The Bookshelf Project
Benjamin Altpeter und Lorenz Sieben

Bookshelf: Open Source eBook Management Software

Abstract

Mit Bookshelf entwickeln wir eine eBook management software, mit welcher Benutzer einfach eBooks verwalten und über mehrere Geräte hinweg abrufen können.
Kern des Projektes ist hierbei bookshelf-server, eine Webanwendung in PHP. bookshelf-server stellt sowohl das Backend, in welchem die Daten zu den Büchern verwaltet werden als auch ein Frontend, welches der Benutzer im Browser abrufen kann, zur Verfügung. Darüber hinaus wird eine externe API angeboten, über welche unabhängige Clients auf die Benutzerdaten zugreifen können.

Projektwebseite

http://getbookshelf.org

The Bookshelf Project
Benjamin Altpeter und Lorenz Sieben

Bookshelf: Open Source eBook Management Software

1. Auflage (Juni 2015)
Gesetzt in Adobe Garamond Pro, 11 pt

ISBN 978-1-5146-3767-8

Verlag:
Gabriele Altpeter, Internet Marketing-Services
Schreinerweg 6
D-38126 Braunschweig

kontakt@gabriele-altpeter.im

Inhaltsverzeichnis

1. Einleitung

Mit der zunehmenden Digitalisierung unseres Lebens lesen wir auch unsere Bücher immer häufiger als eBook statt in gedruckter Form. Dadurch werden natürlich auch neue Maßnahmen erforderlich, um unsere Bücher zu verwalten. War es früher vergleichsweise einfach, seine Bücher in physikalischer Form in einem Regal zu ordnen, gibt es heute nur noch Dateien, die über viele Ordner oder sogar Geräte verteilt sind.

Der Nutzer aber sucht nach einer einfachen Lösung. Er will all seine eBooks komfortabel verwalten und von überall Zugriff auf diese haben – egal, ob er an seinem Laptop, Tablet, Smartphone oder eBook-Reader sitzt.

1.1 Probleme mit vorhandenen Lösungen

Zwar gibt bereits einige Systeme, die Lösungen für eben dieses Problem versprechen, doch leider bringen auch diese selbst eigene Probleme mit sich oder erfüllen nicht all die Kriterien, die der Nutzer an sie stellt.

So gibt es einerseits diverse Lösungen von eBookdistributoren. An dieser Stelle soll beispielhaft Amazon mit der Kindleplattform besprochen werden, die genannten Aspekte treffen jedoch in großen Teilen auch auf ähnliche Systeme anderer Hersteller zu.
Amazon ist einer der größten eBookdistributoren der Welt. Zusätzlich zu dem Onlineshop bietet Amazon mit dem Kindle auch eine Reihe von eBook-Readern an, die sich großer Beliebtheit erfreuen. Über die sogenannte Whispernet-Technologie erlaubt Amazon seinen Kunden, all die über den Händler gekauften Bücher über die verschiedenen Gerä-

te zu synchronisieren, um so stets Zugriff auf diese zu haben.[1]
Während dieses System auch in der Praxis gut funktioniert, gibt es natürlich ein offensichtliches Problem damit: Die Kindleplattform beschränkt sich auf von Amazon verkaufte Bücher, der Kunde hat keine (oder zumindest nur eine sehr aufwendige, den eigentlichen Sinn des Systems verfehlende) Möglichkeit, eigene eBooks, welche er aus anderen Quellen erworben hat, auf die Geräte zu synchronisieren.

Andererseits gibt es jedoch auch einige händlerunabhängige Applikationen, welche die komfortable Verwaltung von eBooks ermöglichen sollen. Das bekannteste und am weitesten verbreitete Beispiel hierfür ist sicherlich die Open Source-Software calibre.[2]
Diese bietet dem Nutzer die Möglichkeit, all seine eBooks, unabhängig von der Bezugsquelle, zu verwalten. Allerdings gibt es auch mit calibre einige Probleme. So wird von vielen Nutzern kritisiert, dass die Bedienung des Programms wenig intuitiv und oftmals umständlich sei. Weiterhin ist es schwer, calibre über mehrere Geräte verteilt zu verwenden, da die Software lediglich für die Verwaltung an einem Computer entwickelt wurde.

1.2 Projektziel

Das Ziel hinter diesem Projekt ist es, eben diese Probleme zu lösen. Getrieben von unserem eigenen Bedarf, wollen wir eine Software entwickeln, die es dem Nutzer erlaubt, von all seinen Geräten und ohne großen Aufwand auf seine gesamte eBooksammlung zuzugreifen.

Dabei werden wir sämtliche Teile unter freien Lizenzen veröffentlichen, um unsere Software einem möglichst breiten Nutzerkreis zur Verfügung zu stellen und in späteren Phasen auch leicht weiteren Entwicklern die Möglichkeit zu geben, auf Bookshelf aufzubauen und die Software zu verbessern.
Konkret ist die vollständige Software unter der MIT License veröffentlicht, während die Dokumentationen unter der Creative Commons Attribution 4.0 International-Lizenz zur Verfügung gestellt werden. Sämtliche Projektergebnisse lassen sich stets in aktueller Version auf der

1 Amazon.com, Inc.: *Receiving Your Kindle Content via Whispernet*. 2009. URL: http://www.amazon.com/gp/help/customer/display.html?nodeId=200375890#whisptrans (Abruf am 14. Juni 2015).

2 Kovid Goyal: *calibre – About*. 2010.
URL: http://calibre-ebook.com/about (Abruf am 14. Juni 2015).

Projektwebseite getbookshelf.org abrufen. Eine Auflistung der wichtigsten Ressourcen findet sich in Kapitel 10.

2. Konzept

2.1 Wahl der Plattformen

Wir haben uns entschieden, die Serveranwendung, auf welche später die verschiedenen Clients zugreifen werden (s. Kapitel 2.2), in der Programmiersprache PHP zu implementieren. Zusätzlich zum Server werden wir auch einen beispielhaften Client in PHP entwickeln, der als Referenz für spätere Clients dient.

Der Hauptgrund für die Entscheidung, PHP zu verwenden liegt darin, dass es sich bei PHP um eine sehr weit verbreitete Websprache handelt. So öffnen wir bookshelf-server einerseits einer großen Community von Entwicklern, die bereits mit PHP vertraut sind und machen es anderseits auch Nutzern einfach, bookshelf-server zu installieren, da die überwiegende Mehrheit aller Webhostingpakete, auch im billigeren Preissegment, die Ausführung von PHP-Anwendungen unterstützt. Somit müssen Nutzer lediglich die Anwendung auf ihren Server hochladen und können bookshelf-server direkt verwenden.

Für die Speicherung der Anwendungsdaten (wie etwa der Metadaten) der Bücher fiel die Entscheidung auf das Datenbanksystem MySQL. Dieses ist nicht nur genau wie PHP sehr verbreitet und somit auf den meisten Maschinen sofort verfügbar, sondern bietet auch umfassende Möglichkeiten, eine Anfrage zu präzisieren. Außerdem finden sich viele verschiedene Datentypen in MySQL wieder. Und schließlich ist MySQL schon seit langem in PHP direkt integriert, was das aufwendige Einbinden von Bibliotheken erspart und eine optimierte Nutzung ermöglicht.

Der mit Windows, Mac OS und vielen Linux-Distributionen kompatible Client basiert auf der Plattform Qt, welche in C++ geschrieben wurde. Sie bietet ein großes Spektrum an Möglichkeiten und ist dabei trotzdem kompatibel zu nahezu allen Plattformen für Desktopcomputer. Deshalb ist Qt weltweit sehr beliebt und stark verbreitet, sodass sich Hilfe und Dokumentationen in großer Zahl und hoher Qualität finden. Das Compiling geschieht mit `gcc` und `qmake`, die beide zur Standardumgebung für Qt gehören.

Unter iOS ist die Wahl auf Objective C statt Swift gefallen. Die Entwickler bevorzugen den Syntax von Objective C. Darüber hinaus ist Swift momentan noch im Entwicklungsstatus.

Der Androidclient benutzt schließlich Java, wie für Androidentwicklung üblich.

2.2 App Concept

Bei bookshelf-server handelt es sich um eine vollständige Webanwendung, die sowohl Back- als auch Frontend beinhaltet. Dabei verwaltet das Backend die Daten (hauptsächlich Bücher und dazugehörige Metadaten) und stellt diese über eine interne und eine externe API in maschinenlesbarem Format zur Verfügung, während das Frontend eine Schnittstelle für den Benutzer darstellt, um über den Webbrowser auf die eigene eBooksammlung zuzugreifen (wobei dies, wie erwähnt, natürlich auch über unabhängige Clients geschehen kann).

Es ist zu beachten, dass es bei der Implementierung von bookshelf-server im Code keine explizite Unterscheidung bzw. Trennung zwischen dem Front- und Backend gibt. Sämtliche Funktionen werden in einer einheitlichen Codebasis (der sog. internen API) implementiert, auf welche sowohl die externe API als auch der Web Client zugreifen.[3]

Auf diese Weise versuchen wir, das bereits erwähnte Ziel, mit bookshelf-server eine Allroundlösung zur eBookverwaltung, welche der Nutzer besonders einfach installieren und verwenden kann, zu erreichen. Die Anwendungsarchitektur wird in Abbildung 2.1 grafisch dargestellt.

3 vgl. hierzu auch Diskussion unter https://git.my-server.in/bookshelf/bookshelf-server/issues/34

Zusätzlich zur Serveranwendung und dem Frontend gehört zu Bookshelf auch der Desktopclient. Er ermöglicht den einfachen und strukturierten Zugriff auf die Buchdatenbank und erleichtert die Bibliotheksverwaltung. In seiner aktuellen Version liefert er – ähnlich wie das Frontend – Informationen zu den Büchern und bietet die Möglichkeit, sie zu lesen.

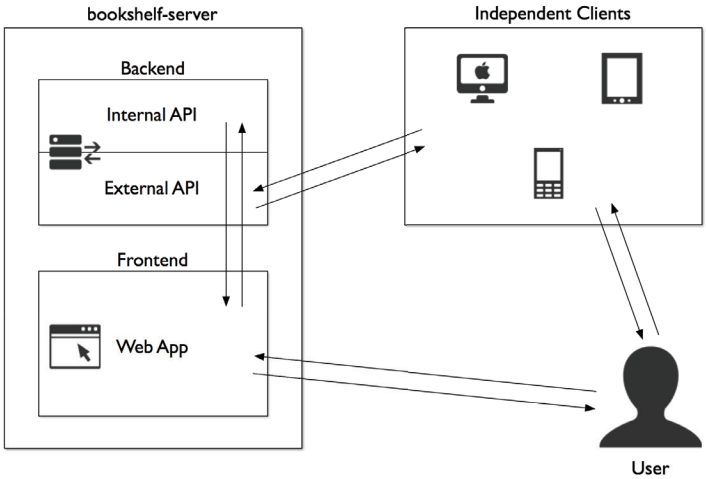

Abbildung 2.1: Architektur von bookshelf-server

2.3 Internal API

Die interne API stellt die Codebasis für alle Komponenten von bookshelf-server zur Verfügung. Hier sind alle wichtigen Funktionen (wie etwa zum Datenbankzugriff, zur Kommunikation mit externen Diensten, zu Datenstrukturen etc.) implementiert.

Die interne API ist in mehrere Namespaces aufgeteilt, die zur Sortierung der einzelnen Klassen und Funktionen dienen. Hier haben wir bei der Konzipierung darauf geachtet, die Anzahl der Namespaces kleiner-gleich fünf zu halten, um eine klare Struktur zu wahren.[4]
Unsere Namespaces und Klassen arbeiten im Einklang mit dem PHP

4 vgl. hierzu auch https://git.my-server.in/bookshelf/bookshelf-server/issues/5

FIG-Standard PSR-4.[5] Deshalb folgen sie diesem Benennungsformat, um nicht mit Drittsoftware zu interferieren:

```
\Bookshelf\<NamespaceName>\<ClassName>
```

Folgende Namespaces sind aktuell in Nutzung:[6]

- `\Bookshelf\Core`
  Dieser Namespace enthält Klassen, die für die Kernfunktion der Anwendung erforderlich sind. Dazu zählen momentan `Application` (`static`, hält grundlegende Konstanten wie die Versionsnummer), `Configuration` (bietet Zugriff auf die Applikationskonfiguration) und `LibraryManager` (ermöglicht Interaktion mit der eBookbibliothek des Nutzers).

- `\Bookshelf\DataIo`
  Dieser Namespace enthält Klassen, die für I/O-Funktionen jeglicher Art verantwortlich sind. Dazu zählen etwa `DatabaseConnection` (zur Kommunikation mir der Datenbank), `FileManager` (zum Umgang mit Dateien im Dateisystem) und `NetworkConnection` (zum Abrufen von Daten über Netzwerke).

- `\Bookshelf\DataType`
  Dieser Namespace enthält alle Datentypen, die von bookshelf-server verwendet werden. Dazu zählen `Book` (enthält sowohl Informationen zur Datei auf dem Dateisystem als auch dazugehörige Metadaten), `BookMetadata` (enthält Metadaten zu Büchern) und `ExternalApiResult` (speichert das Ergebnis der Anfrage an eine externe API, wie etwa die Google Books API).

- `\Bookshelf\ExternalApi`
  Dieser Namespace enthält Klassen, die den Zugriff auf externe APIs ermöglichen (z.B. zum Abrufen von Metadaten). So ist beispielsweise `GoogleBooksApiRequest` für die Google Books API implementiert. Diese Klasse erbt von der generischen `ExternalApiRequest`, sodass problemlos weitere APIs verwendet werden können (s. Kapitel 7.2.1).

5 PHP Framework Interoperability Group: *PSR-4: Autoloader*. 2013. URL: https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md (Abruf am 14. Juni 2015).

6 vgl. hierzu auch https://git.my-server.in/bookshelf/bookshelf-server/wikis/Namespaces

- `\Bookshelf\Utility`
  Dieser Namespace enthält Helferklassen. Es sind `ErrorHandler` zum Verarbeiten von auftretenden Fehlern und `User` zum Authentifizieren von Benutzern implementiert.

2.4 Datenbankstruktur

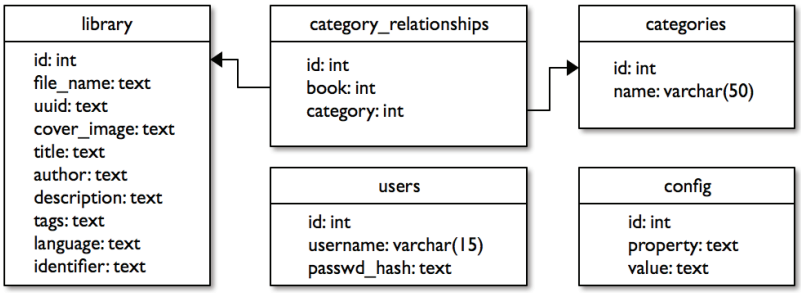

Abbildung 2.2: Datenbankmodell von bookshelf-server

Abbildung 2.2 zeigt die Datenbankstruktur, die von bookshelf-server verwendet wird. Sie besteht zunächst aus drei – zur Zeit – voneinander unabhängigen Systemen:

Die Tabelle `config` ist losgelöst von der Bibliothek eine reine Zuweisung von Key-Value-Paaren. Sie bietet so hohe Flexibilität, die sich vom Code aus gestalten lässt. Außerdem wird hier die Abwärtskompatibilität der Datenbank sichergestellt.

Obwohl von der Tabelle `users` eine Abhängigkeit zur Bibliothek zu erwarten wäre (etwa Eigentümerregelungen der Bücher), ist dies in der aktuellen Version von bookshelf-server nicht der Fall. Die Nutzer dienen einzig der Zugangsbeschränkung auf die gesamte Bibliothek, ein Rechtesystem existiert noch nicht. Dementsprechend simpel ist der Aufbau der Datenbank gestaltet. In zukünftigen Versionen würden – auch zur Wahrung der Kompatibilität – nur neue Beziehungstabellen eingefügt.

Die Kerndaten von Bookshelf, nämlich die Bibliotheksdaten, sind in einem System aus Daten- und Beziehungstabellen gespeichert. Die Haupttabelle `library` enthält die Einträge der einzelnen Bücher und speichert ihre Attribute als einzelne Felder in die Datenbank. Dabei kommt fast immer der Datentyp `text` zur Verwendung, da in diesem Fall große Va-

riationen etwa in der Länge des Titels auftreten können und `text` diese aufgrund seiner flexiblen Natur optimal aufnimmt. Die Nachteile auf Seiten der Ladeperformance und Speichergröße müssen hier akzeptiert werden.

Einige Felder des Datentyp `Books` lassen sich allerdings nicht direkt in die Datenbank speichern. Die Arrays für Tags und Kategorien sind keine in MySQL verfügbaren Standarddatentypen. Um die Tags zu speichern, werden sie also aus einem Array in einen durch Kommata verketteten String konvertiert und als `text` in die Datenbank geschrieben. Beim Auslesen kann das gleiche Verfahren in die andere Richtung angewandt werden. Verloren geht auf diese Weise dann zwar die Möglichkeit, die Bibliothek nach Tags zu strukturieren. Da diese jedoch auf freien, unvorhersehbaren Nutzereingaben basieren, kann auf diese Funktionalität zugunsten der Einfachheit verzichtet werden.

Anders sieht es bei den Kategorien aus. Sie sind häufig feste Begriffe und treffen für mehrere Bücher zu. Ihr Zweck ist es, die Bibliothek zu strukturieren. Aus diesem Grund lässt sich die bei den Tags verwendete Methode nicht anwenden. Hier werden bidirektionale Beziehungen benötigt. Um dies umzusetzen, werden bei bookshelf-server die Kategorien zunächst in eine weitere Tabelle, `categories`, gespeichert und ihnen so eine `id` zugewiesen. Um diese getrennten Daten in Beziehung zu setzen, kommt eine Beziehungstabelle zum Einsatz. In ihr sind alle Verbindungen von Büchern zu Kategorien in Form von Paaren ihrer jeweiligen `ids` gespeichert. So lassen sich einfach und skalierbar Bücher zu Katagorien zuordnen und vice versa. Diese Möglichkeit besitzt nur eine geringe Fehleranfälligkeit und ist vor allem wandlungsfähig: Möchte man eine Kategorie von einem Buch entfernen, genügt es, den Datenbankeintrag in der Beziehungstabelle zu löschen:

```
DELETE FROM category_relationships WHERE book = {book_id} AND
category = {category_id};
```

Listing 2.1: SQL zum Löschen einer Kategorie eines Buches

Sollen das Buch oder die Kategorie gelöscht werden, müssen dann, dem gleichen Schema folgend, alle Beziehungseinträge für das jeweilige Objekt entfernt werden. Dies erleichtert den Löschungsprozess ernorm. Würde die Methode der Tags verwendet, würde ein Löschen der Katego-

rie bedeuten, alle betreffenden Bucheinträge zu ändern. So aber genügen zwei Datenbankanfragen.

