Stoppen wieder zurücksetzt (vgl. Abb. 6.1 Z. 21, Parameter -c oder --clean).

```

1 test:
2     @-echo "-----"
3     @-echo "-"
4     @-echo "-----"
5     # Start Neo4j-Server and create testdata (-t)
6     # The - suppresses warnings
7     @./testing/neo4j-test -tq
8
9     # Run tests
10    @-NODE_ENV=test ./node_modules/.bin/mocha \
11        --require should \
12        --timeout 4000 \
13        --reporter spec \
14        --harmony \
15        --bail \
16        rest/*/test.js
17
18    @-echo "-----"
19    @-echo ""
20    # Stop Neo4j-Server and remove all testdata (-c)
21    @./testing/neo4j-test -cq
22
23 .PHONY: test

```

Abbildung 6.2 - Makefile

Quelle: eigene Darstellung

Nachdem der Server gestartet ist (Z. 7), werden die eigentlichen Tests durchgeführt (Z. 10-16). Wichtig ist an dieser Stelle, dass zuvor der Umgebungsvariable „NODE_ENV“ der Wert „test“ zugewiesen wird, da dies unter anderem dafür sorgt, dass innerhalb des Webservices die zuvor gestartete Testdatenbank verwendet wird. Die eigentlichen Tests sind innerhalb der einzelnen Ressourcen der REST-Schicht in der Datei test.js definiert (vgl. Z. 16).

Um den Webserver innerhalb des Tests zu verwenden dient das Modul „supertest“. Es startet den übergebenen Webserver beziehungsweise die übergebene Funktion auf einem zufällig ausgewählten Port. Da die Testanfrage an den Server asynchron stattfindet, muss eine *done*-Funktion übergeben (Z. 19) und diese am Ende des Tests aufgerufen (Z. 29) werden um mocha mitzuteilen, dass der Test abgeschlossen ist.

In Abbildung 6.3 wird die Testdatei der Ressource „Account“ exemplarisch und verkürzt dargestellt. Mocha bietet mit der Methode „describe“ (Z. 2, 17) die Möglichkeit, die Tests logisch zu strukturieren. Das erste Argument der Methode ist eine Zeichenkette, welche beschreibt, was in den folgenden Tests geprüft wird. Das zweite Argument ist eine Funktion, welche die eigentlichen Tests enthält. Die Testfunktionen werden durch den Aufruf der *it*-Methode (Z. 19) von mocha gekapselt. Auch hier ist das erste Argument eine Beschreibung, was der Test prüft. Im zweiten Argument wird innerhalb einer Funktion der Test selbst formuliert. Durch die Methoden *beforeEach* wird vor jedem Test die korrekte Umgebung für den Test initialisiert, die Methode *afterEach* führt Aufräumarbeiten nach jedem Test durch.¹³⁵

¹³⁵ vgl. [MOCHA]

Der eigentliche Test ist nach dem Schema *Arrange – Act – Assert* (AAA) aufgebaut. Es besagt, dass ein Test stets aus drei, aufeinander aufbauenden Abschnitten besteht.¹³⁶ Die Instanziierung des Webservers in der `beforeEach`-Methode und der Anfrage („request“, Z. 20) in der Test-Methode stellt den ersten Schritt des Tests dar (*Arrange*). Hier wird für die Grundlage des Tests gesorgt. Der nächste Schritt besteht aus der Ausführung (*Act*) der Anfrage mittels der Methode „end“ (Z. 25). Zuletzt wird in der Callback-Methode (Z. 25-30) das tatsächliche mit dem zu erwarteten Ergebnis verglichen (*Assert*).

```

1 var request = require('supertest');
2 describe('ACCOUNT \n -----', function()
3 {
4     var api = require('../');
5     var apiToken = "mIQT8FDbMsvb6eZW63nuOilA9eZqpwEh";
6     var app = {};
7
8     beforeEach(function(done) {
9         app = api();
10        done();
11    });
12    afterEach(function(done) {
13        app = {};
14        done();
15    });
16
17    describe('POST /login', function()
18    {
19        it('should login the user', function(done) {
20            request(app.listen())
21                .post('/login')
22                .send({ email: 'malte@schonvogel.net', password: 'wurstwurst'})
23                .set('api-token', apiToken)
24                .expect(200)
25                .end(function(err, res){
26                    if (err) return done(err);
27                    res.body.should.have.property('token');
28                    res.body.should.have.property('tokenLifetime');
29                    done();
30                });
31        });
32        [...]
33    });
34    [...]
35 });

```

Abbildung 6.3 - Einfacher, exemplarischer Testfall
Quelle: eigene Darstellung

6.2 iPhone-Applikation

Während der Implementierungsphase wurde die Anwendung auf Ihre Funktionalität an Hand von White- und Black-Box-Tests getestet. Dabei lassen sich die durchgeführten Tests in vier Unterkategorien fassen:

6.2.1 Unit Tests

Bei Unit Tests, auch Komponententests, werden einzelne Module der Software getestet, wobei neben der funktionalen Korrektheit und Vollständigkeit der Komponente (Unit) die korrekte Fehlerbehandlung, die Überprüfung der Eingabedaten, Korrektheit der Ausgabedaten sowie die Performance behandelt wird.¹³⁷

Im Gegensatz zum Webservice (Kapitel 6.1) für den die Testumgebung erst aufwändig entwickelt werden musste, ist das Einrichten einer entsprechenden Umgebung für die iPhone-Applikation nicht nötig. Xcode

¹³⁶ vgl. [ARRAS]

¹³⁷ vgl. [RAEMAN] S. 105

richtet bereits bei Erstellung des Projektes automatisch eine Testumgebung (Test Target) ein. Das verwendete Testframework von Xcode, XCTest, ist seit Xcode 5 fester Bestandteil der Entwicklungsumgebung. Tests werden in Klassen die von der Klasse XCTestCase erben gruppiert und jede Testmethode muss einen Namen haben, der mit `test` beginnt. Da Tests einfache Klassen und Methoden sind, ist es möglich Helper-Methoden und Variablen hinzuzufügen.

6.2.2 Simulator

Während der Entwicklung war das wichtigste Testinstrument der in SDK integrierte iPhone-Simulator. Durch die Xcode Funktion *Build and Run* ist es möglich, die Applikation in wenigen Sekunden auf dem Simulator zu testen. Besonders nützlich war diese Möglichkeit während der Erstellung des Userinterfaces. Ferner ermöglicht der Simulator die Simulation von der Hardware ausgelösten Ereignissen. Beispielsweise kann ein Speichernotstand ausgerufen werden, um zu kontrollieren, ob die Methoden zur Behandlung dieses Problems korrekt funktionieren. Bei Views mit vielen Bildern konnte anhand dieser Funktionalität beispielsweise getestet werden, ob der Cache ordnungsgemäß geleert wird.

6.2.3 Geräte

Aufgrund der Tatsache, dass sich der Simulator in vielen Situation leicht anders verhält als ein physisches Gerät, ist dieses ein sehr wichtiges Element der Softwareentwicklung und somit wird es unvermeidlich Tests auf diesem durchzuführen. Grundlegende Unterschiede zwischen Simulator und Gerät sind zum einen durch die unterschiedliche Hardware bedingt. Während Xcode den Software-Stack bei einem „echten“ iPhone für einen ARM-Prozessor kompiliert, wird beim Ausführen des Simulators für die CPU (x86) des Macs kompiliert. Der Simulator nähert sich bezüglich Softwarekompatibilität und Hardware dem physischen Gerät an, verhält sich allerdings nicht identisch. Applikationen, die auf dem Simulator funktionsstüchtig sind und fehlerfrei erscheinen, können auf dem Gerät unerwartet aussetzen und abstürzen.

Als Testgeräte wurde ein iPhone 4 (iOS 7) sowie ein iPhone 6 (iOS 8) verwendet, wobei Ersteres das Haupttestgerät war, da es die Minimalanforderung der Applikation darstellt. Die Wahl dieser vergleichsweise schwachen Hardware als Testgerät hat zum Vorteil, dass das Thema Performance beim Testen allgegenwärtig war und zu diesem Zweck die Applikation stetig optimiert werden musste. Ein Beispiel für den Nutzen dieser Vorgehensweise stellt das in Kapitel 5.4.3.1 beschriebene Problem der Zuweisung von asynchron geladenen Bildern an eine UICollectionView Zelle dar. Das beschriebene Problem trat weder im Simulator, noch auf dem iPhone 6 auf. Ein weiteres Beispiel stellt der Test der in Tabelle 4.7 aufgelisteten Gesten dar. Im Mittelpunkt des Tests stand das vergleichsweise kleine Display des iPhone 4 da einige Gesten unter Umständen mehr als den verfügbaren Platz brauchten, um ausgeführt zu werden (z.B. die Wischgeste auf eine Zelle einer Story). Nach mehreren Versuchen mit Fingern verschiedener Größe, stellten sich alle Gesten als umsetzbar heraus.

6.2.4 Usability

Besonders bei Entwurf und Implementierung ist iteratives Vorgehen wichtig. Bereits mit den, in Kapitel 4 dargestellten *Mockups* wurden kleinere Tests durchgeführt und Testpersonen nach Sinn und Zweck einzelner Komponenten befragt. Dabei sehen die Entwürfe der Oberflächen bewusst unfertig aus, um dem Tester zu suggerieren, dass sich noch einiges verändern lässt. Im weiteren Verlauf der Entwicklung wurde der Prototyp an Personen getestet, die gemäß der in Kapitel xyz definierten Zielgruppe als Tester in Frage kamen. Hier hat sich

die sogenannte *Hallway-Test*-Methode bewährt. Bei dieser Art von Test wird die Software im Prototypen- oder Endstadium von Personen getestet, die nicht zum Entwicklerteam gehören und der Zielgruppe entsprechen, sodass vor allem Probleme bei der Bedienung und mögliche Programmfehler bei der Eingabe erkannt werden.

7 Deployment

In diesem Kapitel wird das Deployment der einzelnen Komponenten des verteilten Systems beschrieben. Um die Applikation ausserhalb der Entwicklungsumgebung zu nutzen, ist es notwendig, dass die Komponenten des verteilten Systems über das Internet erreichbar sind.

Für die Anwendungs- und die Datenschicht wird je eine Server-Instanz benötigt. Um der Anforderung der Skalierbarkeit gerecht zu werden, ist ein Anbieter zu wählen, der diese notwendige Infrastruktur als einen Service bereitstellt (Infrastructure as a Service) und zudem die Möglichkeit bietet, bei erhöhter Belastung weitere Instanzen hinzuzuschalten. Ferner sollte es den Instanzen möglich sein, innerhalb des Rechenzentrums untereinander zu kommunizieren. Der Performance Guide von Neo4j empfiehlt zudem den Einsatz einer SSD als Massenspeicher, um Anfragen schneller zu bearbeiten zu können.

Die in Kapitel 7.1 und 7.2 getätigten Einstellungen wurden auf je einer Server-Instanz mit dem Betriebssystem Debian Linux Version 6.0 x64, 512 MB Ram, 1 CPU und 20 GB SSD durchgeführt.

7.1 Konfiguration Anwendungsschicht

1. Erstellung des API-Users

```
1 useradd -s /bin/bash -m -d /home/apiuser -c "apiuser" apiuser
2 passwd apiuser
3 usermod -aG sudo apiuser
```

Abbildung 7.1 - Erstellung des API-Users
Quelle: eigene Darstellung

2. Installieren von Programmen um Node.js zu kompilieren

```
1 sudo apt-get install git
2 sudo apt-get install build-essential
3 sudo apt-get install curl openssl libssl-dev
```

Abbildung 7.2 - Installieren von Abhängigkeiten
Quelle: eigene Darstellung

3. Installation der aktuellsten Version von Node.js

```
1 git clone https://github.com/joyent/node.git
2 cd node
3 git checkout v0.11.14
4 ./configure
5 make
6 sudo make install
```

Abbildung 7.3 - Node.js Installation
Quelle: eigene Darstellung

4. Hochladen und Entpacken, der für den Zugriff auf das Git-Repository, benötigten Schlüssel

```
1 scp ssh.tar root@141.45.200.123:/home/apiuser
2 cd /home/apiuser
3 tar xfv ssh.tar
```

Abbildung 7.4 - Zugriffschlüssel einrichten
Quelle: eigene Darstellung

5. Clonen des Git-Repositories

```
1 cd /var/www && git clone git@github.com:mschonvogel/foyer-backend.git
```

Abbildung 7.5 - Git-Repository klonen
Quelle: eigene Darstellung

6. Installieren des Prozessmanagers pm2

```
1 sudo npm install pm2@latest -g --unsafe-perm
```

Abbildung 7.6 - Prozessmanager pm2 Installation
Quelle: eigene Darstellung

7. Starten des Prozessmanagers als Nutzer „apiuser“

```
1 pm2 --run-as-user apiuser --run-as-group apiuser start /var/www/foyer-backend/pm2-
2 live.json
3 pm2 save
4 sudo env PATH=$PATH:/usr/local/bin pm2 startup debian -u apiuser
```

Abbildung 7.7 - Starten des Prozessmanagers
Quelle: eigene Darstellung

8. Erstellen eines Firewalls - firewall.sh

```
1#!/bin/sh
2echo "Installing firewall"
3
4for tbl in iptables ip6tables; do
5    # Flush tables
6    $tbl --flush INPUT
7    $tbl --flush FORWARD
8    $tbl --flush OUTPUT
9
10   # Set default policy
11   $tbl --policy INPUT DROP
12   $tbl --policy FORWARD DROP
13   $tbl --policy OUTPUT ACCEPT
14
15   # Allow localhost
16   $tbl -A INPUT -i lo -j ACCEPT
17   $tbl -A OUTPUT -o lo -j ACCEPT
18
19   # Allow established connections
20   $tbl -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
21
22   # Rules
23   #      <chain>    <device>      <proto>    <port>      <action>
24   $tbl -A INPUT  -i eth0       -p tcp     --dport 22     -j ACCEPT
25   $tbl -A INPUT  -i eth0       -p tcp     --dport 80     -j ACCEPT
26   $tbl -A INPUT  -i eth0       -p tcp     --dport 3000   -j ACCEPT
27   $tbl -t nat    -I PREROUTING -p tcp     --dport 80     -j REDIRECT --to-port 3000
28   $tbl -A INPUT  -i eth0       -p tcp     --dport 1338   -j ACCEPT
29   $tbl -A INPUT  -i eth0       -p tcp     --dport 443    -j ACCEPT
30 done
31
32 # Allow ping
33 iptables    -A INPUT          -p icmp        -j ACCEPT  # ping
34 ip6tables   -A INPUT          -p icmpv6     -j ACCEPT  # ping ipv6
```

Abbildung 7.8 - Firewall der Anwendungsschicht
Quelle: eigene Darstellung

9. Firewall bei jedem Start ausführen

```
1 vim /etc/rc.local
2 # add /root/firewall.sh
```

Abbildung 7.9 - Konsolbefehl zum bearbeiten der Datei rc.local
Quelle: eigene Darstellung

7.1.1 Auto-Deployment über GitHub

Für die Versionskontrolle der Anwendungsschicht, wird, wie in Kapitel 5.1 beschrieben, die Versionsverwaltungssoftware *Git* eingesetzt. Für das Hosting des Git Repositories¹³⁸ wird der Hostingdienst GitHub¹³⁹ verwendet. Um eine Trennung zwischen Entwicklungs- und Produktionsumgebung zu erreichen, werden in dem Repository zwei Branches¹⁴⁰ erstellt: *master* für die (lokale) Entwicklungsumgebung und *live* für die Version der Software, die auf dem Server zur Anwendung kommt, der im Produktivbetrieb ist.