Auch das Hinzufügen lässt sich auf diese Weise durch wenig SQL-Code stark vereinfachen und somit die Datenoperationen an den Datenbankserver auslasten, anstatt sie im für diese Operationen ungeeigneten PHP-Code durchzuführen. Dabei müssen existierende Kategorien gefunden und nicht existente hinzugefügt werden. Dies lässt sich in einem Query lösen:

```
INSERT INTO categories (name) SELECT * FROM  (SELECT
'{category_name}') AS tmp WHERE NOT EXISTS (SELECT name FROM
categories WHERE name = '{category_name}') LIMIT 1;
INSERT INTO category_relationships (book, category) SELECT @
book_id, categories.id FROM categories WHERE categories.name
= '{category_name}';
```

Listing 2.2: SQL zum Hinzufügen einer Kategorie

Der Query fügt eine Kategorie hinzu, falls sie noch nicht in `categories` vorliegt und erstellt schließlich eine Beziehung dieser Kategorie zum zugehörigen Buch mit der `id @book_id` in `categories_realtionship`. `@book_id` ist dabei die `id` des gerade eingefügten Buchs und wird über `LAST_INSERT_ID()` bestimmt.

Um die Kategorien auslesen zu können, werden die drei Tabellen mithilfe eines `JOIN`s verbunden. Dabei werden „Schnittmengen“ gebildet und an die aktuelle Antwort angehängt. Abbildung 2.3 illustriert dieses Vorgehen. Die so ergänzten Spalten werden dann mittels `GROUP_CONCAT()` zu einem kommaseparierten, menschenlesbaren String verbunden:

```
SELECT library.*, GROUP_CONCAT(categories.name) AS categories
FROM library
LEFT JOIN category_relationships ON category_relationships.
book = library.id
LEFT JOIN categories ON category_relationships.category =
categories.id
WHERE library.id = {book_id}
```

Listing 2.3: SQL zum Auslesen der Kategorien eines Buches

Die Datenbank basiert also insgesamt auf einem hochskalierbaren, flexiblen, dafür aber performanceschwachen und speicherintensiven System. Dieses ist für die Größe normaler Bibliotheken optimal.

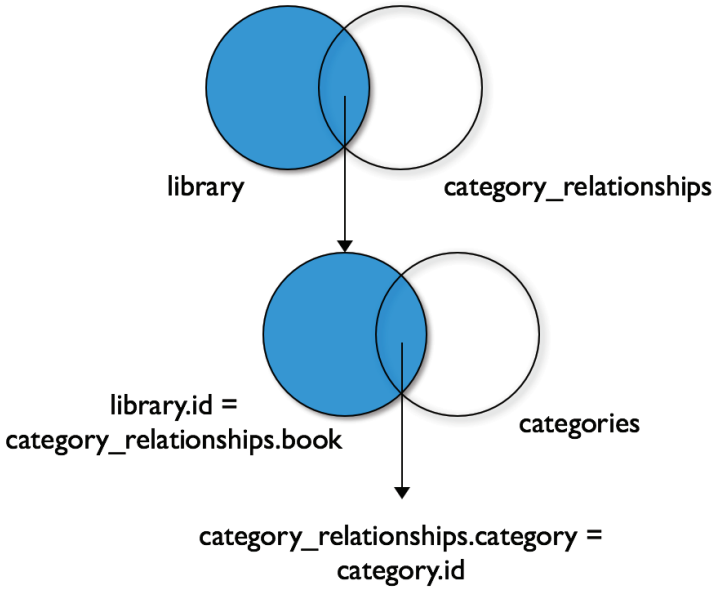

Abbildung 2.3: JOIN-Modell von bookshelf-server

2.5 External API

Bei der externen API handelt es sich praktisch um einen Proxy, welcher unabhängigen Clients die Methoden der internen API zur Verfügung stellt und somit den Zugriff auf die eBookbibliothek des Benutzers erlaubt.

Um API-Anfragen zu vereinheitlichen und eine klare Schnittstelle anzubieten, haben wir uns auf einige Konventionen im Bezug auf die externe API geeinigt.
So soll es sich bei fast allen API-Anfragen um HTTP POST Requests handeln. Lediglich der Download von eBooks wird über GET Requests erledigt, da hier ein Verweis auf eine konkrete Ressource besteht.
Bei dem `action`-Parameter (s. unten) wird nicht auf Groß- und Kleinschreibung geachtet.
Und schließlich antwortet die API auf Fehler mit einem maschinenlesbaren Statuscode und einer menschenlesbaren Beschreibung des Fehlers. Darüber hinaus kann ein HTTP-Statuscode (jedoch nur die folgenden: 200 (OK), 400 (Bad Request), 401 (Unauthorized), 500 (Internal Server

Error), 501 (Not Implemented) or 503 (Service Unavailable))[7] gesendet werden.

Um den Benutzer zu identifizieren, haben wir uns entschieden, auf HTTP Basic Authentication zurückzugreifen.[8]
In der momentanen Implementierung gibt es lediglich eine einzige eBookbibliothek, auf welche sämtliche Nutzer Zugriff haben. Dies werden wir zukünftlich natürlich noch erweitern, um jedem Benutzer eine eigene Bibliothek zuzuweisen.

Je nach angefragter `action` erwartet die API unterschiedliche Parameter. Dabei kann es sich um die folgenden handeln:

Bezeichnung	**Beschreibung**
`action`	s. Tabelle 2.2; z.B. `addBook`
`meta_author`	Autorname für die Metadaten; z.B. `Gabriele Altpeter`
`meta_cover_image`	Coverbild für die Metadaten; base64-enkodiert
`meta_description`	Beschreibung für die Metadaten
`meta_identifier`	ISBN-13 für die Metadaten; z.B. `9783945748008`
`meta_language`	Sprache für die Metadaten, gemäß ISO 639-1; z.B. `en`
`meta_tags`	Tags für die Metadaten; z.B. `cookies, christmas, baking`
`meta_title`	Titel für die Metadaten; z.B. `Homemade German Plätzchen: And Other Christmas Cookies`

7 Roy Thomas Field et al.: *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. Juni 1999, S. 57–71. URL: https://www.ietf.org/rfc/rfc2616.txt.

8 Zur Auswahl standen neben HTTP Basic Authentication auch OAuth 1.0 und OAuth2.0, wir sind jedoch zu dem Schluss gekommen, dass es sich bei ersterem (bei erzwungener TLS-Verschlüsselung) um die effizienteste Variante handelt. Es ist allerdings durchaus denkbar, dass sich diese Entscheidung in der Zukunft ändern wird. (vgl. hierzu auch Diskussion unter https://git.my-server.in/bookshelf/bookshelf-server/issues/33)

Bezeichnung	Beschreibung
`file`	Eine Datei zum Hochladen, s. PHP `$_FILES`
`id`	Die ID des Buches, die in der Datenbank verwendet wird; z.B. `42`

Tabelle 2.1: API-Parameter von bookshelf-server

Die API stellt die in Tabelle 2.2 erläuterten `actions` zur Verfügung:

Action	Benötigte Parameter	Rückgabewert
`addBook`	`file`	ID des hochgeladenen Buches
`deleteBook`	`id`	
`getBookMeta`	`id`	Metadaten des Buches
`downloadBook`	`id`	Tatsächliche Buchdatei
`searchBook`	`field`, `query`	ID des Buches, welches der `query` entspricht
`listBooks`		Metadaten, UUID, ursprünglicher Dateiname und -format und ID aller Bücher in der Bibliothek
`updateBook`	`id`, beliebig viele `meta_*`, die aktualisiert werden sollen	

Tabelle 2.2: `actions` *von bookshelf-server*

Wie erwähnt, antwortet die API auf Fehler u.a. mit einem Statuscode. Sofern möglich folgen diese den HTTP Status Codes. Um Verwirrung vorzubeugen, sind unsere eigenen Statuscodes stets im Format 6xx. Es sind die Tabelle 2.3 beschriebenen in Verwendung.

Statuscode	Beschreibung	HTTP-Statuscode
`200`	OK.	`200`
`401`	Not authenticated.	`401`

Statuscode	Beschreibung	HTTP-Statuscode
501	Feature not yet implemented.	501
601	Invalid action.	400
602	Missing parameters ({list of parameters}).	400
603	User not authorized to perform action.	401

Tabelle 2.3: Statuscodes der API von bookshelf-server

Als Datenformat für die zurückgegebenen Daten dient die Strukturspra-che JSON, die den REST-Anforderungen entspricht und vor allem in vielen Sprachen schnell und einfach zu de- und encodieren ist. Die Rückgabe folgt dabei eine universellen Schema, das alle Antworten einhalten. Die Daten sind von einem JSON-Object umschlossen, das neben ihnen aus noch `status_code` und `status` enthält. So ist die Antwort für die Clients zu Beginn immer gleich und etwaige Probleme des Servers können sofort abgefangen werden. Das in `result` aufgeführte Ergebnis besteht nun wieder entweder aus einem Array oder Objekt, abhängig von der angefragten `action`. Aufgabe der Clients ist dann, die angefragte `action` zu speichern, um auf die unterschiedlichen `results` verschieden reagieren zu können. Der Aufbau ist in Listing 2.4 beispielhaft aufgeführt:

```
{
    "status_code": 200,
    "status": "OK.",
    "result": {
        "cover_image": "[…]",
        "title": "Homemade German Plätzchen - And Other
Christmas Cookies",
        "author": "Gabriele Altpeter",
        "description": "<p>Christmas is the time we call the
most beautiful time of the year. The book Homemade German
PlÃ¤tzchen: And Other Christmas Cookies wants to invite you
to go on a journey through a happy and joyful Pre-Christmas
and Christmas period. Allow yourself to take the time to bake
these homemade Plätzchen and smell the scent of the festive
spices.</p>",
        "tags": [
            "baking",
            "cooking",
            "plätzchen",
            "cookies",
```

```
            "german"
        ],
        "categories": [
            "Non-fiction"
        ],
        "language": "en",
        "identifier": "9783945748015"
    }
}
```

Listing 2.4: Beispielhafte JSON-Rückgabe der `getBookMeta`*-API-*`action`

Im Fall eines Fehlers ist der äußere Aufbau, wie erwähnt, gleich, das Ergebnis jedoch leer. Ein Beispiel hierzu ist in Listing Listing 2.5 zu finden.

```
{
    "status_code": 601,
    "status": "Invalid action.",
    "result": []
}
```

Listing 2.5: Beispielhafte JSON-Rückgabe für eine ungültige `action`

3. bookshelf-server

3.1 Internal API

Die interne API besteht, da sie nur Funktionen und Datenstrukturen definiert, ausschließlich aus Klassen, die sich, wie bereits in Abschnitt 2.3 ausgeführt, in mehrere Namespaces aufteilen. Neben der Funktion der Klassen im ursprünglichen Sinne der objektorientieren Programmierung, übernehmen die Klassen der internen API auch eine Stellung als reine funktionale Zusammenfassungen bestimmter statischer Funktionen oder Konstanten.

Abbildung 3.1 zeigt eine Übersicht der Klassen von bookshelf-server.

Damit die interne API verwendet werden kann, muss in jeder Datei, die auf die API zugreifen möchte, der Autoloader eingebunden werden:

```
require_once __DIR__ . '/../lib/vendor/autoload.php';
```

Listing 3.1: Einbinden des Autoloaders

Alle Dateien der internen API müssen, unabhängig davon, dass sie bereits nach der gleichbenannten Ordnerstruktur in die Namespaces geteilt sind, ihren Namespace deklarieren. Dies passiert in einer einfachen Zeile[9]:

```
namespace Bookshelf\Core;
```

Listing 3.2: Deklarieren des Namespaces

9 PHP Framework Interoperability Group (2013).

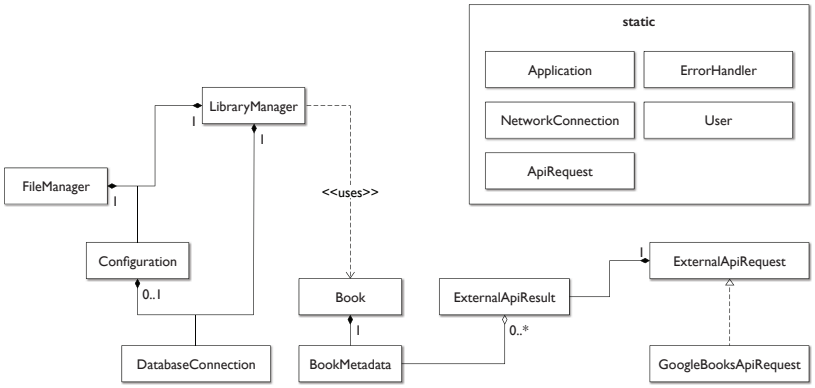

Abbildung 3.1: Klassenübersicht für bookshelf-server

3.1.1 Core

Der Namespace `\Bookshelf\Core` enthält die Klassen:

- `Application`
  Diese Klasse dient als reiner „Behälter" für Konstanten, die die ganze Anwendung betreffen. Zu finden ist hier etwa die Versionsangabe der bookshelf-server-Anwendung. Diese besteht aus einem maschinenlesbaren Versionscode und einem menschenlesbaren Versionstext:[10]

```
const VERSION_TEXT = '0.0.1a';
const VERSION_CODE = 1;
```

Listing 3.3: Das Versionsschema

  Darüber hinaus werden Konstanten für das Root- und Bibliotheksverzeichnis gesetzt, auf welche andere Klassen zugreifen können.

- `Configuration`
  Diese Klasse kümmert sich um die Einstellungen zu bookshelf-server. Diese können entweder in der Datenbank oder in der plattformspezifischen `config.ini` liegen. Einzelne Einstellungen können dann mithilfe von Gettermethoden ausgelesen werden. Da auch `\Bookshelf\DataIo\DatabaseConnection` eine Instanz von `Configuration` verwendet, um die Datenbankzugangsdaten aus der `config.ini` zu

10 vgl. hierzu auch https://git.my-server.in/bookshelf/bookshelf-server/issues/6

erhalten, und `Configuration` ja umgekehrt auch auf die Datenbank durch `\Bookshelf\DataIo\DatabaseConnection` zugreift, entsteht ein Zirkelbezug. Damit dieser Fehler nicht auftritt, kann man dem `Configuration`-Konstruktor einen Parameter `$enable_db` übergeben, der falls auf `false` gesetzt, die Einbindung der Datenbank verhindert, sodass `\Bookshelf\DataIo\DatabaseConnection` diese Klasse instanzieren kann, ohne sich selbst aufzurufen.

Ebenso besteht die Möglichkeit, eine bereits instanzierte Datenbankverbindung zu übergeben.

- `LibraryManager`
  Diese Klasse bildet ganz eindeutig, wie der Namespace ja bereits suggeriert, ein Herzstück der Anwendung. Sie enthält Methoden, die die Verwaltung der Bibliothek ermöglichen, wie etwa `LibraryManager()->addBook()`. Die meisten Methoden sind eine reine Zusammenfügung von Methoden anderer Klassen, sodass `LibraryManager` die Daten lediglich für sie aufbereitet, oder ihrer Rückgaben ordnet. Außerdem stellt `LibraryManager` eine Umgebung mit jeweils einer `Configuration` und `\Bookshelf\DataIo\DatabaseConnection` Instanz zur Verfügung. Als Beispiel hier die Methode `LibraryManager()->deleteBook()` (s. Listing 3.4), sie verbindet zwei Befehle zur Löschung verschiedener Datentypen an einem Ort:

```
public function deleteBook($id) {
    $this->database_connection->deleteBook($id);
    DataIo\FileManager::deleteBook($id);
}
```

Listing 3.4: deleteBook() als Beispiel einer Methode, die mehrere Befehle verbindet

3.1.2 DataType

Der Namespace `\Bookshelf\DataType` beschreibt, wie der Name verrät, verschiedene Datentypen. Die Klassen verhalten sich hier zum Teil analog zum in C/C++ vorkommenden `struct`, das in PHP nicht existiert. Manche Klassen bieten aber über ihre Strukturfunktion hinaus noch weitere Methoden, die die standardisierte Ausgabe in verschiedene Ausgabeformate ermöglichen. Folgende Datentypen sind in diesem Namespace verfügbar:

- `Book`
  Dieser Datentyp ergibt im Hinblick auf die Funktion der Anwendung viel Sinn. Er besteht im Wesentlichen aus der Zusammenführung der Metadaten und der Buchdatei. Dafür stellt er einige öffentliche Attribute bereit, darunter die ID, der ursprüngliche Dateiname, `Book()->metadata`, mit dem Datentyp `BookMetadata`, und `Book()->uuid`, einem String, der den eindeutigen Dateinamen, die UUID, enthält. Außerdem bietet `Book` noch zwei Methoden: Eine, die aus dem ursprünglichen Dateinamen einen Suchquery für externe APIs generiert, und eine, die es ermöglicht, die Buchdatei unter ihrem ursprünglichen Namen auszugeben.

- `BookMetadata`
  Diese Klasse ist tatsächlich ein reines `struct`, das bestimmte Ausgabefunktionen definiert hat: Alle Metadaten sind in ihr jeweiliges Attribut gespeichert und können über die passende Methode entweder als eindimensionaler, assoziativer Array oder Zeile einer HTML Tabelle ausgegeben werden. In letzterer Methode besteht noch die Möglichkeit, über einen Parameter eine Spalte mit Auswahlfeldern anzeigen zu lassen, um die Zeile in ein Formular einbauen zu können.