GitHub bietet über einen Service namens *Webhooks*¹⁴¹ die Möglichkeit, Benachrichtigungen an externe Server zu senden, falls bestimmte Ereignisse auftreten. Diese Funktionalität wird genutzt, um den Server der Anwendungsschicht darüber zu informieren, dass Änderungen am *live*-Branch vorgenommen wurden. Zu diesem Zweck wird ein eigener Node.js Server genutzt (vgl. Abb. 7.10), welcher den durch den Firewall in Abbildung 7.8 (Z. 28) geöffneten Port 1338 verwendet, um Benachrichtigungen von GitHub zu empfangen. Falls diese Benachrichtigungen den *live*-Branch betreffen, (Abb. 7.10, Z. 7) wird die in Abbildung 7.11 dargestellte shell-Datei ausgeführt, welche in das Projektverzeichnis wechselt und das Repository aktualisiert.

```

1 var gith = require('gith').create( 1338 );
2 var execFile = require('child_process').execFile;
3
4 gith({
5   repo: 'mschonvogel/foyer-backend'
6 }).on( 'all', function( payload ) {
7   if( payload.branch === 'live' ) {
8     execFile('/var/www/foyer-backend/deploy/hook.sh', function(err, stdout, stderr) {});
9   }
10 });

```

Abbildung 7.10 - Node.js-Server der vom Webhook aufgerufen wird
Quelle: eigene Darstellung

```

1 #!/bin/bash
2
3 cd /var/www/foyer-backend/foyer-api && git pull origin live > /dev/null

```

Abbildung 7.11 - Shellscript zur Aktualisierung des Repositories
Quelle: eigene Darstellung

7.2 Konfiguration der Datenhaltungsschicht

1. Erstellung des Systemnutzers

```

1 useradd -s /bin/bash -m -d /home/dbuser -c "dbuser" dbuser
2 passwd dbuser
3 usermod -aG sudo dbuser

```

Abbildung 7.12 - Erstellung des Datenbank-Nutzers
Quelle: eigene Darstellung

2. Installieren von Debian Quellen und Neo4j

¹³⁸ Code-Repository / Projektordner

¹³⁹ <https://github.com>, abgerufen am 10.02.2015

¹⁴⁰ Innerhalb eines Repositories kann es mehrere Versionen einer Software geben. Jede Version stellt dabei einen Branch dar.

¹⁴¹ <https://developer.github.com/webhooks/>, abgerufen am 10.02.2015

```

1 wget -O - http://debian.neo4j.org/neotechnology.gpg.key | apt-key add -
2 echo 'deb http://debian.neo4j.org/repo stable/' > /etc/apt/sources.list.d/neo4j.list
3 aptitude update -y
4 aptitude install neo4j -y

```

Abbildung 7.13 - Installation Neo4j und Abhängigkeiten
Quelle: eigene Darstellung

3. Server von außen erreichbar machen

```

1 vim /etc/neo4j/neo4j-server.properties
2 # org.neo4j.server.webserver.address

```

Abbildung 7.14 - Änderung Konfigurationsdatei Neo4j
Quelle: eigene Darstellung

4. Erstellen eines Firewalls - firewall.sh

```

1#!/bin/sh
2echo "Installing firewall"
3
4for tbl in iptables ip6tables; do
5    # Flush tables
6    $tbl --flush INPUT
7    $tbl --flush FORWARD
8    $tbl --flush OUTPUT
9
10   # Set default policy
11   $tbl --policy INPUT DROP
12   $tbl --policy FORWARD DROP
13   $tbl --policy OUTPUT ACCEPT
14
15   # Allow localhost
16   $tbl -A INPUT -i lo -j ACCEPT
17   $tbl -A OUTPUT -o lo -j ACCEPT
18
19   # Allow established connections
20   $tbl -A INPUT -m state --state ESTABLISHED,RELATED -j ACCEPT
21
22   # Rules
23   #      <chain>    <device>    <proto>    <source>          <port>      <action>
24   $tbl -A INPUT -i eth0 -p tcp      --dport 22 -j ACCEPT
25   $tbl -A INPUT -i eth1 -p tcp      -s 10.133.242.69 --dport 7474 -j ACCEPT
26 done
27
28 # Allow ping
29 iptables -A INPUT -p icmp -j ACCEPT # ping
30 ip6tables -A INPUT -p icmpv6 -j ACCEPT # ping ipv6

```

Abbildung 7.15 - Firewall der Datenhaltungsschicht
Quelle: eigene Darstellung

8 Ergebnis

8.1 Zusammenfassung

Ziel dieser Arbeit war die Konzeption und prototypische Entwicklung der iPhone-Applikation sowie des Webservices. Um die gestellte Aufgabe zu lösen, wurden zunächst die benötigten Technologien und Konzepte untersucht. In der anschließenden Analyse wurden unter dem Augenmerk der Erreichung eines minimum viable products die Muss-Anforderungen des Prototyps bewusst ausgewählt und ergänzende Anforderungen in den Kann-Anforderungen formuliert. Im Anschluss wurde auf Basis der zuvor definierten Anforderungen die Architektur des Webservices unter dem Augenmerk der Skalier- und Wartbarkeit entwickelt, die Views der iPhone-Applikation anhand von Mockups detailliert erläutert sowie mögliche Technologien für deren Umsetzung beschrieben. Nachfolgend wurde der Webservice und die iPhone-Applikation prototypisch implementiert und die Umsetzung anhand von Beispielen dargestellt sowie die erstellte Software ausgiebig getestet. Schlussendlich wurden Strategien für das Deployment entwickelt sowie deren Umsetzung beschrieben und schließlich auch durchgeführt.

8.2 Abschließende Betrachtung

Im Rahmen der Arbeit wurden alle definierten Muss- (17/17) und der Großteil (18/22) der Kann-Anforderungen erfüllt. Für die Bearbeitung des Prototyps wurden etwa fünf Wochen aufgebracht; weitere fünf Wochen der Bearbeitungszeit wurden für die schriftliche Ausarbeitung verwendet. Ohne die bereits vorhandene Erfahrung mit den verwendeten Technologien und Frameworks wäre es nicht möglich gewesen, alle Anforderungen umzusetzen.

Besonders die Programmiersprache Swift, die erst drei Monate vor Beginn der Bearbeitungszeit das Beta Stadium verlassen hatte (Literatur war deswegen bis auf die offiziellen Dokumente von Apple nicht vorhanden), bereitete anfangs Probleme, da der Autovervollständigungs-Mechanismus der Entwicklungsumgebung nicht funktionierte und die Entwicklung aufgrund dessen sehr verlangsamt wurde. Auch der Anspruch der Kompatibilität mit dem Betriebssystem iOS 7 und die Unterstützung des iPhone 4 wirkte sich sehr auf die Entwicklung aus: positiv auf die Qualität der Applikation und negativ auf die Entwicklungszeit, da dadurch einige Fehler sichtbar wurden, die sonst nicht zu erkennen gewesen wären. Die konsequente programmatische Umsetzung der Views sorgte zum einen dafür, dass das Entdecken von Fehlern während der Entwicklung erleichtert wurde, brachte zum anderen aber auch viel Boilerplate Code mit sich. Für zukünftige Projekte ist zu evaluieren, inwiefern sich ein hybrider Ansatz aus Interfacebuilder in Kombination mit programmatisch erstellten Klassen besser eignet.

Des Weiteren fällt bei einer abschließenden Betrachtung der implementierten Controller auf, dass sobald zunehmend komplexere UI-Anforderungen implementiert werden, die Businesslogik innerhalb eines Viewcontrollers durch die Kommunikation mit den Views stark ansteigt und in großen, schwer lesbaren Controller Klassen resultiert. Für folgende Projekte sollte überprüft werden, ob das von Apple standardmäßig verwendete MVC-Entwurfsmuster nicht durch ein passenderes ersetzt werden kann. Hier könnte das Entwurfsmuster MVVM Abhilfe schaffen.