- `ExternalApiResult`
  Diese Klasse dient zur Vereinheitlichung der Suchergebnisse projektfremder APIs. Da sich diese in Ausgabeformat und -struktur stark unterscheiden wird so gesichert, dass aller Klassen der internen API mit dem gleichen Format arbeiten und so Fehlerquellen minimiert. Ein `ExternalApiResult` wird dynamisch erweitert, indem über die Methode `ExternalApiResult()->addMetadata()` neue Ergebnisse mit einem eindeutigen Identifer zum Attribut `ExternalApiResult->result_collection` hinzugefügt werden und so ein zweidimensionaler Array entsteht:

```
public function addMetadata($metadata, $api_identifier) {
    $result = array('metadata' => $metadata, 'api_identifier'
=> $api_identifier);
    array_push($this->result_collection, $result);
}
```

Listing 3.5: Dynamisches Hinzufügen neuer Metadaten zu einem `ExternalApiResult`

Die Klasse bietet außerdem Methoden zur Ausgabe als String, HTML Tabelle und zur direkten Ausgabe des `ExternalApiResult->result_collection`-Arrays.

3.1.3 DataIo

Der Namespace `DataIo` stellt Klassen und Methoden zur Verfügung, die zur Kommunikation mit Data I/O Devices dienen. Er enthält deshalb folgende Klassen:

- `DatabaseConnection`
  Die `DatabaseConnection` kapselt alle Anfragen an die Datenbank. Einerseits zum Schutz vor fehlerhaften Anfragen, andererseits aber auch zur einfacheren Verwendung der Datenbank haben wir uns entschieden, niemals andere Klassen als die `DatabaseConnection` direkt auf die MySQL-Datenbank zu greifen zu lassen. Dementsprechend sind alle notwendigen Datenbankoperationen hier implementiert.

  So gibt es beispielsweise die `dumpLibraryData()`-Methode, die sämtliche in der Bibliothek befindlichen Bücher zurückgibt. Sie wird etwa zum Auflisten aller Bücher verwendet:

```
public function dumpLibraryData() {
    if($result = $this->mysqli->query("SELECT * FROM
library")) return $this->fetch_all($result);
}
```

Listing 3.6: Dumpen der Bibliotheksdaten

- `FileManager`
  Der `FileManager` ist dafür zuständig, Daten im Dateisystem zu lesen und schreiben. Darüber hinaus wird hier bei Bedarf auch eine UUID zur Speicherung der Bücher generiert. Dabei handelt es sich laut Spezifikation um einen weltweit einmaligen Identifier, sodass es in der Theorie nicht zum Überschreiben kommen kann. Dennoch haben wir uns entschieden, vor dem tatsächlichen Schreiben auf dem Dateisystem zu prüfen, ob eine Datei mit gleichem Namen bereits existiert:[11]

11 Paul J. Leach et al.: A *Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. Juli 2005, S. 5–15. URL: http://www.ietf.org/rfc/rfc4122.txt.

```
$possible_duplicate = glob($this->config->getLibraryDir() .
$uuid . '.*');

return count($possible_duplicate) > 0 ?
$this->generateUuid() : $uuid;
```

Listing 3.7: Überprüfung auf mögliche (wenn auch sehr unwahrscheinliche) UUID-Duplikate

- `NetworkConnection`
  Die `NetworkConnection` ist für sämtliche Verbindungen zu Netzwerken verwenden. Aktuell ist lediglich eine Methode für curl requests implementiert, es ist jedoch vorstellbar, diese Klasse auch für andere Methoden zu erweitern.
  Wichtig ist die Encapsulation von Netzwerkzugriffen insbesondere, um für Server einheitlich aussehende Anfragen zu erstellen, sodass bookshelf-server in den Serverlogs klar erkannt werden kann. Dazu setzen wir beispielsweise einen eindeutigen User Agent String, der u.a. auch die Versionsnummer enthält:

```
curl_setopt($ch, CURLOPT_USERAGENT, 'Bookshelf/' .
Application::VERSION_TEXT . ' (gzip; +http://getbookshelf.
org)');
```

Listing 3.8: Setzen des User Agent Strings

  Die Klasse erlaubt uns jedoch auch etwa, Optionen bezüglich der von bookshelf-server akzeptierten Komprimierungsencodings zu spezfizieren:

```
curl_setopt($ch, CURLOPT_ENCODING, 'gzip');
```

Listing 3.9: Setzen des akzeptierten Komprimierungsencodings

3.1.4 ExternalApi

Der Namespace `ExternalApi` bietet die notwendigen Klassen und Methoden, um mit externen APIs (wie etwa der Google Books API) zu kommunizieren und die entsprechenden Ergebnisse zurückzugeben.
Da die grundlegende Struktur von Anfragen bei allen APIs gleich ist, haben wir zunächst eine generische Methode implementiert, von welcher dann die spezifischen Methoden erben. Dementsprechend existieren in diesem Namespace folgende Klassen:

- `ExternalApiRequest`
  Bei der `ExternalApiRequest` handelt es sich um die beschriebene allgemeine (bzw. `abstract`) Klasse. Sie implementiert keine konkreten Methoden, sondern gibt lediglich ein Grundgerüst für Childklassen.

  So beschreibt sie etwa eine allgemeine `request()`-Methode, die von allen erbenden Klassen implementiert werden muss. Obwohl oftmals für diese spezifischere Requestarten vorliegen, dient diese Methode als allgemeines Fallback.

  Auch müssen diese `getBookByIdentifier` implementieren, damit die Metadaten für ein vorher vom Benutzer ausgewähltes Datenset gefunden werden können. Dies ist etwa beim Hochladen von Büchern über die Web App nötig.

- `GoogleBooksApiRequest`
  `GoogleBooksApiRequest` ist momentan die einzige spezifisch implementierte externe API. Das liegt daran, dass sie für die aktuelle Entwicklungsphase mehr als ausreichende Ergebnisse liefert. Dennoch werden später natürlich noch weitere APIs implementiert werden.

3.1.5 Utility

Der Namespace `\Bookshelf\Utility` stellt Werkzeuge zur Verfügung und fungiert als der Namespace, der keine funktionskritischen Bestandteile enthält. Diese Klassen gehören dazu:

- `ErrorHandler`
  Der `ErrorHandler` ist im Grunde nur eine sehr einfache Klasse, die einzig statischen Funktionen eine zusammenfassende Kategorie bietet. Die Funktion `ErrorHandler::throwError()` schreibt einen Fehler in die `$_SESSION` Variable und alle in dieser Variable gespeicherten Fehler auf Aufruf der Funktion `ErrorHandler::displayErros()` als einfaches HTML flushed. Ein Fehler besteht aus `'message'`, der Fehlernachricht, und `'error_level'`, einer Zahl, die die Schwere des Fehlers beschreibt.

- `ErrorLevel`
  Diese Klasse ist nun schon nicht mehr analog zu einem `struct`, sondern stellt nur noch einen reinen `ENUM` dar. In ihr sind verschiede-

ne Konstanten, die das zu `ErrorHandler::throwError()` gehörende `'error_level'` festlegen.

- `User`
  Auch die Klasse `User` kategorisiert statische Funktionen, die Authentifizierungsfunktionen übernehmen. Eine Funktion überprüft mit einer schlichten Datenbankabfrage, ob Zugangsdaten valide sind, die andere gibt schlicht mittels `print` Befehl ein statisches Loginformular zurück, das in einem String gespeichert ist.

3.2 External API

Die externe API ist ausschließlich in der Datei `./api/index.php` definiert. So wird gewährleistet, dass auf der einen Seite eine kurze und einfache URL zum Erreichen der API (`<serverdomain>.<tld>/api`) verwendet werden kann, zum anderen wird verhindert, dass es zum versehentlichen Aufruf weiterer Dateien anderer Verzeichnisse kommt. Dadurch, dass die eigentlichen Prozesse in der internen API ablaufen und dort auch definiert sind, sodass die externe API wie ein Proxy funktioniert, muss die externe API entsprechende Funktionen lediglich aufrufen und notwendige Klassen importieren (siehe auch Kapitel 2.5).

Um auszuwählen, welche Funktionen ausgeführt werden müssen, wird mittels eines switch statements der Wert der Variablen `$request->action` mit den verfügbaren `actions` verglichen (siehe Tabelle 2.2). Der Code verwendet hier statt der tatsächlichen von PHP übergebenen Variable `$_POST['action']` nur ein Attribut einer Instanz der Klasse `\Bookshelf\DataIo\ApiRequest`. Diese kann bereits einfache Umwandlungen der Daten vornehmen (etwa den `action`-Parameter case-insensitive machen, s. Listing 3.10), und klärt die Datenstruktur eines Requests unzweifelhaft.

```
$this->action = strtolower($post['action']);
```

Listing 3.10: Ignorieren von Groß- und Kleinschreibung beim `action`*-Parameter*

Um die angeforderten Daten zurückzugeben, werden die Rückgabewerte der ausgeführten Funktionen in ein Array `$result` gespeichert. Um die in Kapitel 2.5 angesprochene uniforme äußere Struktur zu erzeugen, wird dieses Array in ein zweites, `$api_reply`, eingebettet, welches außerdem die Stausinformationen enthält. Dieses wird dann am Ende in Form

von JSON maschinenlesbar ausgeben, sodass die unabhängigen Clients darauf einfach zugreifen können:

```
echo json_encode($api_reply);
```

Listing 3.11: Enkodieren des Ergebnisarrays als JSON

Auch Fehler werden in dieses Array eingefügt. Diese bestehen zum einen aus dem Errorcode (`"error_code"`), einer maschinenlesbaren Zahl, die die Meldung eindeutig identifiziert (s. Tabelle 2.3). Zum anderen enthält ein Fehler die Fehlerbeschreibung (`"error"`), die einen Fehler für einen Menschen lesbar erklärt und eventuelle Fehlerquellen beschreibt:

```
{
    "error_code": 602,
    "error": "Missing parameters (field, query)."
}
```

Listing 3.12: Eine beispielhafte JSON-Rückgabe für einen Fehler (fehlende Parameter)

Zusätzlich dazu wird über einen HTTP Response Code ein Fehler an den Client angezeigt. So wird REST-Kompatibilität gewährleistet.[12]

```
http_response_code(400);
```

Listing 3.13: Zurückgeben eines REST-kompatiblen HTTP Status Codes

3.3 Frontend und Web App

Wie bereits vorher erwähnt, bringt bookshelf-server ein Webfrontend mit, welches gleichzeitig als Beispielimplementation weiterer Clients dient. Das Frontend ermöglicht die Nutzung aller Bookshelf-Features direkt aus dem Browser.

3.3.1 Benutzeroberfläche

Beim Einloggen wird der Benutzer zunächst aufgefordert, seinen Benutzernamen und Passwort einzugeben, um sich zu authentifizieren.

12 Roy Thomas Fielding: *Architectural Styles and the Design of Network-based Software Architectures*. Diss. University of California, Irvine, 2000.

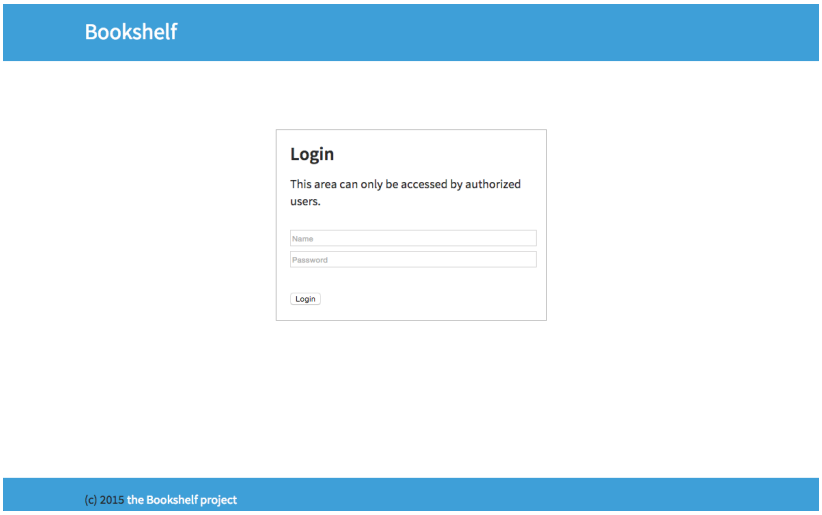

Abbildung 3.2: Loginmaske von bookshelf-server

Anschließend bekommt der Nutzer eine Auflistung aller Bücher in seiner Bibliothek in Form einer Grid View der Cover.

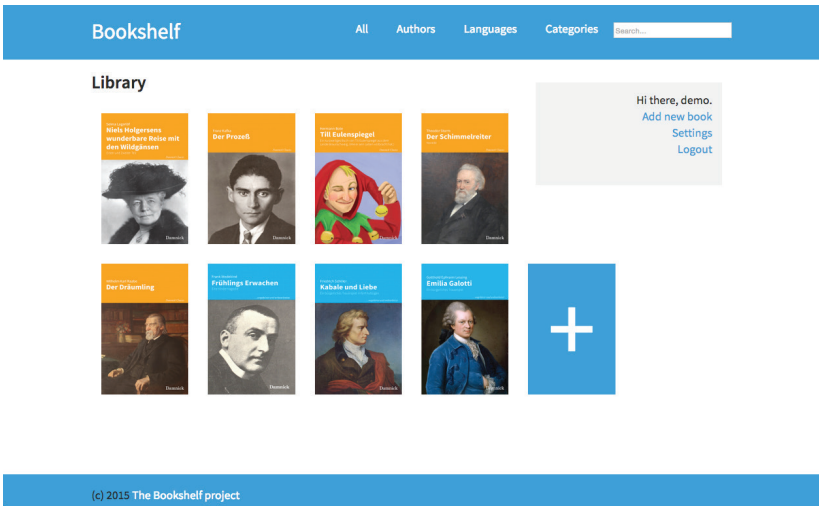

Abbildung 3.3: Grid View der Bibliothek

Er hat die Möglichkeit, einzelne Bücher auszuwählen und sich die jeweiligen Detailseiten anzusehen. Dort bekommt er zum einen die Metadaten des entsprechenden Buches angezeigt und kann zum anderen das

Buch herunterladen, es direkt im Browser lesen (s. Kapitel 3.3.3), die Metadaten bearbeiten oder das Buch löschen.

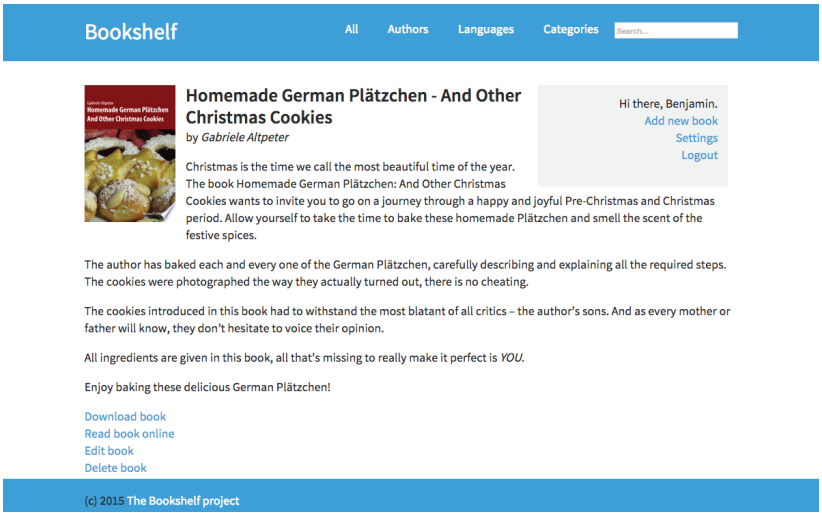

Abbildung 3.4: Detailansicht eines Buches

Die Web App bietet dem Nutzer verschiedene Möglichkeiten, ein bestimmtes Buch zu finden. So gibt es zunächst die Möglichkeit, die Bibliothek nach gewissen Parametern (Autoren, Sprachen, Kategorien) zu sortieren. Weiterhin lässt sich die Suchfunktion (s. Kapitel 3.3.4) nutzen.

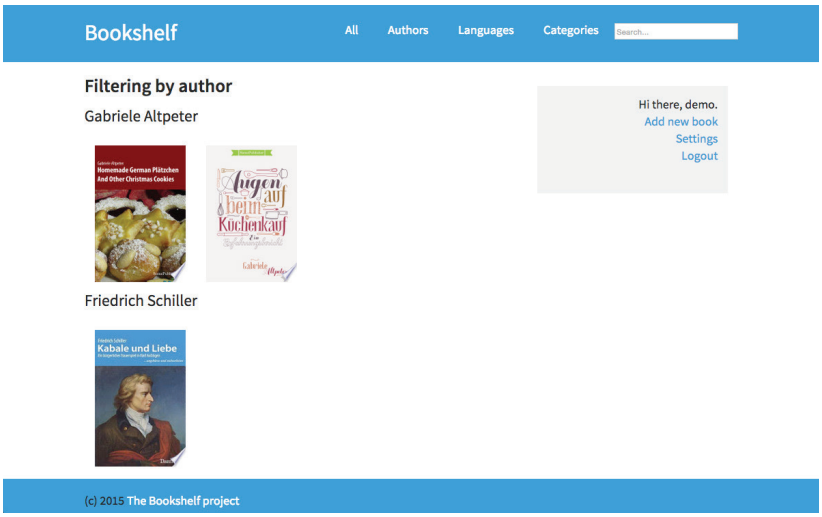

Abbildung 3.5: Sortieren der Bibliothek nach Autoren

3.3.2 Grid View

Aufgrund der Flexibilität der internen API lässt sich ein beliebiges Array von `\Bookshelf\DataType\Book` einfach als Grid View darstellen:

```
foreach($books as $book) {
    echo '<a href="book.php?id=' . $lib_man->getBook('uuid',
$book->uuid, true) . '"><img class="book" src="' .
$book->metadata->cover_image . '"></a>';
}
```

Listing 3.14: Darstellung einer Grid View von Büchern

Diese Objekte werden wie folgt im CSS gestylt:

```
.book {
  height: 200px;
  width: 133.33px;
  margin: 15px;
  display: inline-block;
  vertical-align: top;
  background-color: #999999;
}
```

Listing 3.15: Styling der `book`*-Objekte*

3.3.3 Web Reader

Der Web Client bietet zwar die Möglichkeit, die eBooks herunterzuladen, doch gerade an fremden Endgeräten kann es für den Nutzer durchaus praktisch sein, die Bücher direkt im Browser zu lesen. Deshalb haben wir einen Web Reader implementiert.
Dieser unterstützt aktuell die Vorschau von PDF- und EPUB-Dateien.

Im Hintergrund werden dafür leicht angepasste Versionen von PDF.js (entwickelt von Mozilla) und EPUB.js (entwickelt von Future Press) verwendet. Dabei handelt es sich um freie Softwarelösungen zur Darstellung von eBooks im Browser. Sie bieten eine hohe Flexibilität, sodass selbst eine Darstellung in modernen Mobilbrowsern möglich ist.

In Abhängigkeit vom Buchformat wird entweder der entsprechende Reader eingebunden oder eine Fehlermeldung ausgegeben:

```
$book = $lib_man->getBookById($id);
$file = $config->getLibraryDir() . '/' . $book->uuid .
$book->original_extension;
```

```
$mime_type = finfo_file(finfo_open(FILEINFO_MIME_TYPE),
$file);

switch($mime_type) {
    case 'application/epub+zip':
        include __DIR__ . '/inc/reader-epub/reader.php';
        break;
    case 'application/pdf':
        include __DIR__ . '/inc/reader-pdf/web/viewer.php';
        break;
    default:
        insertHeader();
        echo '<p>Unfortunately, viewing files of that
type online is currently not supported.<br>You can always
<a href="' . $base_url . '/download.php?id=' . $id .
'">download</a> the file and read it on your computer,
though.</p>';
        echo '<p>Go back to the ebook <a href="' . $base_url
. '/book.php?id=' . $id . '">detail page</a>.</p>';
        insertFooter();
}
```

Listing 3.16: Anzeigen des korrekten Web Readers

3.3.4 Suchfunktion

Um das Auffinden bestimmter Bücher zu erleichtern, bietet die Web App eine umfangreiche Suche. Der Nutzer kann entweder einen einfachen Suchstring (z.B. `Homemade German Plätzchen`) oder einen komplexeren Querystring mit den verfügbaren Operatoren eingeben.

In letzterem Fall hat er die Möglichkeit, konkrete Felder in der Suchanfrage abzufragen, die über ein logisches `AND` in der Datenbankabfrage verbunden werden. Ein möglicher Querystring wäre etwa `author: altpeter title: plätzchen`.

Intern wird der Querystring anfangs vom `\Bookshelf\Core\LibraryManager` aufbereitet. Dazu werden zunächst einige Umbenennungen vorgenommen, sodass der Nutzer auch diverse Synonyme für Suchoperatoren verwenden kann:

```
$courtesy_renames = array('tag' => 'tags', 'description' =>
'desc', 'language' => 'lang', 'identifier' => 'isbn');
$query = str_replace(array_keys($courtesy_renames),
$courtesy_renames, $query);
```

Listing 3.17: Umbenennen gewisser Suchfelder

Anschließend wird der String mittels Regular Expressions in einen URL-String umgewandelt, welcher schließlich mithilfe der `parse_str`-Funktion in ein zweidimensionales Array geparst wird:

```
$query = preg_replace('/(author|desc|isbn|lang|tags|title):\
?/i', '&$1=', $query, -1, $count);
parse_str($query, $query_array);
```

Listing 3.18: Parsen des Querystrings

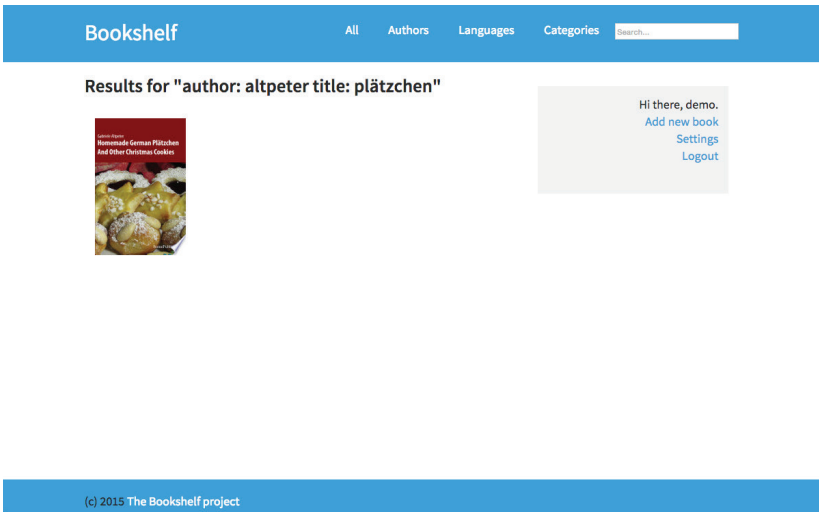

Abbildung 3.6: Suche anhand konkreter Felder

Nach dieser Aufbereitung wird die Anfrage an die `\Bookshelf\DataIo\DatabaseConnection` weitergeleitet. Dort wird nach einer Überprüfung der angefragten Feldern und einem Escaping und Purifying zum Verhindern von SQL Injections die SQL-Query generiert:

```
foreach($query_array as $property => $value) {
    if(in_array($property, DatabaseConnection::$ALLOWED_BOOK_
PROPERTIES, true)) {
        $value = $this->purify($value);
        $value = $this->escape($value);

        // First item does not need an AND
        if ($value === reset($query_array)) {
            $query .= " {$property} LIKE '%{$value}%'";
        }
        else {
            $query .= " AND {$property} LIKE '%{$value}%'";
        }
    }
```

```
    else {
        unset($query_array[$property]);
    }
}
```

Listing 3.19: Bau der SQL-Query

3.3.5 Hinzufügen neuer Bücher

Der Uploadprozess neuer Bücher ist möglichst simpel gestaltet, sodass es dem Benutzer leicht fällt, neue eBooks zu seiner Bibliothek hinzuzufügen. Zunächst muss natürlich die eigentliche eBookdatei auf den Server hochgeladen werden. Dies geschieht über ein einfaches HTML-Formular.

Anschließend können dem Buch noch Metadaten hinzugefügt werden. Dieser Schritt ist zwar theoretisch optional, praktisch ergibt es jedoch wenig Sinn, Bookshelf ohne Metadaten zu nutzen. Wir haben uns dennoch bemüht, auch diesen Vorgang so einfach wie möglich zu machen. Daher bieten wir dem Nutzer die Möglichkeit, die Metadaten direkt aus einer externen API (vgl. Kapitel 3.1.4) zu übernehmen, sodass diese nicht manuell eingegeben werden müssen.
Um dem Nutzer noch mehr Arbeit abzunehmen, versucht bookshelf-server, eine Suchanfrage für die externe API aus dem Dateinamen vorzuschlagen. Diese kann zwar angepasst werden, in vielen Fällen liefert dieser simple Algorithmus jedoch gute Ergebnisse:

```
$strings_to_replace = array('-', '_', '.', ',', ';', ':',
'/', '|', '+');
$query_string = str_replace($strings_to_replace, ' ',
$query_string);
```

Listing 3.20: Vorschlagen einer Suchanfrage

Sollte sich einmal kein passendes Ergebnis in der externen API finden, hat der Benutzer natürlich dennoch die Möglichkeit, selbst manuell die Metadaten für sein Buch zu setzen.

3.3.6 Einstellungen

Die Web App verfügt auch über eine einfache Ansicht der Einstellungen, die durch den Benutzer gesetzt sind. Hierbei werden aus Sicherheitsgründen jedoch lediglich die Werte aus der Datenbank angezeigt, nicht jene,

welche über die `config.ini` gesetzt sind. In dieser Datei befinden sich sensible Einstellungen wie die Zugangsdaten für die Datenbank und der Salt.

3.3.7 Details zur Implementierung

Alle funktionalen Dateien des Frontends binden bestimmte Skripte ein, die für jede Seite verwendet werden. Diese liegen im Ordner `./inc/` und werden von dort mittels `include()` eingebunden.

Eine Aufgabe, die so übernommen wird, ist die der Authentifizierung: Die Datei `./inc/auth.php` übernimmt diese. Wenn die Datei eingebunden ist, wird überprüft, ob ein Nutzer angemeldet ist. Ist dies der Fall, so ist eine entsprechende `$_SESSION`-Variable gesetzt. Diese ist für eine Browsersitzung von allen Skripten des Frontends aus zu erreichen und somit die beste Möglichkeit einer skriptübergreifenden Authentifizierung, die die weitaus kompliziertere und rechtlich problematischere Variante der Cookies übertrifft. Falls eine solche Variable nicht gesetzt ist, so bestehen drei Möglichkeiten:

- Der Nutzer hat sich gerade angemeldet.
  In diesem Fall ist die `$_SESSION`-Variable noch nicht gesetzt, es existieren jedoch `password` und `'name'` im PHP `$_POST`-Array. Um zu überprüfen, ob die eingegebenen Daten zu einem Nutzer gehören, der in der Datenbank mit passendem Passwort eingetragen ist, wird eine entsprechende Funktion der Klasse `\Bookshelf\Utility\User` aufgerufen (s. Listing 3.21). Stimmen die Daten, wird die `$_SESSION` Variable aktualisiert und der Nutzer ist eingeloggt. Andernfalls wird er zum Loginformular weitergeleitet.

```
if(Bookshelf\Utility\User::isAuthenticated($_POST['name'],
$_POST['password'])) {
    $_SESSION['name'] = $_POST['name'];
}
else {
    header('Location: index.php');
}
```

Listing 3.21: Authentifizierung mit der `User`*-Klasse*

- Der Nutzer ist nicht angemeldet.
  Möglicherweise wurde die Seite versehentlich aufgerufen, oder die

Sitzung ist abgelaufen. In diesem Fall wird der Nutzer direkt zum Loginformular geleitet, damit er sich erneut anmelden kann.

- Der Nutzer hat sich mit falschen Daten anzumelden versucht.
  Hat ein Nutzer – unbeabsichtigt oder vorsätzlich – falsche Logindaten angegeben, so wird er nur auf die Startseite weitergeleitet (s. Listing 3.21). So wird aber auch das `$_POST`-Array gelöscht und der Nutzer erscheint, als ob er schlicht noch nicht angemeldet wäre.

Als weitere Dateien werden außerdem `./inc/header.php` und `./inc/footer.php` eingebunden. Sie enthalten die HTML-Anweisungen des `<head>`-Bereichs wie etwa `meta`-Daten oder Skripte und Stylesheets, die für die Seite notwendig ist und den am Ende der Seite anzuzeigenden Inhalt, in diesem Fall den für das Debugging hilfreichen Fehlercache:

```
\Bookshelf\Utility\ErrorHandler::displayErrors(true);
```

Listing 3.22: Ausgeben der Sessionfehler

4. bookshelf-qt

Mit dem Qt-Client bookshelf-qt stellen wir eine beispielhafte Implementierung eines Desktopclients für bookshelf-server zur Verfügung. Dank der Verwendung des Qt-Frameworks ist dieser mit allen gängigen Desktopplattformen kompatibel.

4.1 Infrastruktur

Der Qt-Client verwendet die externe API um mit dem Bookshelf Server zu kommunizieren. Dementsprechend ist es notwendig für ausreichende Infrastruktur zu sorgen, die die Serverkommunikation übernimmt, und die passenden Datentypen zur Verfügung stellt.

4.1.1 Objekthierarchie

Zur Infrastruktur gehören verschiedene Klassen des Projektes bookshelf-qt:

- `Book`
  Diese Klasse repräsentiert denselben Datentyp, der auch in bookshelf-server vorkommt. Er ist einzig eine Strukturierung der zu den Büchern gehörenden Daten. Außerdem besitzt die Klasse die als `static` deklarierte Methode `Book::fromJson(QJsonObject)`, mit deren Hilfe die von dem Server empfangenen Antworten in der Programmstruktur entsprechende Datentypen umgewandelt werden können.

- `BookMetadata`
  Diese Klasse bildet, wie schon `Book` ausschließlich den auch in bookshelf-server verfügbaren Datentyp ab.

- `ApiQuery`
  Um die Anfragen an den Server zu vereinfachen, lassen sich diese in die Datenstruktur von `ApiQuery` einfügen. Die Klasse übernimmt dann die Anpassung der allgemeinen `QtNetwork/QNetworkRequests` an die spezifischen Gegebenheiten der externen API von bookshelf-server. Dabei ist die Klasse sehr flexibel und kann alle Serveranfragen an die externe API umfassen.

- `WebApiAdapter`
  Instanzen von `WebApiAdapter` handeln als Workerobjekt und übernehmen die Kommunikation mit dem Server über `QtNetwork/QNetworkAccessManager` sowie die Umwandlung der Serverantworten in lokale Datentypen.

Die gesamte Objekthierarchie wird in Abbildung 4.1 dargestellt:

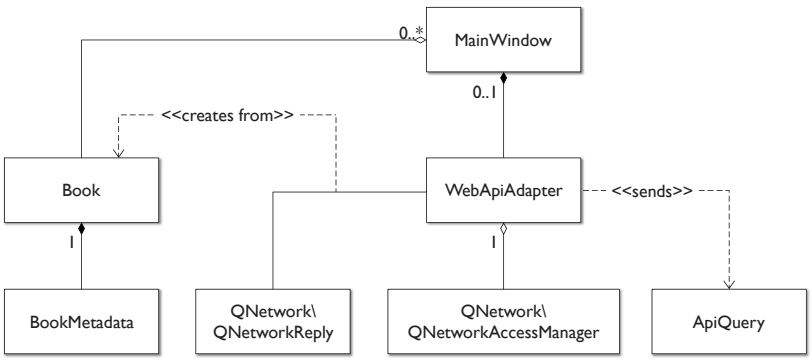

Abbildung 4.1: UML der Objekthierarchie von bookshelf-qt

4.1.2 Threading

Da Kommunikation über das Internet, vor allem das Warten auf Serverantworten, enorm arbeitsaufwändig sind, ist es zwingend notwendig, diese in einen zweiten Thread auszulagern. Andernfalls wäre die Event Loop des Mainthreads belegt und die Benutzeroberfläche könnte nicht mehr auf den Nutzer reagieren, die Applikation wäre „abgestürzt".

Qt bietet als Plattform diverse Möglichkeiten, Aufgaben in einen neuen Thread auszulagern. Allerdings ist die wichtigste Klassen in Qt, von der nahezu alle weiteren Klassen erben, `QObject`, nicht thread-save. Dies bringt also die Herausforderung mit sich, Objekte im richtigen Thread zu initalisieren und ggf. in einen neuen Thread zu verschieben. Um die Thread-Organisation einfach zu halten, werden für die Kommunikation mit dem Server nicht immer neue Threads erstellt, sondern nur einer mit einer Event Loop, die über die meiste Zeit auf Eingaben wartet. Qt bringt diese in seiner Klasse `QtCore/QThread` von Anfang an mit. Nachdem die Event Loop von `QThread` am Ende seiner `QThread::run()` Methode gestartet wurde, „leben" alle dem Thread gehörenden Objekte in ihm. Dazu gehört auch das Workerobjekt `webApiAdapter` vom Typ `WebApiAdapter`. Es kann auf seinen Slots Anfragen empfangen, die dann im neuen Thread ausgeführt werden, bis die Event Loop durch Aufrufen des Slots `QThread::quit()` beendet wird. Deshalb wird es nach der Initialisierung an den Thread übergeben:

```
QThread *workerThread = new QThread();
WebApiAdapter *webApiAdapter = new WebApiAdapter();
webApiAdapter->moveToThread(workerThread);
webApiAdapter->initialize();
```

Listing 4.1: Transferieren des Worker-Objekts zum Thread

So ist es dann möglich, zu jeder Zeit dem Adapter über den Slot `WebApiAdapter::sendQuery()` einen `ApiQuery` zu übergeben, der diesen dann bearbeitet. Arbeitet der Thread gerade an einer Anfrage, so wird die neue Anfrage einfach in eine Queue hinzugefügt und bearbeitet, sobald neue Ressourcen verfügbar sind.

Da der von `WebApiAdapter` eingesetzte `QNetworkAccessManager` an einen bestimmten Kontext gebunden ist, muss er im Threadkontext des zweiten Threads initialisiert werden. Geschähe die Initialisierung im Konstruktor, welcher vom Hautthread aufgerufen wird, so wäre der `QNetworkAccessManager` im falschen Thread instanziert und folglich im parallelen Thread nicht zu benutzen. Deshalb besitzt `WebApiAdpater` außerdem den Slot `WebApiAdapter::initialize()`, welcher nach auch nach Aufruf aus dem Hautthread im richtigen Thread ausgeführt wird und deshalb die Instanzierung des `QNetworkAccessManager` übernehmen kann. Der Slot ist also vor der ersten Anfrage, aber nach dem Verschieben des `WebApiAdapter` Objektes, auszuführen, wie auch in Listing 4.1 zu erkennen.

Ist die Anfrage beendet, so gibt `webApiAdapter` je nach Ergebnis verschiedene Signale ab:

- `WebApiAdapter::done()` wird emittiert, sofern die Serveranfrage fehlerfrei und erfolgreich ablaufen konnte. Als Ladung transportiert das Signal dabei je nach Art der vorhergegangenen Anfrage – und insbesondere deren `action` – die verschiedenen Ergebnisse. So z.B. einen `QVector<Book>`, also einer Liste aus Büchern, für eine Anfrage mit der `action listBooks`.

- `WebApiAdapter::badStatus()` wird emittiert, falls die API mit einem anderen Status als `200` antwortet. Der zurückgegebene Status wird von dem Signal über `int status_code, QString status` übermittelt. Der Hauptthread kann dann entscheiden, wie der entsprechende Status dem Nutzer darzustellen ist.