Die Abfragen an die Graphdatenbank Neo4j lieferten anfangs nicht die gewünschte Performance. Ein persönliches Gespräch mit einem Entwickler der Datenbank (und Autor des Buches „Neo4j - Eine Graphdatenbank für alle“) im Rahmen eines Neo4j-Meetups¹⁴² brachte viele neue Erkenntnisse. Ein anschließender reger Austausch per E-Mail ermöglichte weitere, entscheidende Verbesserungen. Allerdings bleibt anzumerken, dass die Möglichkeiten der Datenbank im Rahmen dieser Bachelorarbeit nicht ausgereizt wurden. So wurde das eigentliche Potential der Datenbank, die Erstellung von Echtzeit Empfehlungen, aufgrund der begrenzten Bearbeitungszeit nicht genutzt.

Eine willkommene Abwechslung stellte das Deployment des Webservices dar, dessen Einrichtung nötig war um die Prototypen der iPhone-Applikation auf dem Gerät zu testen, ohne dass eine Verbindung zum Computer besteht. Durch die Aufteilung in mehrere unabhängige Komponenten und die Nutzung von Cloud Computing hat der Prototyp ebenfalls viele Vorteile erfahren.

8.3 Ausblick

Zum Stand dieser Arbeit eignet sich der Prototyp bereits aufgrund des Funktionsumfangs für den produktiven Einsatz. Folgende Kann-Anforderungen konnten mangels Zeit nicht umgesetzt werden:

1. Über die Schnittstelle muss es möglich sein, sich über das soziale Netzwerk Facebook einzuloggen / zu registrieren.
2. Über die Schnittstelle muss es möglich sein, als Nutzer Stories zu kommentieren.
3. Die App muss es dem Nutzer ermöglichen, sich über das soziale Netzwerk Facebook einzuloggen / zu registrieren.
4. Die App muss es dem Nutzer ermöglichen, Stories zu kommentieren.

Neben der Implementierung der nicht umgesetzten Kann-Anforderungen, können unter anderem noch folgende Verbesserungen und Erweiterungen durchgeführt werden:

- **Im AppStore veröffentlichen**

Bevor die Applikation im AppStore veröffentlicht werden kann, sollte eine Testrunde mit mehreren Testern erfolgen. Apple bietet für diesen Zweck das Tool TestFlight¹⁴³ an. Außerdem müssen noch zahlreiche grafische Elemente, beispielsweise ein Appicon erstellt und Texte für den AppStore verfasst werden.

- **Testabdeckung erhöhen**

Um die Qualität bei zukünftigen Änderungen weiterhin zu gewährleisten, sollte die Testabdeckung der iPhone-Applikation durch weitere Komponententest erhöht werden. Zusätzlich zu den bestehenden Integrationstests, sollte die Qualität der Schnittstelle ebenfalls über Komponententest gesichert werden.

- **Nutzer-Empfehlungen**

Damit

- **Webversion umsetzen**

Ein plattformübergreifende Web-Version soll es ermöglichen, dass sich Nutzer auch ohne die iOS App-

¹⁴² Meetup mit dem Titel „Neo4j Expert Talks“ vom 19. Januar 2015 bei der innoQ Deutschland GmbH

¹⁴³ <https://developer.apple.com/testflight/>

likation qualitativ hochwertiger Beiträge bedienen können sowie dass dadurch passive virale Marketing Effekte¹⁴⁴ wirken.

- **Push-Benachrichtigungen umsetzen**

Push-Benachrichtigungen ermöglichen die unmittelbare Kommunikation des Webservices mit dem Gerät des Nutzers. Sie werden verwendet, um den Nutzer über Ereignisse (z.B. einen neuen Abonnenten, neue „gefällt mir“ Angaben) zu informieren und somit die Nutzung der Applikation auszulösen. Ähnlich wie die Umsetzung der Webversion ist dies eine Maßnahme, welche die Nutzung der Applikation befähigen wird.

- **Hashtags**

Um nach Stories und besonders Inhalten in Stories suchen zu können, soll dem Nutzer die Möglichkeit gegeben werden, Hashtags zu verwenden.

- **Automatisierter E-Mail-Newsletter**

Eine Wöchentliche Aussendung der neusten Stories soll Nutzer auf dem Laufenden halten und sie primär animieren, die Applikation zu nutzen.

- **Tagging auf Fotos ermöglichen**

Nutzern sollte es ermöglicht werden, Hashtags und andere Profile auf Bildern innerhalb von Stories zu markieren.

- **Filter verbessern**

Die momentan eingesetzten Filter dienen mehr einem Proof-of-Concept als einer tatsächlichen Aufwertung von Fotos. Hier gilt es ansprechende Filter zu entwickeln.

¹⁴⁴ vgl. [WIKI03]

A Literaturverzeichnis

- ALAM01 **Alamofire.** GitHub. *Alamofire*
<https://github.com/Alamofire/Alamofire>
[letzter Aufruf: 26.01.2015]
- APA01 **Apache.** *Apache Performance Tuning*
<http://httpd.apache.org/docs/current/misc/perf-tuning.html#completetime>
[letzter Aufruf: 26.01.2015]
- APLDEV01 **Apple.** Apple Developer. *About the iOS Technologies*
<https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/Introduction/Introduction.html>
[letzter Aufruf: 26.01.2015]
- APLDEV02 **Apple.** Apple Developer. *Core OS Layer*
<https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreOSLayer/CoreOSLayer.html>
[letzter Aufruf: 26.01.2015]
- APLDEV03 **Apple.** Apple Developer. *Core Services Layer*
<https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/CoreServicesLayer/CoreServicesLayer.html>
[letzter Aufruf: 26.01.2015]
- APLDEV04 **Apple.** Apple Developer. *Media Layer*
<https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/MediaLayer/MediaLayer.html>
[letzter Aufruf: 26.01.2015]
- APLDEV05 **Apple.** Apple Developer. *Cocoa Touch Layer*
<https://developer.apple.com/library/ios/documentation/Miscellaneous/Conceptual/iPhoneOSTechOverview/iPhoneOSTechnologies/iPhoneOSTechnologies.html>
[letzter Aufruf: 26.01.2015]
- APLDEV06 **Apple.** Apple Developer. *Model View Controller*
<https://developer.apple.com/library/mac/documentation/General/Conceptual/DevPedia-CocoaCore/MVC.html>
[letzter Aufruf: 26.01.2015]
- APLDEV07 **Apple.** Apple Developer. *Introduction to Core Foundation Design Concepts*
<https://developer.apple.com/library/ios/documentation/CoreFoundation/Conceptual/CFDesignConcepts/CFDesignConcepts.html>
[letzter Aufruf: 26.01.2015]
- APLDEV08 **Apple.** Apple Developer. *Collection View Programming Guide for iOS*
<https://developer.apple.com/library/ios/documentation/WindowsViews/Conceptual/CollectionViewPGforiOS/CollectionViewPGforiOS.pdf>
[letzter Aufruf: 26.01.2015]
- APLDEV09 **Apple.** Apple Developer. *Navigation Controllers*
<https://developer.apple.com/library/prerelease/ios/documentation/WindowsViews/Conceptual/ViewControllerCatalog/Chapters/NavigationControllers.html>
[letzter Aufruf: 26.01.2015]