Um auf die Signale reagieren zu können, besitzt `MainWindow` diese auffangende Slots, die entsprechend der jeweiligen Signale Veränderungen an der Benutzeroberfläche durchführen. Alle weiteren Aufgaben sind per Konvention in anderen Threads vor dem Senden das Signals durchzuführen, um zu verhindern, dass der GUI-Thread für längere Zeit reaktionsunfähig bleibt. Signale und Slots werden bei der Initialisierung von `MainWindow` verbunden, sodass die Aufgaben den Thread und das Objekt wechseln:

```
connect(webApiAdapter, SIGNAL(badStatus(int, QString)), this,
SLOT(onBadStatus(int QString)));
connect(webApiAdapter, SIGNAL(done(QVector<Book>)), this,
SLOT(onDone(QVector<Book>)));
```

Listing 4.2: Verbinden der Signale und Slots in `MainWindow`

Die Threadstruktur lässt sich am Flussdiagramm in Abbildung 4.2 zusammenfassend darstellen. Die Objekte `ApiQuery` und `QVector<Book>` wechseln, wie deutlich zu erkennen, die Threads und müssen deshalb thread-save sein. Um das sicherzustellen, werden die Ergebnisse nicht als Pointer, sondern ausschließlich als Referenzen übergeben, damit nicht ein anderer Thread die Attribute der Objekte verändern könnte. Außerdem passiert in den Konstruktoren von sowohl `Book` als auch `ApiQuery` keine Instanzierung in besonderem Kontext, die in einem anderen Thread ihre Bedeutung verlieren würde.

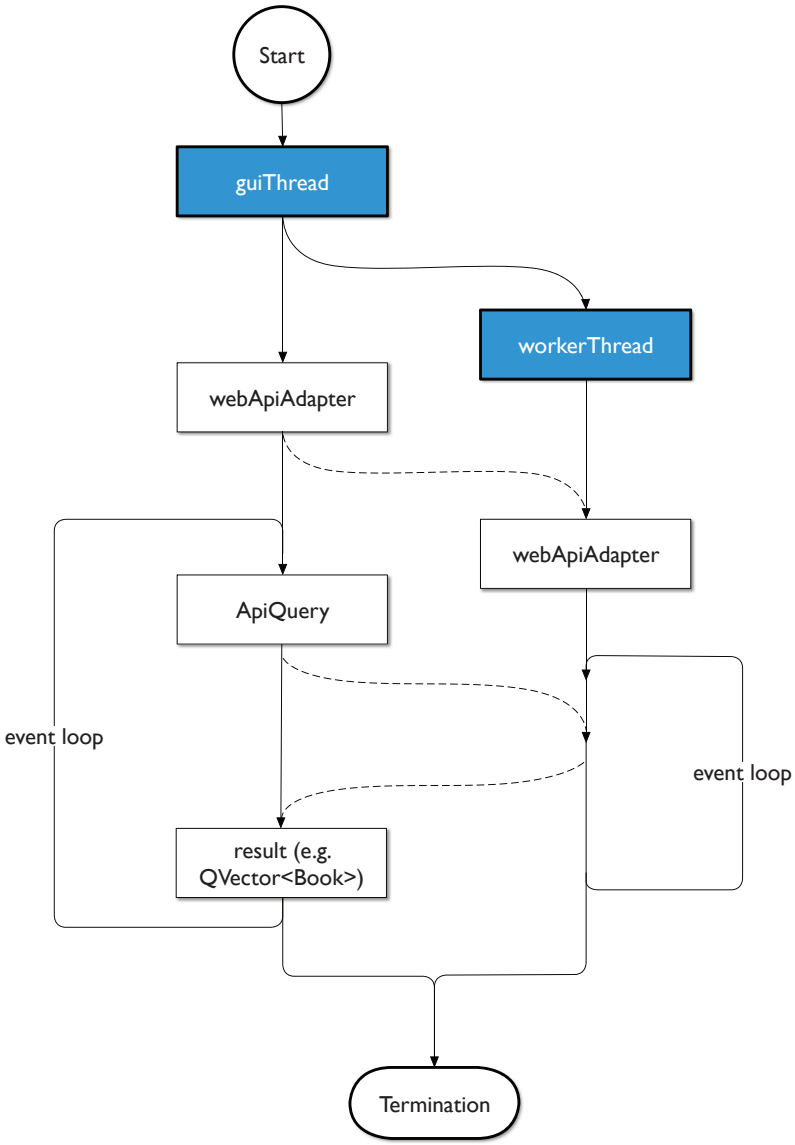

Abbildung 4.2: UML der Threads von bookshelf-qt

4.1.3 Serverkommunikation

Die Kommunikation mit dem Server ist für die Funktionalität des Clients entscheidend, schließlich dient er als Schnittstelle zwischen Server und Nutzer. Diese wichtigen Aufgaben werden von der Kernklasse `WebApiAdapter` übernommen. Dazu verwendet der Adapter einen `QNetworkAccessManager`, welcher HTTP Post Requests versenden kann. Dazu erhält er `QNetworkRequests` und den dazu passenden Header in Form eines `QByteArrays`. Diese werden aus den `WebApiAdapter` übergebenen `ApiQuerys` generiert. `ApiQuery` besitzt dazu eine eigene Methode `ApiQuery::toPostRequest()`. Da `QNetworkAccessManager` mehrere Argumente für die POST Request benötigt, fasst `ApiQuery` diese in einem `struct ApiQuery::PostRequest` zusammen:

```
struct PostRequest {
        QNetworkRequest request;
        QByteArray params;
};
```

Listing 4.3: Deklaration von `ApiQuery::PostRequest`

Um zu erfahren, was nach der Rückgabe zu tun ist, wird der noch nicht gefüllte `QNetworkReply` durch eine Eigenschaft `"action"` ergänzt. Diese lässt sich auch später vom gefüllten `QNetworkReply` zurückholen. Liefert die Anfrage an den Server eine Antwort, so sendet `QNetworkAccessManager` das Signal `QNetworkAccessManager::finished()` aus, welches außerdem den mit der Antwort gefüllten `QNetworkReply` übergibt. Dieses wird mit dem Slot `WebApiAdapter::onFinished()` verbunden, sodass auf die Serverantwort reagiert werden kann:

```
PostRequest postRequest = query->toPostRequest();
QNetworkReply *reply = network_manager->post(
postRequest.request, postRequest.params);
reply->setProperty("action", query->getAction());
connect(network_manager, SIGNAL(finished(QNetworkReply*)),
this, SLOT(onFinished(QNetworkReply*)));
```

Listing 4.4: Senden einer POST-Request

Empfängt der Slot eine Antwort, wird aus dem erhaltenen `QNetworkReply` zunächst der Antwortinhalt ausgelesen und dann mittels `QJsonDocument::fromJson()` in ein `QJsonDocument` umgewandelt, dass es ermöglicht, die in JSON gefasste Serverantwort strukturiert im Programmkontext zu entschlüsseln. Das `QJsonDocument` wird im Weiteren

durch mehrere if-Statements geprüft, sodass sicher ist, dass das für die externe API von bookshelf-server standardisierte Objektmodell vorliegt. Ist dies nicht der Fall, so wird das Signal `WebApiAdapter::error()` abgesendet und mit ihm auch die Fehlermeldung mitgeschickt, um eventuell auf eine Antwort wartende Objekte auf den vorliegenden Fehler hinzuweisen. Zusätzlich zu den Bookshelf spezifischen Fehlern wird, falls nötig, auch der `QJsonParseError` geprüft, falls statt eines Modellfehlers ein Einlesefehler entstanden ist.

Stellt sich das Objektmodell als korrekt heraus, wird dann im zweiten Schritt zunächst der `status_code` überprüft. Stellt sich hier heraus, dass der Server einen Fehler in der Anfrage gefunden hat, oder ein sonstiges, serverseitiges Problem entstanden ist, kann der weitere Vorgang ebenfalls abgebrochen werden. Damit auch hier die wartenden Objekte informiert werden, wird ein `WebApiAdapter::badStatus()` Signal gesendet. Dieses teilt den Empfängern außer dem `status_code` auch die in `status` verfügbaren, menschenlesbare Statusnachricht mit, falls sie diese dem Nutzer ausgeben wollen:

```
if(document.isObject()) {
        QJsonObject wrapperObject = document.object();
        QJsonValue status_code = wrapperObject.value("status_
code");
        if(status_code.isDouble()) {
            if(status_code.toDouble() == WebApiAdapter::OK) {
                QJsonValue result =
wrapperObject.value("result");
                if(!result.isUndefined()) {
                    processResult(reply->property("action").
toString(), result);
                } else {
                    // Error
                }
            } else {
                QJsonValue status = wrapperObject.
value("status");
                if(status.isString()) {
                    emit badStatus(status_code.toDouble(),
status.toString());
                } else {
                    // Error
                }
            }
        } else {
            // Error
        }
```

Listing 4.5: Testsequenz für die Serverantwort

Ist nun `status_code == 200`, kann der Client von einem erfolgreichen Request ausgehen und sich der Umwandlung des im JSON als `"result"` bezeichneten Ergebnisses widmen. Dazu verwendet er seine Methode `WebApiAdapter::processResult()`, welcher neben der `QJsonValue`, die die angefragten Daten enthält, auch die zuvor in dem `QNetworkReply` gespeicherte Information über die angefragte `action` (vgl. Listing 4.5).

Die Methode prüft dann `action` auf die verschiedenen durch die externe API vorgegeben und vom Client unterstützen `actions` und führt die zu den dafür zu erwartenden Daten passende Umwandlung durch. Da der Client in seiner aktuellen Version ausschließlich die `action listBooks` unterstützt, die allerdings sehr mächtig ist, wird der weitere Verlauf der Serverdaten an ihrem Beispiel illustriert:

Damit sicher ist, dass die Daten das erwartete Format besitzen, wird dieses im Weiteren überprüft. So muss im Falle von `listBooks` ein `QJsonArray` vorliegen. Dieser Array wird dann mithilfe einer for-Schleife durchlaufen und die einzelnen Items – Bücher – werden mithilfe von `Book::fromJson()` in den Datentyp `Book` umgewandelt.

Diese statische Methode von `Book` geht durch das `QJsonObject` und wandelt die einzelnen passenden `QJsonValues` in die entsprechenden C++- bzw. Qt-Datentypen um. Diese umgeschriebenen Variablen schreibt die Methode dann in eine `Book` Instanz und gibt diese zurück.

Das so umgewandelte Buch geht dann – in Verbindung mit anderen angefragten Büchern in einem `QVector<Book>` – durch das Signal `WebApiAdapter::done()` an alle wartenden Objekte, sodass die Anfrage nun auch für diese Objekte beantwortet ist.

4.1.4 Benutzeroberfläche

Die Benutzeroberfläche besteht in der aktuellen Version des Clients aus nur zwei Screens:

Die Home-Seite bildet die Startoberfläche für den Nutzer, von welcher aus er sich in seiner Bibliothek orientieren kann. Die Bücher findet er in einer Grid-Anordnung vor, repräsentiert durch ihre Cover, oder – falls kein Cover verfügbar – durch ihren Titel (s. Abbildung 4.3).

Sind noch keine Bücher verfügbar, zeigt die Startseite eine Ladeanimation.

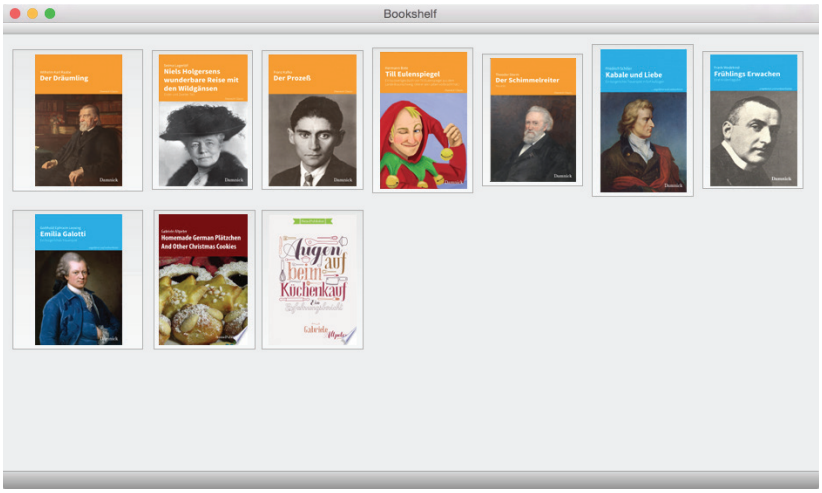

Abbildung 4.3: Grid View im Qt-Client

Klickt der Nutzer auf eines der Cover, so öffnet sich die Detailseite des Buches. Dem Nutzer werden, ähnlich wie im Web Client, einige Informationen zum Buch, wie etwa die Inhaltsbeschreibung oder der Autor geboten. Außerdem hat er die Möglichkeit, durch einen Klick auf den Button „Read now“ auf der unteren rechten Seite, den Browser zu öffnen, um das Buch zum Lesen herunterzuladen. Ein Klick auf den Buttton „Back“ führt ihn zurück zum Home-Seite.

Abbildung 4.4: Detailansicht im Qt-Client

4.1.5 Details zur Implementierung

Das Grunddesign der GUI wurde zum Großteil über die Oberfläche des Qt Creators erstellt. Sie generiert automatisch eine `.ui`-Datei, die vom Compiler zur fertigen Nutzeroberfläche umgesetzt wird.

Um die beiden unterschiedlichen Seiten in einem Fenster darzustellen, bietet es sich an, als zentrales Widget im Fenster ein `QStackedWidget` einzusetzen. Es kann auf mehreren Seiten verschiedene Layouts beherbergen und bietet die Möglichkeit, zwischen diesen umzuschalten. Damit das Design des Widgets nicht mit den beinhaltenden interferiert, besitzt es keinen `margin` und es ist auch kein `spacing` definiert.

Die Home-Seite besteht aus einem, in eine `QScrollArea` eingebundenem `ShelfWidget`, welches für den Bookshelf-Client spezifisch ist (s. Abschnitt ShelfWidget). In dieses `ShelfWidget` werden, sobald aus dem Internet geladen, die Bücher der Bibliothek des Nutzers eingefügt. Dazu werden die Cover als `QIcon` auf `QToolButtons` gesetzt und diese dann zu dem `ShelfWidget` hinzugefügt.

Die Ladeanimation besteht aus einem einfachen `QLabel`, das ausgeblendet wird, sobald der Ladevorgang beendet ist. Im Label wird eine `gif`-Animation eines Ladekreises mithilfe der Klasse `QMovie` angezeigt:

```
QMovie *loadingMovie = new QMovie(":/icons/loading");
ui->loadingSign->setMovie(loadingMovie);
ui->loadingSign->show();
loadingMovie->start();
```

Listing 4.6: Initialisierung der Ladeanimation

Die Detailseite besitzt hingegen keine fensterfüllende `QScrollArea`, sondern hat am unteren Fensterende noch eine Buttonleiste, die durch ein `horizontalLayout` realisiert wird. In der `QScrollArea` befinden sich ein `QLabel`, das das Cover anzeigt, und eines, das weitere Informationen angibt. Um den Text im zweiten `QLabel` zu stylen, bedient sich der Client den umfangreichen Rich Text-Features dieser Klasse und ergänzt die Informationen durch eine HTML-Struktur. Dies geschieht in der Methode `MainWindow::generateHtmlInformation()`.

ShelfWidget

Qt bietet von Haus aus kein Grid Widget an, das je nach verfügbarer Fenstergröße die Zeilen des Grids umbricht. Dieses wird allerdings für die GUI des Bookshelf-Clients benötigt, um die Bücher in einem Grid untereinander anzuordnen und dabei an die Fanstergröße anzupassen. Deshalb gehört zu den Klassen des Clients auch `ShelfWidget`. Diese Klasse erbt von `QWidget` und lässt sich deshalb genauso verwenden. Das Widget enthält ein `QGridLayout`, in das die `QToolButtons` oder auch andere `QWidgets` eingefügt werden können. Dazu wird die Methode `ShelfWidget::addWidget()` verwendet, die die gleichnamige Methode der Elternklasse überschreibt:

```
void ShelfWidget::addWidget(QWidget *widget) {
    if(current_accumulated_row_width > this->size().width())
{
        current_column = 0;
        current_accumulated_row_width = 0;
        layout->addWidget(widget, layout->rowCount(),
current_column, 1, 1);
    } else {
        layout->addWidget(widget, layout->rowCount() - 1,
current_column, 1, 1);

    }
    layout->setAlignment(widget, Qt::AlignVCenter);
    widgets.append(widget);
    current_accumulated_row_width += widget->sizeHint().
width() + margin;
    current_column++;
}
```

Listing 4.7: `ShelfWidget::addWidget()`*-Methode*

Um ein „Überlaufen“ der Zeile zu verhindern, beobachtet die Methode die maximale (`this->size()`) und tatsächliche Länge (`current_accumulated_row_width`) und fügt die Widgets entsprechend in die gleiche oder eine neue Zeile ein. Außerdem werden die Widgets vertikal in der Mitte zentriert, damit die Cover alle untereinander stehen.

5. bookshelf-ios

Mit dem iOS-Client bookshelf-ios stellen wir eine beispielhafte Implementierung eines mobilen Clients für bookshelf-server zur Verfügung. Dabei ist der Client in der Sprache Objective C geschrieben (s. Kapitel 2.1).

5.1 Benutzeroberfläche

Da es sich bei dem iOS-Client momentan lediglich um eine Demoimplementierung handelt, ist auch die Benutzeroberfläche noch sehr einfach aufgebaut. Beim Start bekommt der Nutzer zunächst eine List View mit sämtlichen Bücher in seiner Bibliothek angezeigt (s. Kapitel 5.3 und Abbildung 5.1).
Um eine größere Performance – auch auf schwächeren Internetverbindungen – bieten zu können, werden in dieser Bibliotheksansicht, anders als im Web- und Desktopclient, noch keine Cover angezeigt. Stattdessen werden sämtliche Bücher schlichtweg übersichtlich aufgelistet. In späteren Versionen sollte überlegt werden, die aus dem Webclient bekannten Such- und Filterfunktionen einzubauen (s. dazu auch Kapitel 7.3.1).

Beim Antippen eines Buches wird die Detailansicht (s. Kapitel 5.4 und Abbildung 5.2) angezeigt. Dort bekommt der Nutzer zusätzlich die Beschreibung des Buches angezeigt. In späteren Versionen ist eine noch umfangreichere Darstellung geplant. So müsste beispielsweise auch das Cover (welches mobilverträglich erst bei Bedarf heruntergeladen wird) angezeigt werden.
Außerdem hat der Benutzer die Möglichkeit, das Buch online zu lesen. Dazu wird auf bookshelf-server verlinkt. Hier ist angedacht, das Lesen später direkt auf dem Gerät zu implementieren (s. dazu auch Kapitel 7.3.2).

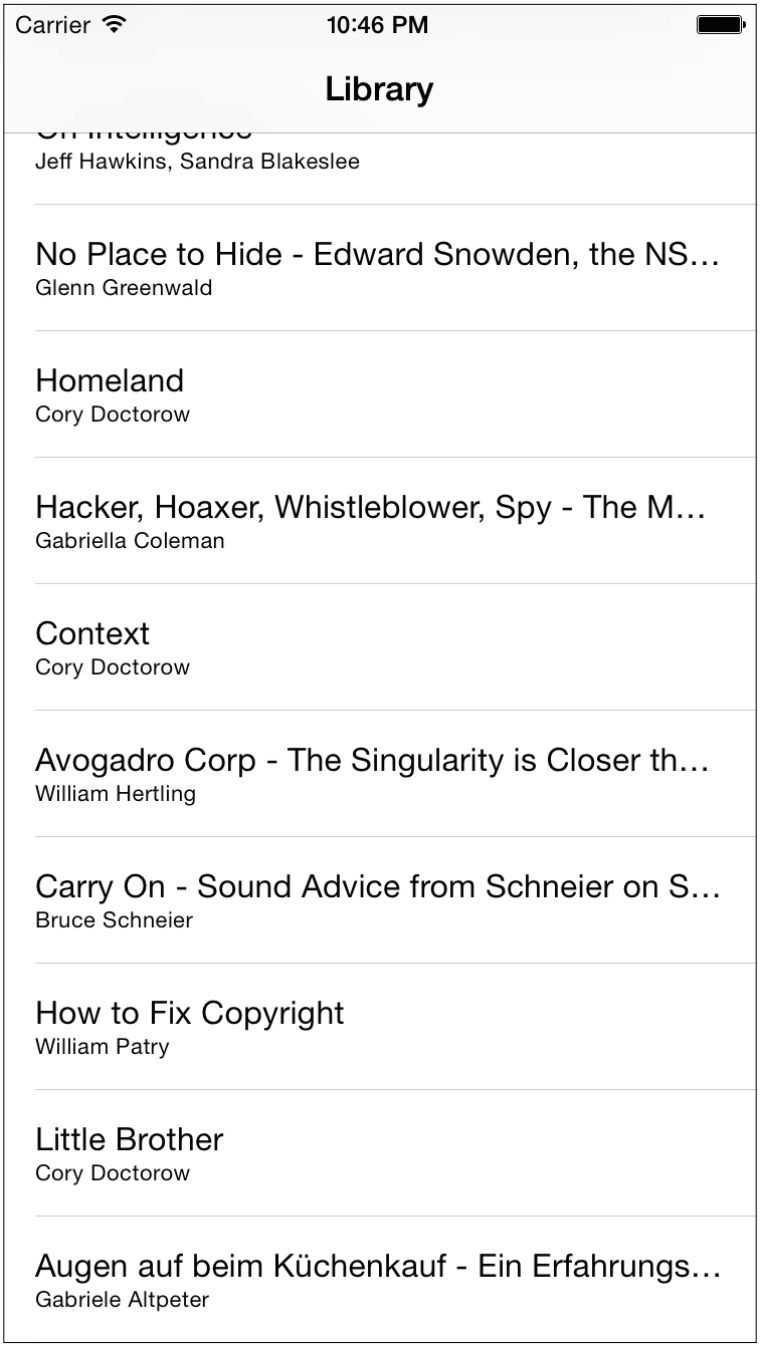

Abbildung 5.1: List View der Bibliothek

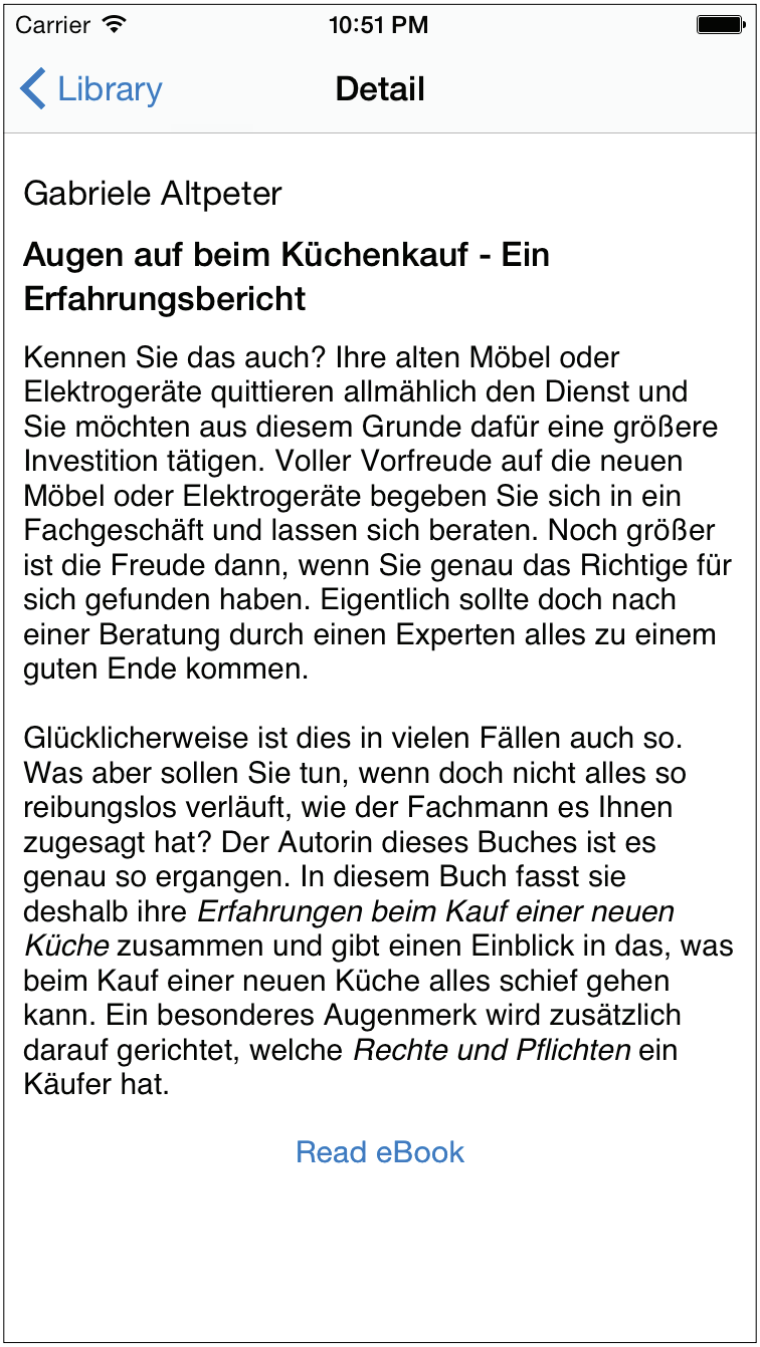

Abbildung 5.2: Detailansicht eines Buches

5.2 Serverkommunikation

Da der iOS-Client bisher wie gesagt nur sehr rudimentär ist und lediglich im Groben zeigt, wie sich ein Client für iOS-Geräte umsetzen lässt, sind auch viele Funktionen der externen API noch nicht implementiert.

So ist momentan lediglich die grundlegendste Funktion, das Ausgeben der Liste aller Bücher in der Bibliothek mitsamt ihrer Metadaten, vorhanden.

Dazu wird eine einfache POST Request an die API von bookshelf-server gestellt (s. Kapitel 2.5). Das Ergebnis im JSON-Format wird anschließend mithilfe von `NSJSONSerialization` in ein `NSArray` serialisiert:

```
- (NSArray *)getBookList {
    NSDictionary *headers = @{ @"accept": @"application/json"
};
    NSDictionary *parameters = @{ @"user": username,
@"password": password, @"action": @"listBooks" };

    UNIHTTPJsonResponse *response = [[UNIRest
post:^(UNISimpleRequest *request) {
        [request setUrl:apiUrl];
        [request setHeaders:headers];
        [request setParameters:parameters];
    }] asJson];

    NSError *error;
    NSArray *result = [[NSJSONSerialization
JSONObjectWithData:response.rawBody options:0 error:&error]
valueForKey:@"result"];
    return result;
}
```

Listing 5.1: Methode zum Abfragen der Bücherliste

5.3 List View

Auch bei der Umsetzung der Benutzeroberfläche lässt sich noch der frühe Status des iOS-Clients erkennen. So wurde hierfür die Xcodevorlage `Master-Detail Application` genutzt. Diese bietet den Vorteil, dass der Client ohne weitere manuelle Anpassung nicht nur auf iPhones und iPods, sondern auch auf iPads eine angemessene Darstellung erfährt und sich bequem nutzen lässt.
Dafür muss man sich jedoch mit Beschränkungen in den Gestaltungsmöglichkeiten abgeben. Angesichts der Tatsache, dass diese Vorlage sehr

gut auf das Konzept von bookshelf-ios passt, ist sie für diese anfängliche Umsetzung ausreichend.

Beim Laden der View (in der `viewDidLoad`-Methode) wird zunächst eine Instanz der Klasse `BookshelfApiRequest` erstellt und über die `getBookList`-Methode (s. Kapitel 5.2) ein Array aller Bücher in der Bibliothek abgefragt:

```
BookshelfApiRequest *bookshelfApiRequest =
[[BookshelfApiRequest alloc] init];
    _books = [bookshelfApiRequest getBookList];
```

Listing 5.2: Abfragen der Bücherliste beim Laden der `MasterView`

Die einzelnen Einträge des Arrays werden dann in die List View hinzugefügt:

```
for(int i = 0; i < [_books count]; i++) {
    [self insertNewObject:nil];
}
```

Listing 5.3: Hinzufügen der Bücher zur List View

Dies geschieht über die `insertNewObject`-Methode:

```
- (void)insertNewObject:(id)sender {
    if(!_books) { _books = [[NSMutableArray alloc] init]; }

    NSIndexPath *indexPath = [NSIndexPath indexPathForRow:0
inSection:0];
    [self.tableView insertRowsAtIndexPaths:@[indexPath]
withRowAnimation:UITableViewRowAnimationAutomatic];
}
```

Listing 5.4: `insertNewObject`*-Methode*

Anschließend müssen nur noch die tatsächlichen Zellen befüllt werden. Da es sich bei der List View im Endeffekt nur um eine spezielle Form der Tabelle handelt, werden die entsprechenden Methoden für Table Views überschrieben.
Dann werden sämtliche HTML-Tags aus den Titel- und Autorangaben entfernt und anschließend der Text für die entsprechenden Labels gesetzt:

```
- (UITableViewCell *)tableView:(UITableView *)tableView
cellForRowAtIndexPath:(NSIndexPath *)indexPath {
    UITableViewCell *cell = [tableView
dequeueReusableCellWithIdentifier:@"Cell"
forIndexPath:indexPath];
```

```
    cell.textLabel.text = [self stringByStrippingHTML:
[[[_books objectAtIndex:[indexPath item]]
valueForKey:@"metadata"] valueForKey:@"title"]];
    cell.detailTextLabel.text = [self stringByStrippingHTML:
[[[_books objectAtIndex:[indexPath item]]
valueForKey:@"metadata"] valueForKey:@"author"]];
    return cell;
}
```

Listing 5.5: Befüllen der Zellen in der List View

5.4 Detailansicht

Die Detailansicht wird über einen `DetailViewController` umgesetzt. Dazu wird die `configureView`-Methode überschrieben.

Damit die `UILabels`, die zur Darstellung der Texte verwendet werden, wird zunächst das `numberOfLines`-Attribut auf `0` gesetzt. Nach dem Setzen des Textes wird schließlich die Methode `sizeToFit` ausgeführt. So wird sichergestellt, dass das Label genau den Platz einnimmt, den es tatsächlich braucht und dass die anderen Labels direkt darauffolgen.

bookshelf-server erlaubt die Verwendung von HTML in der Buchbeschreibung. Dies muss dementsprechend natürlich auch im iOS-Client implementiert werden. Dazu wird ein `NSAttributedString` mit den entsprechenden Daten initialisiert. Da die Standarddarstellung die Beschreibung in Times New Roman stattfände, was zum einen nicht in das Design des Clients passt und zum anderen zu klein ist, muss auch diese Darstellung angepasst werden. Dazu wird einfach vor den HTML-String ein `<style>`-Tag mit entsprechenden CSS-Definitionen gesetzt:

```
- (void)configureView {
    if(_detailItem) {
        _authorLabel.numberOfLines = 0;
        _titleLabel.numberOfLines = 0;
        _descriptionLabel.numberOfLines = 0;

        _authorLabel.text = [[_detailItem
valueForKey:@"metadata"] valueForKey:@"author"];
        _titleLabel.text = [[_detailItem
valueForKey:@"metadata"] valueForKey:@"title"];

        NSString *descriptionHtmlString = [[_detailItem
valueForKey:@"metadata"] valueForKey:@"description"];
        descriptionHtmlString = [NSString
stringWithFormat:@"%@%@", @"<style>body, * { font-family:
```

```
Helvetica, sans-serif; font-size: 15px; }</style>",
descriptionHtmlString];

        NSAttributedString
*descriptionAttributedString = [[NSAttributedString
alloc] initWithData:[descriptionHtmlString
dataUsingEncoding:NSUnicodeStringEncoding] options:@{
NSDocumentTypeDocumentAttribute: NSHTMLTextDocumentType }
documentAttributes:nil error:nil];
        _descriptionLabel.attributedText =
descriptionAttributedString;

        [_authorLabel sizeToFit];
        [_titleLabel sizeToFit];
        [_descriptionLabel sizeToFit];
    }
}
```

Listing 5.6: `configureView`*-Methode*

Bei Antippen des „Read eBook“-Buttons muss lediglich ein Webbrowser auf die entsprechende Webseite geöffnet werden. Dazu wird zunächst die korrekte URL generiert:

```
- (IBAction)readButtonClick:(id)sender {
    NSString *url = [NSString stringWithFormat:@"%@%@",
@"https://bookshelf.my-server.in/bookshelf/read.php?id=",
[_detailItem valueForKey:@"id"]];
    [[UIApplication sharedApplication] openURL:[NSURL
URLWithString:url]];
}
```

Listing 5.7: Aufrufen von bookshelf-server zum Lesen des eBooks

6. bookshelf-android

Auch auf dem größten mobilen Betriebssystem soll Bookshelf natürlich verfügbar sein. Deshalb gibt es auch dort einen nativen Client. Wie auf iOS ist er allerdings im aktuellen Stadium nicht mehr als ein Proof-of-Concept. Als Democlient verfügt er über die Möglichkeit, eine Liste der Bücher anzuzeigen und zu den einzelnen Büchern Detailinformationen auszugeben. Das Design ist dabei allerdings noch nicht optimiert, sondern ist ein schlichtes Standarddesign von Android. Trotzdem ist die App auch für Tablets und weitere Geräte mit großen Bildschirmen optimiert, da diese gerade für eBooknutzer die interessantesten Plattformen sind.

6.1 Benutzeroberfläche

Androidanwendungen setzten sich aus mindestens einer, meist aber mehreren sog. `Activitys` zusammen. Diese verwalten immer den aktuell auf dem Bildschirm zu sehenden, oder gerade erst gestoppten Teil der Applikation. Die Bookshelf-App bedient sich aktuell zweier `Activitys`, von denen eine, `BookListActivity`, immer zum Einsatz kommt, während die andere, `BookDetailActivity`, nur auf Geräten mit kleinem Bildschrim gebraucht wird.

Die `Activitys` initialisieren in ihrem `Context` die zugehörigen `Fragments`, die sich dann selbst mithilfe eines `LayoutInflaters` in die GUI einfügen. Diese wird über über in XML gefasste Layoutdateien gestaltet, die in der `resource`-Umgebung eingefügt werden. Android bietet die Möglichkeit, mithilfe von Präfixen für verschiedene Bildschirmgrößen verschiedene Layouts anzubieten. So lassen sich für die verschiedenen `Fragments`, also funktionale Untereinheiten einer `Activity`, bestimmte Bereiche im Lay-

out definieren: Auf Geräten mit großen Bildschirmen kann das `BookListFragment` links neben dem `BookDetailFragment` auftauchen:

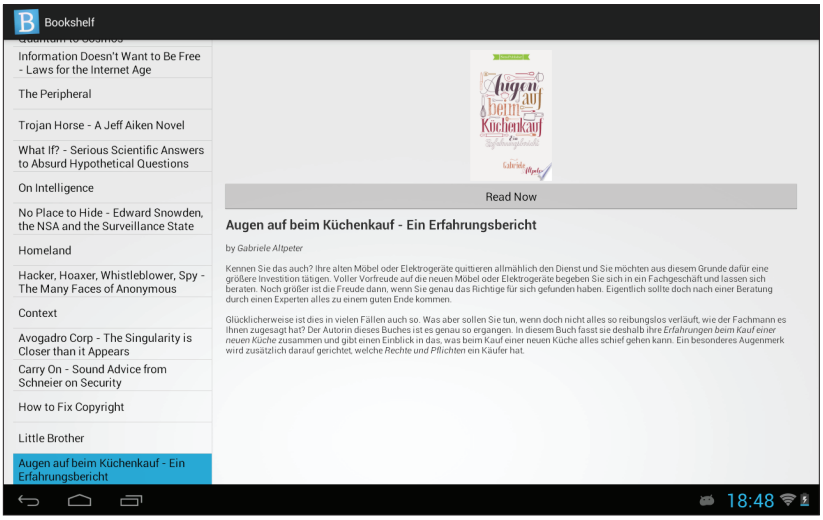

Abbildung 6.1: bookshelf-android auf einem Tablet

Da der Platz auf kleineren Geräten begrenzt ist, kann immer nur ein Fragment gleichzeitig angezeigt werden, sodass hier nun auch zwei `Activitys` verwendet werden müssen. Wird auf ein Item aus der `ListView` des `BookListFragments` geklickt, wird der Nutzer an die zweite Activity weitergeleitet (s. Abbildung 6.2 und Abbildung 6.3).