- APLDEV10 **Apple.** Apple Developer. *iOS Human Interface Guidelines*
<https://developer.apple.com/library/ios/documentation/UserExperience/Conceptual/Mobile-HIG>
[letzter Aufruf: 26.01.2015]
- APLDEV11 **Apple.** Apple Developer. *App Store Review Guidelines*
<https://developer.apple.com/appstore/resources/approval/guidelines.html>
[letzter Aufruf: 26.01.2015]
- APLDEV12 **Apple.** Apple Developer. *UIActionSheet*
<https://developer.apple.com/library/prerelease/ios/documentation/UserExperience/Conceptual/UIKitUICatalog/UIActionSheet.html>
[letzter Aufruf: 26.01.2015]
- APLDEV13 **Apple.** Apple Developer. *Core Data Basics*
<https://developer.apple.com/library/mac/documentation/Cocoa/Conceptual/CoreData/Articles/cdBasics.html>
[letzter Aufruf: 26.01.2015]
- APLDEV14 **Apple.** Apple Developer. *NSUserDefaults*
https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes/NSUserDefaults_Class/index.html
[letzter Aufruf: 26.01.2015]
- APLDEV15 **Apple.** Apple Developer. *UIImagePickerController*
https://developer.apple.com/library/prerelease/ios/documentation/UIKit/Reference/UIImagePickerController_Class/index.html
[letzter Aufruf: 26.01.2015]
- APLDEV16 **Apple.** Apple Developer. *ALAssetsLibrary*
https://developer.apple.com/library/ios/documentation/AssetsLibrary/Reference/ALAssetsLibrary_Class/
[letzter Aufruf: 26.01.2015]
- APLDEV16 **Apple.** Apple Developer. *Properties*
https://developer.apple.com/library/prerelease/ios/documentation/Swift/conceptual/swift_programming_language/Properties.html
[letzter Aufruf: 26.01.2015]
- APLDEV16 **Apple.** Apple Developer. *ViewController Programming Guide*
<https://developer.apple.com/library/ios/featuredarticles/ViewControllerPGforiPhoneOS/Introduction/Introduction.html>
[letzter Aufruf: 26.01.2015]
- APLDEV17 **Apple.** Apple Developer. *NSIndexPath*
https://developer.apple.com/library/mac/documentation/Cocoa/Reference/Foundation/Classes NSIndexPath_Class
[letzter Aufruf: 26.01.2015]
- APLDEV18 **Apple.** Apple Developer. *Properties*
https://developer.apple.com/library/ios/documentation/Swift/Conceptual/Swift_Programming_Language/Properties.html
[letzter Aufruf: 26.01.2015]
- APLSWI01 **Apple.** Apple – Swift
<https://www.apple.com/de/swift>
[letzter Aufruf: 26.01.2015]

- APLSWI02 **Apple.** Apple Developer. *Swift – Overview*
<https://developer.apple.com/swift>
[letzter Aufruf: 26.01.2015]
- ARRAS **Arrange Act Assert.** *Why and what is Arrange Act Assert?*
www.arrangeactassert.com/why-and-what-is-arrange-act-assert
[letzter Aufruf: 03.02.2015]
- AWS01 **Amazon Web Services.**
<http://aws.amazon.com/de/campaigns/neu>
[letzter Aufruf: 26.01.2015]
- AWS02 **Amazon Web Services.** *AWS-Fallstudien*
<http://aws.amazon.com/de/solutions/case-studies/all/>
[letzter Aufruf: 26.01.2015]
- AWS03 **Amazon Web Services.** *Tools für Amazon Web Services*
<http://aws.amazon.com/de/tools>
[letzter Aufruf: 26.01.2015]
- AWS04 **Amazon Web Services.** Documentation. *Query Requests and Amazon SES*
<http://docs.aws.amazon.com/ses/latest/DeveloperGuide/query-interface-requests.html>
[letzter Aufruf: 26.01.2015]
- AWS05 **Amazon Web Services.** Documentation. *Authenticating Email in Amazon SES*
<http://docs.aws.amazon.com/ses/latest/DeveloperGuide/authentication.html>
[letzter Aufruf: 26.01.2015]
- AWS06 **Amazon Web Services.** Documentation. *Amazon Simple Storage Service*
<http://docs.aws.amazon.com/AmazonS3/latest/API/RESTObjectPUT.html>
[letzter Aufruf: 26.01.2015]
- AWS07 **Amazon Web Services.** Documentation. *Uploading Objects Using Pre-Signed URLs*
<http://docs.aws.amazon.com/AmazonS3/latest/dev/PresignedUrlUploadObject.html>
[letzter Aufruf: 26.01.2015]
- AWS08 **Amazon Web Services.** *Amazon Simple Storage Service*
<http://aws.amazon.com/de/s3>
[letzter Aufruf: 07.02.2015]
- BALHE01 **Balzert, Heide.** *Lehrbuch der Objektmodellierung.* 2. Auflage, Akademischer Verlag, 2004.
978-3-8274-2903-2
- BAUCH01 **Baun, Christian, et al.** *Cloud Computing - Web-basierte dynamische IT-Services.* 2. Auflage.
Springer, 2013. 978-3-642-18436-9
- BIT01 **BITKOM.** *Alle Smartphone-Nutzer machen Fotos*
http://www.bitkom.org/de/markt_statistik/64018_79882.aspx
[letzter Aufruf: 26.01.2015]
- BRUGRI Rik Van Bruggen. *Learning Neo4j.* 1. Auflage. Packt Publishing, 2014. 978-1849517164
- CLIDA01 **Clifford, Daniel.** *Google I/O 2012. Breaking the JavaScript Speed Limit with V8*
<https://www.youtube.com/watch?v=UJPdhx5zTaw>
[letzter Aufruf: 26.01.2015]
- CLOFL01 **CloudFlare.** *CloudFlare CDN*
<https://www.cloudflare.com/static/media/pdf/cloudflare-whitepaper-cdn.pdf>
[letzter Aufruf: 26.01.2015]
- COUGE01 **Coulouris, Georg, et al.** *Distributed Systems - Concepts and Design.* 5th Edition, International Edition. s.l.: Pearson Education, 2012. 978-0-273-76059-7

- DAHRY01 **Dahl, Ryan.** *JSConf 2011*
<http://nodejs.org/jsconf.pdf>
[letzter Aufruf: 26.01.2015]
- DAILYJS01 **DailyJS.** *Generators and Suspend*
<http://dailyjs.com/2013/05/31/suspend/>
[letzter Aufruf: 26.01.2015]
- DAZMI01 **Dazer, Michael.** Technical University Berlin. *RESTful APIs - Eine Übersicht*
http://www.snet.tu-berlin.de/fileadmin/fg220/courses/WS1112/snet-project/restful-apis_dazer.pdf
[letzter Aufruf: 26.01.2015]
- DEHNJ01 **njdehoog.** GitHub. *NHBalancedFlowLayout*
<https://github.com/njdehoog/NHBalancedFlowLayout>
[letzter Aufruf: 26.01.2015]
- ECMA01 **ECMA.** *ECMAScript Language Specification*
<http://www.ecma-international.org/publications/files/ECMA-ST-ARCH/ECMA-262,%203rd%20edition,%20December%201999.pdf>
[letzter Aufruf: 26.01.2015]
- ECMA02 **ECMA.** *The JSON Data Interchange Format*
<http://www.ecma-international.org/publications/files/ECMA-ST/ECMA-404.pdf>
[letzter Aufruf: 26.01.2015]
- EILKA01 **Eilebrecht, Karl und Starke, Gernot.** *Patterns kompakt.* 3. Auflage. Spektrum, 2010.
978-3-8274-2525-6
- FIERO01 **Fielding, Roy Thomas.** *Architectural Styles and the Design of Network-based Software Architectures*
https://www.ics.uci.edu/~fielding/pubs/dissertation/fielding_dissertation.pdf
[letzter Aufruf: 26.01.2015]
- FIRMA01 **Firtmann, Maximiliano.** *Programming the Mobile Web.* 2. Auflage, O'Reilly 2013.
978-0-5968-0778-8
- FLOLA01 **FahimF.** GitHub. *FloatLabelFields*
<https://github.com/FahimF/FloatLabelFields>
[letzter Aufruf: 26.01.2015]
- GAMER01 **Gamma, Erich et al.** *Entwurfsmuster.* 5 Auflage. Addison-Wesley, 1996. 3-8273-1862-9
- GIGAOM01 **Gigaom.** *AWS in fight of its life as customers like Dropbox ponder hybrid clouds and Google pricing*
<https://gigaom.com/2014/07/25/aws-in-fight-of-its-life-as-customers-like-dropbox-ponder-hybrid-clouds-and-google-pricing/>
[letzter Aufruf: 26.01.2015]
- GKPOP01 **gekitz.** GitHub. *GKPopoverController*
<https://github.com/gekitz/GKPopoverController>
[letzter Aufruf: 26.01.2015]
- GOO01 **Google.** Developers. *Chrome V8*
https://developers.google.com/v8/design#prop_access
[letzter Aufruf: 26.01.2015]
- HANEKE01 **Haneke.** GitHub. *Haneke / HanekeSwift*
<https://github.com/Haneke/HanekeSwift>
[letzter Aufruf: 26.01.2015]

HENCA01	Henderson, Carl. Building Scalable Websites. 1. Auflage. O'Reilly, 2006. 978-0-5961-0235-7
HUGTO01	Hughes-Croucher, Tom und Wilson, Mike. Node: Up and Running: Scalable Server-Side Code with JavaScript. 1. Auflage, O'Reilly, 2012. 978-1-4493-9858-3
HUNMI01	Hunger, Michael. Neo4j 2.0 - Eine Graphdatenbank für alle - schnell + kompakt. 1.Auflage. entwickler.press, 2014. 978-3-8680-2315-2
INFOQ01	InfoQ. Neo4j - an Embedded, Network Database http://www.infoq.com/news/2008/06/neo4j [letzter Aufruf: 26.01.2015]
INSTA	Instagram. Press Page http://instagram.com/press [letzter Aufruf: 09.02.2015]
JOUSA01	Jouili, Salim und Vansteenbergh, Valentin. An empirical comparison of graph databases http://www.odbms.org/wp-content/uploads/2014/05/an-empirical-comparison-of-graph-databases.pdf [letzter Aufruf: 26.01.2015]
JSON01	-. Introducing JSON http://json.org [letzter Aufruf: 26.01.2015]
KANG01	kangax. ECMAScript compatibility table http://kangax.github.io/compat-table/es6/ [letzter Aufruf: 26.01.2015]
KOA01	koa. GitHub. koajs / koa https://github.com/koajs/koa [letzter Aufruf: 26.01.2015]
KOA02	koa. GitHub. Koa vc Express https://github.com/koajs/koa/blob/master/docs/koa-vs-express.md#koa-vs-express [letzter Aufruf: 26.01.2015]
KOEDI	Dr. Köller, Dirk. iPhone-Apps entwickeln. 2. Auflage. Franzis, 2011. 978-3-645-60081-1
KUNJE01	Kunkle, Jeff. Node.js explained http://de.slideshare.net/JeffKunkle/nodejs-explained [letzter Aufruf: 26.01.2015]
LEAN01	The Lean Startup. Methodology http://theleanstartup.com/principles [letzter Aufruf: 26.01.2015]
MASMA01	Massé, Mark. REST API Design Rulebook. 1. Auflage. O'Reilly, 2012. 978-1-4493-1050-9
MDN01	Mozilla Developer Network. function* https://developer.mozilla.org/de/docs/Web/JavaScript/Reference/Statements/function%2A [letzter Aufruf: 26.01.2015]
MELIN01	Melzer, Ingo. Service-orientierte Architekturen mit Web Services http://cs.emis.de/LNI/Proceedings/Proceedings73/GI-Proceedings.73-10.pdf [letzter Aufruf: 26.01.2015]
MOCHA	Mocha. Mocha - the fun, simple, flexible JavaScript test framework http://mochajs.org [letzter Aufruf: 03.02.2015]

- NEO4J01 **Neo4j.** *Manual. Transaction Management*
<http://neo4j.com/docs/stable/transactions.html>
[letzter Aufruf: 26.01.2015]
- NEO4J02 **Neo4j.** *Manual. Cypher Introduction*
<http://neo4j.com/docs/stable/cypher-introduction.html>
[letzter Aufruf: 26.01.2015]
- NIST01 **NIST.** *The NIST Definition of Cloud Computing.*
<http://faculty.winthrop.edu/domanm/csci411/Handouts/NIST.pdf>
[letzter Aufruf: 26.01.2015]
- NODEJS01 **Node.js.**
<http://nodejs.org>
[letzter Aufruf: 26.01.2015]
- NODIN01 **Node.js.**
<http://nodejs.org/industry/>
[letzter Aufruf: 26.01.2015]
- NPM01 **NPM.** *What is NPM?*
<https://docs.npmjs.com/getting-started/what-is-npm>
[letzter Aufruf: 26.01.2015]
- NPM02 **NPM.** *packages people 'npm install' a lot*
<https://www.npmjs.com/#explicit>
[letzter Aufruf: 26.01.2015]
- NPM03 **NPM.** *express. Fast, unopinionated, minimalist web framework*
<https://www.npmjs.com/package/express>
[letzter Aufruf: 26.01.2015]
- ORACLE01 **Oracle.** *Online Transaction Processing.*
http://docs.oracle.com/cd/A87860_01/doc/server.817/a76992/ch3_eval.htm
[letzter Aufruf: 26.01.2015]
- PM201 **Unitech.** GitHub. PM2
<https://github.com/Unitech/pm2>
[letzter Aufruf: 26.01.2015]
- QUIRK01 **quirksmode.** *Mobile browsers*
<http://www.quirksmode.org/mobile/browsers.html>
[letzter Aufruf: 26.01.2015]
- RAEMAN **Rätzmann, Manfred.** *Software-Testing.* 1. Auflage. O'Reilly, 2002. 3-89842-271-2
- ROBIA01 **Robinson, Ian, Webber, Jim und Eifrem, Emil.** *Graph Databases.* 1. Auflage. O'Reilly, 2013.
978-1-4493-5626-2
- ROBIA02 **Robinson, Ian.** *Cypher: Adding a Node to a Linked List*
<http://iansrobinson.com/2013/06/20/cypher-adding-a-node-to-a-linked-list>
[letzter Aufruf: 26.01.2015]
- RODKL01 **Rodewig, Klaus M. und Wagner, Clemens.** Apps programmieren für iPhone und iPad. *Collection Views.* Rheinwerk Verlag.
http://openbook.rheinwerk-verlag.de/apps_programmieren_fuer_iphone_und_ipad/1915_05_009.html
[letzter Aufruf: 26.01.2015]
- SADER01 **Sadun, Erica.** *Das große iPhone Entwicklerbuch.* 1. Auflage. Addison-Wesley, 2012.
978-3-8273-2917-2