Um herauszufinden, welche Fragments initialisiert werden müssen, also ob die Applikation auf einem großen oder kleinen Gerät läuft, überprüft die Hauptaktivität `BookListActivity`, ob der Platz für das `BookDetailFragment` verfügbar ist. In diesem Fall hätte der Android-Interpreter das für die zum Gerät passende Layout ausgewählt, in dem im Falle des großen Geräts dieser Platz, `R.id.book_detail_container`, eingefügt ist:

```
if (findViewById(R.id.book_detail_container) != null) {
     mTwoPane = true;
    ((BookListFragment) getSupportFragmentManager()
        .findFragmentById(R.id.book_list))
        .setActivateOnItemClick(true);
}
```

Listing 6.1: Initialisieren des BookDetailFragment auf großen Bildschirmen

Im `BookDetailFragment` funktioniert die GUI dann ähnlich wie im Qt-Client: Es werden Cover und interessante Informationen angezeigt, außerdem ist ein Button verfügbar, über den das Aufrufen des Buches im Browser möglich sein soll. Das Cover wird in einer `ImageView` eingefügt, nachdem das von der externen API gesendete, base64-enkodierte Bild dekodiert und von Android in ein `Bitmap` umgewandelt wurde.

Die Informationen werden wieder – wie beim Qt-Client – durch HTML strukturiert und dann über die HTML-Anzeigemöglichkeit der `TextView` im Standardstil ausgegeben:

```
((TextView) rootView.findViewById(R.id.label_information)).
setText(Html.fromHtml(makeHtmlInformation(book_metadata)));
((ImageView) rootView.findViewById(R.id.image_cover)).
setImageBitmap(decodeBase64(book_metadata.getString("cover_
image")));
```

Listing 6.2: Setzen des Inhalts des `BookDetailFragment`

6.2 Concurrency

Um auch im Android-Client den GUI-Thread nicht zu stark zu belasten, werden die rechenaufwendigen bzw. lange dauernden Operationen in einen zweiten Thread ausgelagert. Dazu bietet Android diverse Möglichkeiten. So ist es z.B. auch möglich, wie beim Qt-Client, von der `Thread` Klasse zu erben und so einen eigenen Thread zu erstellen. Doch für wenig komplexe Aufgaben bietet Android mit `AsyncTask` eine bessere Möglichkeit. Für die eigene, asynchron durchzuführende Aufgabe lässt sich das Interface implementieren. Die Realisierungen von `AsyncTask` können dann eine Aufgabe in einem asynchronen Thread durchführen und lösen sich danach wieder auf.

In der Methode `GetBookList::doInBackground()` der Klasse `GetBookList`, welche `AsyncTask` implementiert, werden dann asynchron die Daten mithilfe von `HttpsURLConnection` von Server in einem `OutputStream` heruntergeladen und in ein `JSONObject` umgewandelt. Das Ergebnis der Aufgabe – eine Liste aller Buchtitel – wird dann an die wieder im Hauptthread laufende Methode `GetBookList::onPostExecute()` weitergegeben. Diese gibt Daten dann über einen `ArrayAdapter` an die `ListView` der GUI weiter. Die gesamten heruntergeladenen Daten verbleiben in einem Attribut des `BookListFragments`.

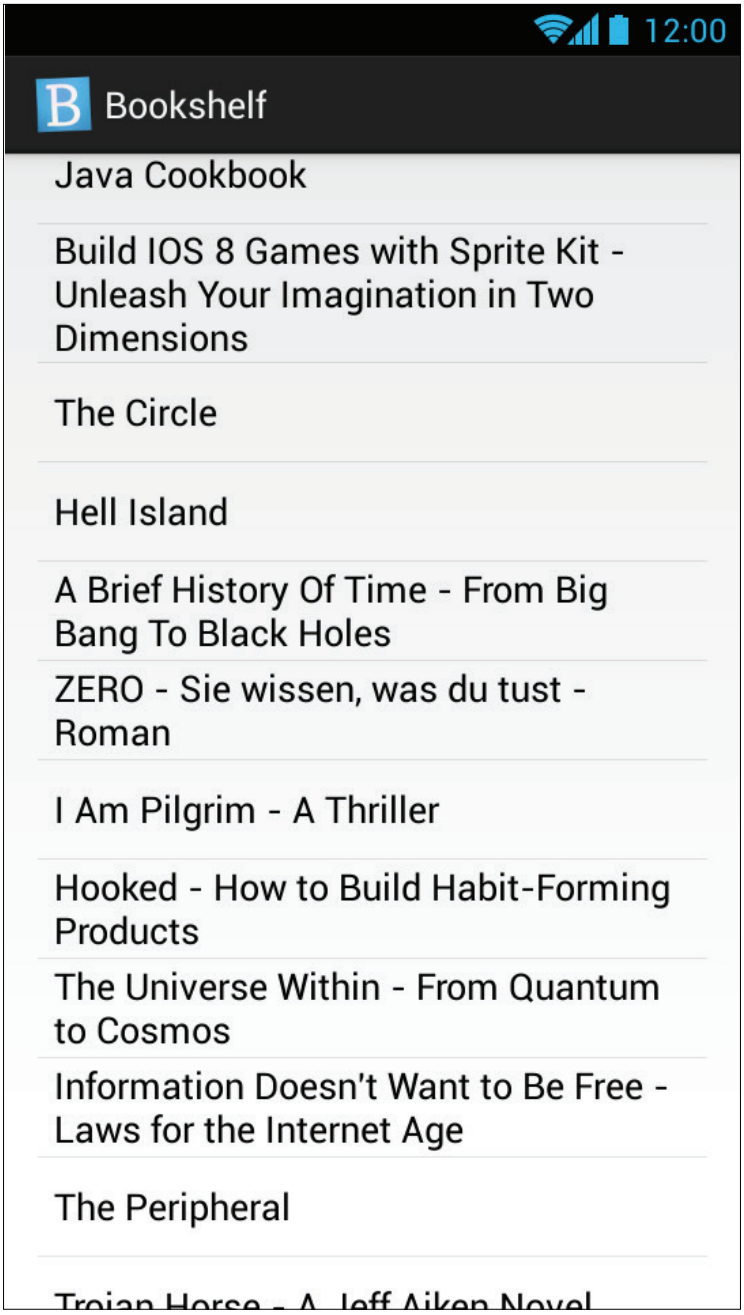

Abbildung 6.2: Listenansicht auf kleineren Geräten

Abbildung 6.3: Detailansicht auf kleineren Geräten

Wird auf ein Listenelement gedrückt, werden mithilfe der vom `ArrayAdapter` erhaltenen `id` die passenden Daten als `JSONObject` aus dem Attribut geholt und dann, zur einfachen Versendung, in einen String umgewandelt. Dann wird dieser String an die übergeordnete `BookListActivity` weitergeleitet und diese somit auch informiert:

```
public void onListItemClick(ListView listView, View view, int
position, long id) {
    super.onListItemClick(listView, view, position, id);

    try {
        mCallbacks.onItemSelected(book_array.
getJSONObject(position).toString());
    } catch (JSONException e) {
        e.printStackTrace();
    }
}
```

Listing 6.3: Senden der Daten des gewählten Buches an die `BookListActivity`

Die `BookListActivity` entscheidet dann, je nachdem ob beide `Fragments` zu sehen sind, wie weiter verfahren werden soll:

```
public void onItemSelected(String json) {
    if (mTwoPane) {
        Bundle arguments = new Bundle();
        arguments.putString(BookDetailFragment.ARG_JSON,
json);
        BookDetailFragment fragment = new
BookDetailFragment();
        fragment.setArguments(arguments);
        getSupportFragmentManager().beginTransaction()
                .replace(R.id.book_detail_container,
fragment)
                .commit();

    } else {
        Intent detailIntent = new Intent(this,
BookDetailActivity.class);
        detailIntent.putExtra(BookDetailFragment.ARG_JSON,
json);
        startActivity(detailIntent);
    }
}
```

Listing 6.4: `onClick`*-Event von* `BookListActivity`*, geworfen durch das* `BookListFragment`

Ist nur ein Fragment zu sehen, muss eine neue `BookDetailActivity` geöffnet und die aktuelle gestoppt werden. Dazu erstellt die `BookListActi-`

vity ein in Android Intent genanntes Signal für die neue BookDetailActivity. An diesen können dann noch weitere Informationen, wie etwa die Buchdetaildaten, gehängt werden. Die neue BookDetailActivity dient dann nur als Container für ein BookDetailFragment, an welches die BookDetailActivity die erhaltenen Informationen einfach weiterleitet.

Sind beide Fragments auf dem Bildschirm zu sehen, sendet die BookListActivity die Daten direkt an das bekannte BookDetailFragment weiter, das damit wie in Abschnitt 6.1 beschrieben verfährt.

7. Perspektiven/Weiterentwicklung

In diesem Kapitel soll nun auf Perspektiven und mögliche Weiterentwicklungen von bookshelf-server, aber auch bookshelf als Gesamtprojekt eingegangen werden. Dazu werden die Vorschläge in verschiedene Kategorien unterteilt.

7.1 Backend

7.1.1 Optimierung des Error Handlings

Aktuell ist nicht eindeutig definiert, was im Falle eines Fehlers zu geschehen hat. Wir haben zwar einen `ErrorHandler` implementiert, dieser entspricht jedoch leider nicht vollständig unseren Anforderungen.
So bietet dieser uns zwar die Möglichkeit, jederzeit unabhängig von der aktuellen Methode einen Fehler zu throwen und separat darauf zu reagieren, allerdings bietet diese Herangehensweise keine Möglichkeit, direkt in der Methode auf den Fehler zu reagieren, sodass diese zumeist entweder trotz des Fehlers weiterläuft oder manuell ein vorzeitiges Abbrechen einprogrammiert werden muss. Doch auch in diesem Fall ist nicht klar, welcher Wert zurückgegeben werden sollte, so dass jene Methoden, welche die Methode, in der der Fehler auftritt, aufrufen, klar wissen, wie mit diesem Fehler umzugehen ist.

Es kam deshalb die Idee auf, statt des `ErrorHandlers` ein `ErrorObject` zu implementieren, welches im Falle eines Fehlers zurückgegeben wird und die aktuelle Methode somit direkt beendet. Anhand der im `ErrorObject`

gespeicherten Werte wäre es dann auch anderen Methoden möglich, spezifisch auf den Fehler zu reagieren.[13]

7.2 Web App

7.2.1 Mehr Metadatenquellen

Zum Abrufen von Metadaten greift bookshelf-server in der aktuellen Version lediglich auf die Google Books API[14] zu. Diese bietet zwar eine große Menge an Daten, die für die aktuelle frühe Entwicklungsversion auch völlig ausreichend sind, doch um dem Nutzer in späteren Versionen mehr Komfort zu bieten, empfiehlt es sich dennoch, auch weitere Quellen anzubieten.

Da bookshelf-server bereits eine `ExternalApiRequest`-Klasse bietet, müssen für andere APIs lediglich die spezifischen Funktionen integriert werden, was ohne Probleme möglich ist.

Eine Liste möglicher Kandidaten für eine Integrierung in bookshelf-server findet sich in Issue #1 (https://git.my-server.in/bookshelf/bookshelf-server/issues/1).

7.2.2 Optimierung der Coverbilder

Zur Darstellung (und Weitergabe über die externe API) von Coverbilder setzt bookshelf-server momentan auf als base64-enkodierte data URIs. Diese haben den Vorteil einer einfachen und portablen Nutzung, was auch der Grund ist, weshalb wir uns anfangs für diesen Weg entschieden haben.
Es muss jedoch beachtet werden, dass diese Methode auch gewisse Nachteile mit sich zieht. So ist zu allem das große Datenvolumen zu erwähnen, dass bei jedem Abruf der Metadaten verbraucht wird.

Aus diesem Grund haben wir uns entschieden, die Coverbilder zu optimieren. Es ist angedacht, dass Coverbilder in Zukunft einfach über eine

13 s. hierzu auch Diskussion unter https://git.my-server.in/bookshelf/bookshelf-server/issues/25

14 Google Inc.: *Google Books API Family*. 2012. URL: https://developers.google.com/books/ (Abruf am 14. Juni 2015).

URL, die einen entsprechenden ID-Parameter enthält (ähnlich wie beim Download von eBooks) weitergegeben werden.

Darüber hinaus bietet es sich an, bei diesem Weg direkt die Möglichkeit anzubieten, dass Coverbild zu skalieren und so verschiedene Thumbnailgrößen zur Verfügung zu stellen, um das verbrauchte Datenvolumen noch einmal zu begrenzen.

Als mögliches Beispiel für so eine Umsetzung sei hier die Cover-API des VLB (Verzeichnis Lieferbarer Bücher) genannt. Diese strukturiert Anfragen wie folgt:

```
https://vlb.de/GetBlob.aspx?strIsbn=978-3-945748-00-8&size=L
```

Es können also ISBN (in userem Fall wäre dies die ID) und eine Größe spezifiziert werden.

7.2.3 Application Installer

Um dem Nutzer eine einfache Einrichtung von bookshelf-server zu ermöglichen, wollen wir einen Installer entwickeln, der alle wichtigen Schritte übernimmt. So soll dieser zum Einen überprüfen, ob alle notwendigen Bedingungen erfüllt sind, ob also z.B. alle notwendigen Abhängigkeiten vorhanden sind und eine TLS-verschlüsselte Verbindung vorliegt. Weiterhin soll er den Benutzer nach den gewünschten Einstellungen fragen und diese entsprechend setzen (wofür eine Erweiterung der Klasse `\Bookshelf\Core\Configuration` nötig, sodass diese auch Konfigurationswerte schreiben kann). Anhand dieser Werte soll der Installer dann die Anwendung komplett für den Benutzer einrichten, also die Datenbanken erstellen, Dateiberechtigungen entsprechend setzen etc.
Sollten während der Installation Fehler auftreten, wird er natürlich entsprechende Fehlermeldungen ausgeben und, wenn nötig, Hilfe bieten, diese Probleme zu beheben.

Im Rahmen dessen bietet es sich auch an, eine Updatefunktionalität zu implementieren, die in regelmäßigen Abständen eine Paketquelle überprüft und bei Bedarf Updates herunterlädt und installiert.

7.3 iOS-Client

Ein Großteil der hier erwähnten Perspektiven trifft auch auch den Androidclient zu.

7.3.1 Filter- und Suchfunktion

Momentan bietet der iOS-Client nur die Möglichkeit, sich eine vollständige Liste aller Bücher in der Bibliothek anzeigen zu lassen. Es ist jedoch natürlich zu erwarten, dass sich der Nutzer weitere Möglichkeiten wünscht, bestimmte Bücher zu finden. Hier seien die Filter- und Suchfunktionalitäten des Webclients erwähnt (s. Kapitel 3.3.1 und Kapitel 3.3.4). Der Nutzer hat dort die Möglichkeit, entweder ein konkretes Buch zu suchen oder die Bibliothek nach bestimmten Kriterien wie Autor oder Sprache sortieren zu lassen. Eine solche Funktionalität wäre auch in bookshelf-ios sinnvoll.

Dafür muss jedoch zunächst geklärt werden, wo das Filtern bzw. Suchen erfolgen soll. Dies könnte schließlich entweder über eine spezielle Anfrage an bookshelf-server, das dann eine Liste der entsprechenden Bücher zurückgeben würde, oder aber lokal auf dem Endgerät des Benutzers, geschehen.
Hier müssten zunächst die Vor- und Nachteile beider Vorgehensweisen abgewogen werden.

7.3.2 Anzeige der eBooks auf dem Gerät

Zum aktuellen Zeitpunkt lassen sich eBooks aus dem iOS-Client lediglich über den in bookshelf-server integrierten Web Reader (s. Kapitel 3.3.3) lesen. Dies zieht allerdings eine Reihe von Nachteilen mit sich. So ist es zunächst einmal natürlich unpraktisch, eBooks in einem Webbrowser lesen zu müssen. Wir bieten diese Möglichkeit zwar an, es sollte sich damit jedoch nicht um den grundsätzlichen Weg handeln. Darüber hinaus unterstützt der Web Reader aktuell lediglich eBooks im EPUB- und PDF-Format. Damit ist zwar ein großer Teil aller eBooks abgedeckt, es ist aber dennoch nicht unwahrscheinlich, dass der Nutzer einmal ein Buch in einem anderen Format lesen möchte. Dies sollte ihm durchaus möglich sein.

Leider bietet Apple unter iOS nur sehr beschränkte Möglichkeiten zum Download von Dateien. Selbst wenn uns ein Herunterladen der eBookdateien gelingen würde, wird es sehr schwer bis unmöglich, diese in andern eBook-Readeranwendungen zu öffnen.[15] Es wird daher nötig sein, einen eigenen eBook-Reader unter iOS zu implementieren und die Bücher direkt darin anzuzeigen. Dies ist auch der Ansatz, den viele andere Entwickler von eBookanwendungen unter iOS wählen.

7.3.3 Ansprechendere Gestaltung

Der iOS-Client basiert aktuell zu großen Teilen auf der `Master-Detail Application`-Vorlage. Dadurch wird zwar ein Design gewährleistet, dass im Einklang mit üblichen iOS-Richtlinen steht, wir sind layouttechnisch jedoch auch ziemlich stark eingeschränkt.

In späteren Versionen ist daher angedacht, dieses Layout zu erweitern und an unseren speziellen Nutzungszweck anzupassen.

Unter diesen Aspekt fällt allerdings auch die Einbindung weiterer Metadaten in den Client. So werden auch in der Detailansicht momentan lediglich Autor, Titel und Beschreibung angezeigt. Auch dies wird sich in späteren Versionen natürlich ändern.
Sowohl hinsichtlich des Designs als auch der angezeigten Informationen planen wir, uns an bookshelf-server zu orientieren, um dem Nutzer eine einheitliche Erfahrung zu bieten.