- SCHAL01 **Schill, Alexander und Springer, Thomas.** *Verteilte Systeme*. 2. Auflage. Springer, 2012. 978-3-642-25795-7
- SERAPH01 **BRIK AS.** GitHub. *Seraph.js*
<https://github.com/briktekhnologier/seraph>
[letzter Aufruf: 26.01.2015]
- SILTH01 **Sillmann, Thomas.** *Apps für iOS 8 professionell entwickeln*. 1. Auflage. Carl Hanser Verlag, 2014. 978-3-4464-4018-0
- SKIST01 **Skiena, Steven.** Department of Computer Science State University of New York. *The Partition Problem*
<http://www8.cs.umu.se/kurser/TDBAfl/VT06/algorithms/BOOK/BOOK2/NODE45.HTM>
[letzter Aufruf: 26.01.2015]
- SPRSE01 **Springer, Sebastian.** *Node.js - Das umfassende Handbuch*. 1. Auflage. Galileo Press, 2013. 978-3-8362-2119-1
- STAMA01 **Stäuble, Markus.** *Programmieren für iPhone und iPad*. 4. Auflage, dpunkt.verlag, 2012 978-3-8986-4810-3
- STRLO01 **StrongLoop.** *Generators in Node.js: Common Misconceptions and Three Good Use Cases*
<http://strongloop.com/strongblog/how-to-generators-node-js-yield-use-cases/>
[letzter Aufruf: 26.01.2015]
- SWIFO01 **ortuman.** GitHub. *SwiftForms*
<https://github.com/ortuman/SwiftForms>
[letzter Aufruf: 26.01.2015]
- W3C01 **W3C.** *Web Services Glossary*
<http://www.w3.org/TR/2004/NOTE-ws-gloss-20040211/#webservice>
[letzter Aufruf: 26.01.2015]
- W3C02 W3C. SOAP Version 1.2 Part 1: Messaging Framework (Second Edition)
<http://www.w3.org/TR/soap12-part1/>
[letzter Aufruf: 26.01.2015]
- WESERV01 **Images.weserv.nl.** *Image cache & resize proxy*
<http://images.weserv.nl/>
[letzter Aufruf: 26.01.2015]
- WIKI01 **Wikipedia.** *User-generated content*
http://de.wikipedia.org/wiki/User-generated_content
[letzter Aufruf: 26.01.2015]
- WIKI02 **Wikipedia.** *Xcode*
<http://de.wikipedia.org/wiki/Xcode>
[letzter Aufruf: 26.01.2015]
- WIKI03 **Wikipedia.** *Virales Marketing*
http://de.wikipedia.org/wiki/Virales_Marketing
[letzter Aufruf: 26.01.2015]
- WIKI04 **Wikipedia.** *Schichtenarchitektur*
<http://de.wikipedia.org/wiki/Schichtenarchitektur>
[letzter Aufruf: 26.01.2015]
- WIKI05 **Wikipedia.** *Representational State Transfer*
http://en.wikipedia.org/wiki/Representational_state_transfer
[letzter Aufruf: 26.01.2015]

- WIKI06 **Wikipedia.** *Node.js*
<http://de.wikipedia.org/wiki/Node.js>
[letzter Aufruf: 26.01.2015]
- WIKI07 **Wikipedia.** *ECMAScript*
<http://en.wikipedia.org/wiki/ECMAScript>
[letzter Aufruf: 26.01.2015]

B Glossar

Die Arbeit verwendet in der Softwareentwicklung allgemein übliches Vokabular. Einige spezielle Fachbegriffe sollen an dieser Stelle gesondert erläutert werden.

View	
Callback	
Query String	
API	
Deployment	

C Eigenständigkeitserklärung

Hiermit versichere ich, dass ich die vorliegende Bachelorarbeit selbstständig und nur unter Verwendung der angegebenen Quellen und Hilfsmittel verfasst habe. Die Arbeit wurde bisher in gleicher oder ähnlicher Form keiner anderen Prüfungsbehörde vorgelegt.

Berlin, den 17.02.2015

Malte Schonvogel

D Anhang

```

1  [
2    {
3      "title": "Meine Neue Story.",
4      "objectId": "7k4evX-Q",
5      "createdAt": 1420048450,
6      "updatedAt": 1420098403,
7      "cover": {
8        "objectId": "7klVlDXWQ",
9        "width": 745,
10       "height": 669,
11       "fileName": "20dbb3ee8a0ee059d83d1ff82bd12110D.JPG"
12     },
13     "sections": [
14       {
15         "objectId": "7kUNlvmZX",
16         "type": "fullscreen",
17         "title": "Black and White",
18         "text": null,
19         "items": [
20           {
21             "objectId": "XkD4xD7-m",
22             "fillRow": false,
23             "width": 1280,
24             "height": 850,
25             "fileName": "380266d91d4184bc5159cdab83e4d9deD.JPG"
26           }
27         ],
28       },
29     {
30       "objectId": "mJWNxwQZQ",
31       "type": "autolayout"
32       "title": "Im Wald",
33       "text": "Im Wald dort ist es schön. Also mir gefällt's da!",
34       "items": [
35         {
36           "objectId": "XJM4gP7-Q",
37           "fillRow": true,
38           "width": 1500,
39           "height": 1001,
40           "fileName": "f153a8a714f1382606dd9715959975a3D.JPG"
41         },
42         {
43           "objectId": "717ExDQWQ",
44           "fillRow": false,
45           "width": 1280,
46           "height": 850,
47           "fileName": "380266d91d4184bc5159cdab83e4d9deD.JPG"
48         }
49       ],
50     }
51   ]
52 },
53 [...]
54 ]
55 ]

```

Abbildung D.1 - Antwort auf die REST-Methode GET /stories

Quelle: eigene Darstellung

In der folgenden Grafik werden zur Übersichtlichkeit lediglich Klassennamen dargestellt. Controller, die in der TabBar referenziert werden, sind **fett** umrandet.

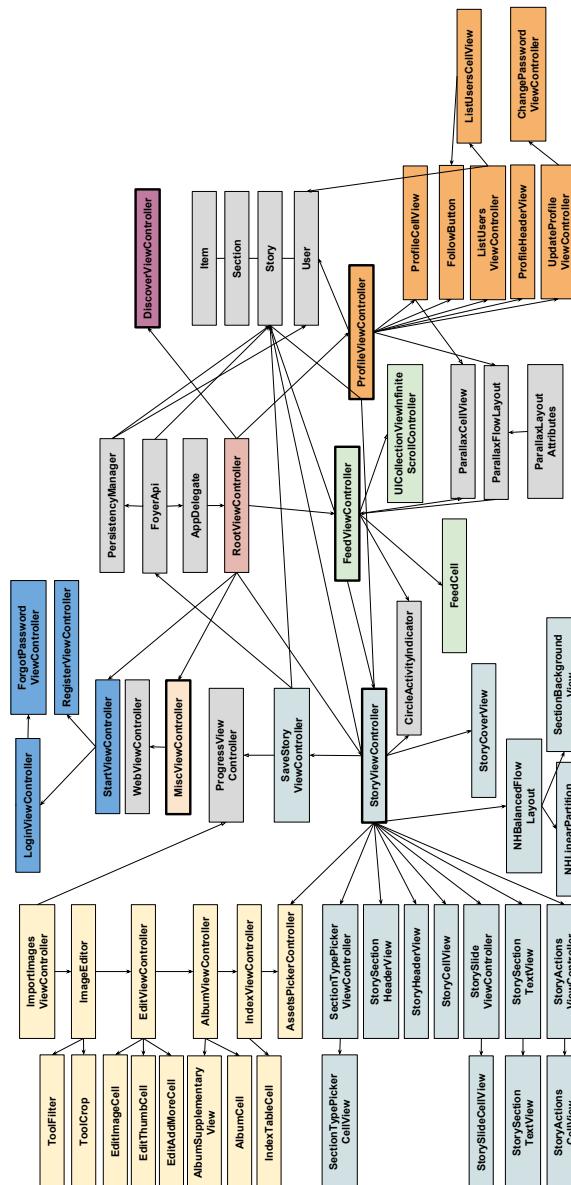


Abbildung D.2 - Klassendiagramm des iPhone-Prototypen
Quelle: eigene Darstellung

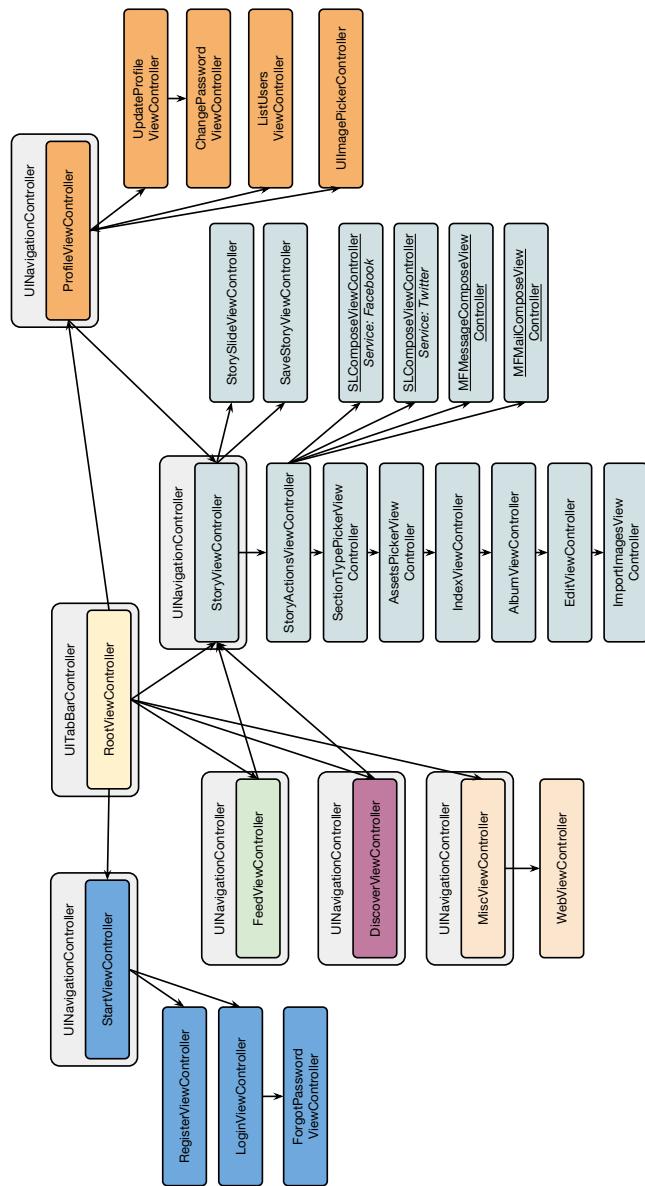


Abbildung D.3 - Navigationsbaum
Quelle: eigene Darstellung

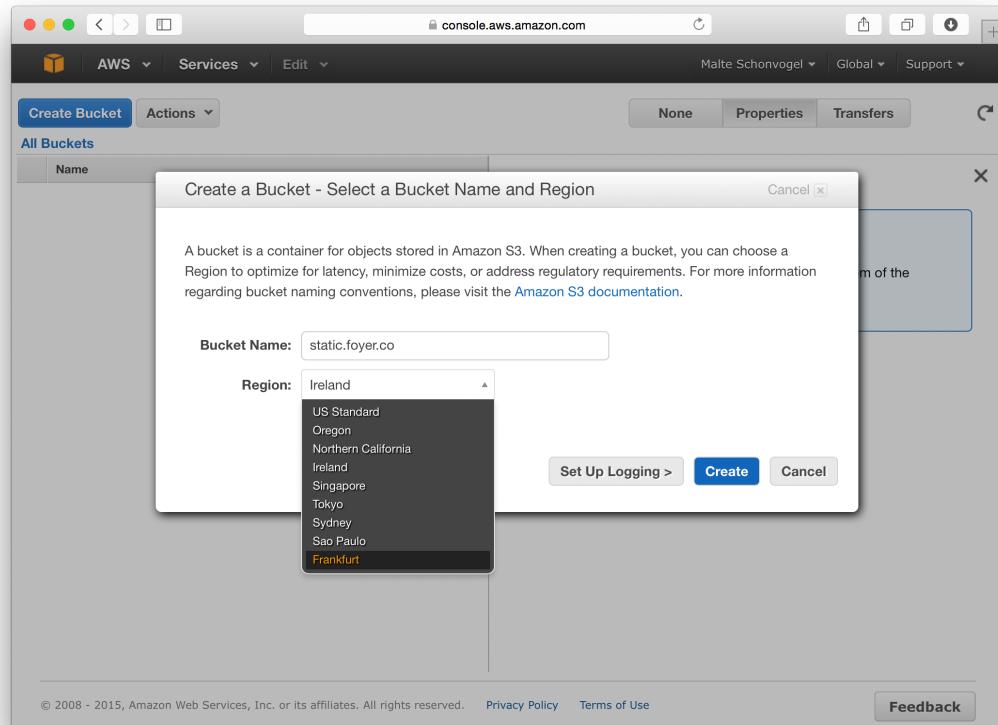


Abbildung D.4 - Einrichtung eines Buckets in AWS

Quelle: eigene Darstellung

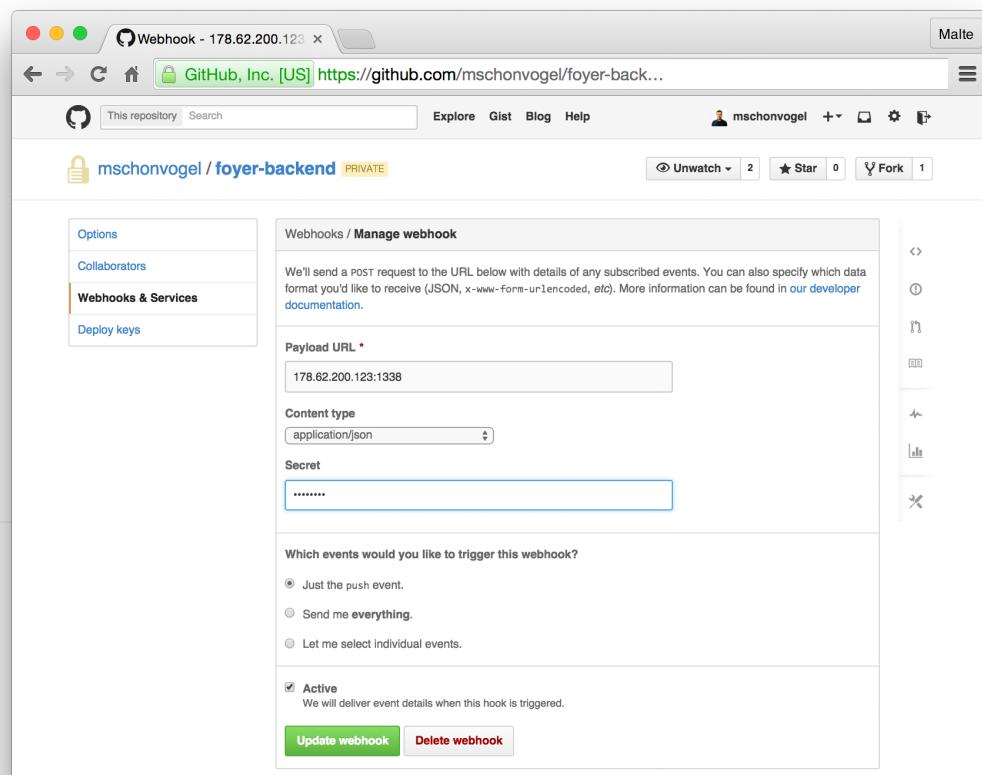


Abbildung D.5 - Einrichtung des Webhooks

Quelle: eigene Darstellung

In der folgenden Tabelle werden die Kapitel kenntlich gemacht, die in *Zusammenarbeit* entstanden sind:

Kapitel	Titel / Beschreibung
2.1	Webservices
2.3	Node.js
2.4	Skalierbarkeit
2.5	Grapgdatenbanken
2.6	Cloud Computing
2.7	Amazon Web Services
2.8	Smartphone-App
3.1	Definition „Story“
3.2	Funktionale Anforderungen
3.3	Nichtfunktionale Anforderungen
3.4	Anwendungsfälle
4.1	Architektur
4.2	Datenhaltungsschicht
4.3	Anwendungsschicht
4.4.2	Views (ausgeschlossen die Kapitel, die durch die Überschrift „Umsetzung“ gekennzeichnet sind)
5.2	Anwendungsschicht
5.3	Datenhaltungsschicht

Tabelle D.1 - Kenntlichmachung der Zusammenarbeit