7.4 Sonstiges

7.4.1 Entwicklung weiterer Clients

Mit bookshelf-server, bookshelf-qt, bookshelf-android und bookshelf-ios bieten wir zwar bereits Clients für den Browser, sowie sämtliche gängigen Desktop- und Mobilumgebungen, insbesondere im Desktopbereich handelt es sich dabei aktuell aber nur um einen allgemeinen Client für alle Plattformen. Hier wäre es wünschenswert, gewisse Plattformen in Zukunft nativ mit einem speziellen Client zu targetten, da alle System gewisse eigene Bedienkonzepte und -strukturen mit sich bringen und es für den Nutzer angenehmer ist, wenn diese auch von der Software entsprechend umgesetzt werden.

15 vgl. hierzu auch https://apple.stackexchange.com/a/25690

Als ein Beispiel sei an dieser Stelle die Entwicklung eines nativen Clients für Mac OS X in Cocoa erwähnt.

Und schließlich besteht auch die Überlegung, direkt für einzelne eBookreader zu entwickeln. Hier stellen sich natürlich deutlich mehr Schwierigkeiten in den Weg. So laufen diese Geräte auf den unterschiedlichsten Architekturen und Systemen, teils ist es überhaupt nicht möglich, eigene Programme darauf zu starten, sodass die Entwicklung einer eigenen Firmware vonnöten wäre. Als positives Beispiel wären hier zwar Geräte wie der Pocketbook 622 Touch[16] zu nennen, der sogar eine Qt Runtime mitbringt. Dennoch muss gesagt werden, dass diese Idee natürlich nur eine recht geringe Priorität hat.

16 PocketBook International SA: *PocketBook Touch*. 2010. URL: http://www.pocketbookint.com/us/products/pocketbook-touch (Abruf am 14. Juni 2015).

Appendix

8. Bookshelf PHP-Codingrichtlinien

Wir beabsichtigen, eindeutige und allgemeine Codingstandards festzulegen, um Konsistenz sicherzustellen.
Dieses Dokument legt die Codingrichtlinien für PHP fest.[17]

8.1 Allgemeine Anmerkungen

Die Intention dieses Dokuments ist es, dem Entwickler Richtlinien zu geben. In manchen Fällen ist es jedoch angemessen, von diesen abzuweichen. Diese Entscheidung bleibt dem Entwickler überlassen.

Die Schlüsselworte MUSS, DARF NICHT, ERFORDERLICH, SOLLTE, SOLLTE NICHT, EMPFOHLEN, KANN und OPTIONAL in dieser Spezifikation sind zu interpretieren wie in RFC 2119 beschrieben

Bookshelf beabsichtigt nicht nur, eigene Richtlinien festzulegen, sondern strebt an, mit offiziellen Standards im Einklang zu sein. Dementsprechend basiert diese Richtlinie auf den PHP FIG-Standards. Es ist jedoch zu beachten, dass es einige Unterschiede gibt.

Als Code formatierte Fragmente bieten Beispiele:

```
Beispiel eines als Code formatierten Fragments, das ein
Beispiel bietet.
```

17 übersetzt nach https://git.my-server.in/bookshelf/bookshelf-meta/blob/master/bookshelf_php_coding_standards.md

8.2 Richtlinien zur objektorientierten Programmierung

8.2.1 Namespaces

Namespace und Klassenname MÜSSEN die folgende Struktur haben:

```
\Bookshelf\[Namespacename]\[Klassenname]
```

Namespace- und Klassenname DÜRFEN NUR Zeichen aus dem ASCII-Zeichensatz verwenden und müssen in „upper camel case“ („PascalCase“) benannt werden.

```
\Bookshelf\ExternalApi\GoogleBooksApiRequest
```

Auf dem Dateisystem muss jedes Fragment in ein separates Verzeichnis übertragen werden und die Dateiendung `.php` muss an den Klassennamen angehängt werden.

```
/Bookshelf/ExternalApi/GoogleBooksApiRequest.php
```

8.2.2 Autoloading

Bookshelf bringt eine `autoload.php`-Datei mit, die in jeder Datei genutzt werden SOLLTE, um die notwendigen Klassen automatisch zu laden.

```
require __DIR__ . '/lib/vendor/autoload.php';
```

8.3 Codingstilrichtlinien

Das folgende Beispiel umfasst die wichtigsten Regeln als schnelle Übersicht:

```
<?php

namespace \Bookshelf\ExternalApi;

use \Bookshelf\DataType;
use \Bookshelf\DataIo\DatabaseConnection;

class GoogleBooksApiRequest extends ExternalApiRequest {
    public function sampleRequest($some_variable,
$other_variable = 'default value') {
        if($some_variable == $other_variable) {
            $this->someMethod();
        }
```

```
        elseif(strtolower($some_variable) == $other_variable)
{
            // TODO: Show warning to the user
        }
        else {
            echo 'Some message to the user.';
        }
    }
}
```

8.3.1 Dateien

Dateien DÜRFEN NUR `<?php` (bzw. `?>`) nutzen.

Dateien MÜSSEN folgendes Encoding verwenden: UTF-8 ohne BOM.

Alle Dateien MÜSSEN Unix Line Feeds verwenden.

Alle Dateien MÜSSEN mit einer einzelnen, leeren Zeile enden.

Der schließende `?>`-Tag MUSS in Dateien, die nur PHP enthalten, ausgelassen werden.

8.3.2 Konstanten

Konstanten MÜSSEN vollständig in Großbuchstaben und mit Unterstrichen als Leerzeichen deklariert werden.

```
const VERSION_CODE = 1;
```

Wenn eine Klassenkonstante auf eine Expression gesetzt werden soll, kann `define()` verwendet werden.
Der für `define()` verwendete Name SOLLTE dem Konstantennamen in Kleinbuchstaben und ohne Separatoren entsprechen.

```
define('rootdir'), __DIR__ . '/../../../../');

class Application {
    const ROOT_DIR = rootdir;
}
```

8.3.3 Einrücken

Code SOLLTE eingerückt werden, um die Lesbarkeit zu erhöhen.

Code DARF NUR mit vier Leerzeichen eingerückt werden und DARF NICHT mit Tabs eingerückt werden.

8.3.4 PHP-Keywords und -Konstanten

PHP-Keywords und -Konstanten MÜSSEN kleingeschrieben werden.

```
$some_bool = false;
$my_var = null;

exit();
```

8.3.5 `namespace`- und `use`-Deklarationen

Vor und nach der `namespace`-Deklaration MUSS sich eine leere Zeile befinden.

Die `use`-Deklaration MUSS hinter die `namespace`-Deklaration geschrieben werden.

Hinter der `use`-Deklaration MUSS sich eine Leerzeile befinden.

8.3.6 Klassen

Die `extends`- und `implements`-Keywords MÜSSEN in der gleichen Zeile deklariert werden wie der Klassenname.

Die öffnende geschweifte Klammer für die Klasse MUSS in der gleichen Zeile stehen wie der Klassenname. Die schließende geschweifte Klammer für die Klasse MUSS in einer neuen Zeile nach dem Klasseninhalt stehen.

```
class SomeClass extends ParentClass implements \Countable {
    // class definition
}
```

8.3.7 Properties

Alle Properties MÜSSEN ihre Sichtbarkeit deklarieren.

In einem Statement DARF NICHT mehr als ein Property deklariert werden.

Propertynamen MÜSSEN in Snake Case deklariert werden.

Propertynamen SOLLTEN KEIN Präfix haben, das Variablentyp oder Sichtbarkeit anzeigt.

```
public $cover_image;
private $user;
```

8.3.8 Methoden

Alle Methoden MÜSSEN ihre Sichtbarkeit deklarieren.

Methodennamen SOLLTEN KEIN Präfix haben, das Rückgabetyp oder Sichtbarkeit anzeigt.

Methodennamen MÜSSEN in Camel Case deklariert werden.

Methodennamen MÜSSEN mit einem Leerzeichen nach dem Methodennamen deklariert werden. Die öffnende geschweifte Klammer DARF NICHT in eine neue Zeile geschrieben werden und die schließende geschweifte Klammer MUSS in einer neuen Zeile nach dem Methodeninhalt stehen. Es DARF KEIN Leerzeichen nach der öffnenden vor der schließenden Klammer der Argumente stehen.

In der Liste der Argumente DARF KEIN Leerzeichen vor den Kommas stehen und es MUSS ein Leerzeichen nach den Kommas folgen.

Methodenargumente mit Standardwerten MÜSSEN am Ende der Argumenteliste stehen.

```
public function volumeSearch($q, $limit = 0) {
    // function definition
}
```

8.3.9 Methodenaufrufe

Bei Methoden- und Funktionsaufrufen DARF KEIN Leerzeichen zwischen dem Methoden- oder Funktionsnamen und der öffnenden Klammer stehen, es DARF KEIN Leerzeichen nach der öffnenden und vor der schließenden Klammer stehen. In der Liste der Argumente DARF KEIN Leerzeichen vor den Kommas stehen und es MUSS ein Leerzeichen nach den Kommas folgen.

```
$database_connection->executeQuery($query);
volumeSearch($q, 3);
```

8.3.10 Strings

Strings DÜRFEN NUR mit doppelten Anführungszeichen " deklariert werden, wenn sie Expressions enthalten. Andernfalls MÜSSEN einfache Anführungszeichen ' verwendet werden.

Variablen in Strings mit doppelten Anführungszeichen SOLLTEN in geschweifte Klammern eingeschlossen sein, um Fehler beim Parsing zu vermeiden.

```
$my_string = 'Hello, World.';
$personal_string = "Hello, {$user}.";
```

8.3.11 Ternäre Operatoren

Um den Code effizienter zu machen, SOLLTE der ternäre Operator verwendet werden, wenn eine direkte bedingte Rückgabe nötig ist.

Der ternäre Operator KANN von Klammern umschlossen werden, um die Lesbarkeit sicherzustellen und das Fehlerrisiko zu verringern. Vor unter hinter dem ? und : MUSS ein Leerzeichen stehen.

Hinter der öffnenden und vor der schließenden Klammer DARF KEIN Leerzeichen stehen.

```
echo 'Some variable: ' . (empty($some_variable) ? 'Nothing' :
$some_variable);
```

8.3.12 `if`-Statements

Ein `if`-Statement MUSS wie folgt aussehen:

```
if($some_variable == $another_variable) {
    $this->someMethod();
}
elseif(strtolower($some_variable) == $another_variable) {
    // TODO: Show warning to the user
}
else {
    echo 'Some message to the user.';
}
```

Falls ein kurzes `if`-Statement nur eine einzelne Zeile umschließt und ohne `else` und `elseif` auskommt, SOLLTE es nur eine einzelne Zeile nutzen.

Solche einzeiligen Statements MÜSSEN die geschweiften Klammern weglassen. Hinter der schließenden Klammer des einzeiligen `if`-Statements MUSS ein Leerzeichen folgen.

```
if($some_variable == $another_variable) echo 'They are
equal';
```

8.3.13 `switch case`-Statements

Ein `switch case`-Statement MUSS wie folgt aussehen. Es MUSS ein Kommentar (wie `// no break`) gesetzt werden, wenn das Durchrutschen ohne `break;` beabsichtigt ist.

```
switch($expression) {
    case 0:
        echo 'That was a zero.';
        break;
    case 1:
        // no break
    default:
        echo 'Default case';
}
```

8.3.14 Schleifen

`for`, `foreach` und `while`-Schleifen MÜSSEN wie folgt aussehen:

```
for($i = 0; $i < 42; $i++) {
    // body
}

foreach($iterable as $key => $value) {
    // body
}

while($expression) {
    // body
}
```

8.3.15 `require`, `include`

`require_once` und `include_once` SOLLTEN genutzt werden, wenn eine Datei nur ein einziges Mal eingebunden werden soll.

`require`, `include`, `require_once` und `include_once` DÜRFEN NUR wie im folgenden Beispiel genutzt werden. Die Klammern werden weggelassen.

```
require Application::ROOT_DIR . 'config.php';
```

8.3.16 Kommentare

Kommentare SOLLTEN nur verwendet werden, um unerwartetes Programmverhalten zu erklären. Code, der einfach zu verstehen ist, SOLLTE NICHT kommentiert werden.

Mit `//` deklarierte Kommentare MÜSSEN um ein Leerzeichen eingerückt werden und DÜRFEN NICHT direkt auf den letzten Slash folgen.

Das `TODO`-Keyword KANN verwendet werden.

```
// implemented according to https://developers.google.com/
books/docs/v1/using#PerformingSearch

// TODO: Change according to coding standards
```

8.3.17 Abkürzungen

Abkürzungen SOLLTEN NICHT in Klassen-, Property- und Methodennamen verwendet werden, um die Lesbarkeit zu erhöhen.

Abkürzungen MÜSSEN für die Benennungskonventionen wie Wörter behandelt werden.

```
class GoogleBooksApiRequest { // Api is not spelled API
    // class body
}
```

9. Quellenverzeichnis

Amazon.com, Inc.: *Receiving Your Kindle Content via Whispernet*. 2009. URL: http://www.amazon.com/gp/help/customer/display.html?nodeId=200375890#whisptrans (Abruf am 14. Juni 2015).

Kovid Goyal: *calibre - About*. 2010. URL: http://calibre-ebook.com/about (Abruf am 14. Juni 2015).

PHP Framework Interoperability Group: *PSR-4: Autoloader*. 2013. URL: https://github.com/php-fig/fig-standards/blob/master/accepted/PSR-4-autoloader.md (Abruf am 14. Juni 2015).

Roy Thomas Fielding et al.: *Hypertext Transfer Protocol – HTTP/1.1*. RFC 2616. Juni 1999, S. 57–71. URL: https://www.ietf.org/rfc/rfc2616.txt.

Paul J. Leach et al.: *A Universally Unique IDentifier (UUID) URN Namespace*. RFC 4122. Juli 2005, S. 5–15. URL: http://www.ietf.org/rfc/rfc4122.txt.

Roy Thomas Fielding: *Architectural Styles and the Design of Network-based Software Architectures*. Diss. University of California, Irvine, 2000.

PocketBook International SA: *PocketBook Touch*. 2010. URL: http://www.pocketbook-int.com/us/products/pocketbook-touch (Abruf am 14. Juni 2015).

Google Inc.: *Google Books API Family*. 2012. URL: https://developers.google.com/books/ (Abruf am 14. Juni 2015).

10. Ressourcen

Wie bereits erwähnt, sind sämtliche Ressourcen im Bezug auf das Bookshelf-Projekt unter freien Lizenzen verfügbar und für jeden zugänglich. Die Projektwebseite getbookshelf.org bietet einen Überblick über sämtliche Inhalte. Dennoch sollen auch an dieser Stelle die URLs der wichtigsten Projektteile erwähnt werden.

Es ist möglich, dass sich die folgenden URLs in der Zukunft ändern. In diesem Fall findet sich unter getbookshelf.org jederzeit der aktuelle Verweis.

10.1 Git Repositories

Sämtlicher Code des Bookshelf-Projekts wird über die Versionsverwaltung git verwaltet. Alle git repositories sind öffentlich einsehbar.

https://git.my-server.in/bookshelf/bookshelf-server
https://git.my-server.in/bookshelf/bookshelf-qt
https://git.my-server.in/bookshelf/bookshelf-android
https://git.my-server.in/bookshelf/bookshelf-ios

10.2 Issue Tracker

Um einen Überblick über zu erledigende Aufgaben und Bugs zu behalten, verwendet Bookshelf Issue Tracker. Diese sind spezifisch für jedes Repository.

https://git.my-server.in/bookshelf/bookshelf-server/issues
https://git.my-server.in/bookshelf/bookshelf-qt/issues

https://git.my-server.in/bookshelf/bookshelf-android/issues
https://git.my-server.in/bookshelf/bookshelf-ios/issues

10.3 Sonstiges

Auch sämtliche Dokumentationen und Standards (darunter auch Kapitel 8) von Bookshelf werden über git verwaltet. Hierfür dient das bookshelf-meta Repository.

https://git.my-server.in/bookshelf/bookshelf-meta

11. Glossar

API

Application Programming Interface (deutsch Programmierschnittstelle), Schnittstelle, über welche externe Software interne Infrastruktur nutzen kann.

Client

Software, die auf einen Dienst zugreift, der von einem Server angeboten wird.

EPUB

Electronic PUBlication, offener Standard für eBooks

GUI

Graphical User Interface (deutsch grafische Benutzeroberfläche), grafische Bedienschnittstelle, über welche der Benutzer mit einem Programm interagieren kann.

JSON

JavaScript Object Notation, ursprünglich von JavaScript stammendes Format zur Darstellung von Objekten. Unterstützt sechs grundlegende Datentypen.

Metadaten

„Daten über Daten“

MySQL

My Structure Query Language, relationales Datenbankamangementsystem unter der GNU General Public-Lizenz.

PDF

Portable Document Format, plattformunabhängiges Dateiformat für Dokumente, u.a. eBooks

PHP

PHP: Hypertext Preprocessor (ursprünglich Personal Home Page), serverseitige Skriptsprache für Webentwicklung.

REST

REpresentational State Transfer, Programmierparadigma für Anwendungen, die über das HTTP-Protokoll implementiert sind und gängigen Webstandards folgen.

Server

Software, die Anfragen von Clients entgegennimmt und entsprechend beantwortet.

SQL Injection

Ausnutzen deiner Sicherheitslücke in Programmen, welche SQL-Datenbanken verwenden. Verursacht durch die mangelnde Maskierung oder Überprüfung von Sonderzeichen in Benutzereingaben.

Threading

Aufteilung eines Programms in mehrere Subprozesse, die nebeneinander ausgeführt werden können. Häufig verwendet, damit der Nutzer die grafische Benutzeroberfläche trotz rechenintensiver Aufgaben weiterhin verwenden kann.

User Agent

Identifizierung einer Software, die auf Webserver zugreift.

UUID

Universally Unique IDentifier, Bezeichner, dessen Ziel es ist, Informationen und Ressourcen in verteilten Systemen ohne zentrale Koordination eindeutig kennzuzeichen.

12. Index

—— Notizen ——

—— Notizen ——

—— Notizen ——