**Fachhochschule Aachen**
Campus Jülich

Fachbereich: Medizintechnik und Technomathematik
Studiengang: Scientific Programming



# Entwicklung einer REST-Schnittstelle zum Zugriff auf Bibliotheksinformationssysteme unter Berücksichtigung des Einsatzes im mobilen Umfeld

Bachelorarbeit von

**Jannis Konopka**

Jülich, August 2017

## Erklärung

Diese Arbeit ist von mir selbstständig angefertigt und verfasst. Es sind keine anderen als die angegebenen Quellen und Hilfsmittel benutzt worden.

_____

Unterschrift

Die Arbeit wurde betreut von:

Erstprüfer: Prof. Dr. rer. nat. Volker Sander
Zweitprüfer: Dipl. Phys. Ing. Waldemar Hinz

Diese Arbeit wurde erstellt in der
Zentralbibliothek (ZB)
des Forschungszentrums Jülich

## Zusammenfassung

Für die Zentralbibliothek des Forschungszentrums Jülich ist die Aktualität des eigenen Angebots ebenso wichtig, wie die Verwendung zeitgemäßer Technik zur Präsentation dieser Inhalte. Deshalb ist es nun an der Zeit, die Services der Bibliothek für alle denkbaren Anwendungsprogramme nutzbar zu machen.

Vorstellbare mobile Anwendungen sollen nicht direkt mit den Bibliothekssystemen kommunizieren, sondern auf eine wohldefinierte Schnittstelle zurückgreifen. Im Rahmen dieser Arbeit wird deshalb eine serverseitige REST-API entwickelt, um die wichtigsten bibliothekseigenen Informationsquellen einheitlich ansprechen zu können. Zu diesen Quellen gehören die Datenbanksysteme mit erworbener Literatur, sowie die im Forschungszentrum publizierten Schriften. Ebenso soll die Schnittstelle den Zugriff auf das Benutzerkontenverwaltungssystem und die Abfrage nach der Verfügbarkeit schriftlicher Volltext-Dokumente ermöglichen.
Die generellen Anforderungen an die API sind zum einen performante Antwortzeiten und zum anderen eine intuitive Nutzbarkeit, sodass die Schnittstelle auch von anderen Entwicklern ohne viel Einarbeitungszeit verwendet werden kann. Die Kommunikation zwischen den möglichen Anwendungen und der API wird auf Basis des Client-Server Modells realisiert. Der Datenaustausch der beteiligten Systeme soll außerdem sicher über das Internet erfolgen, damit lizenzierte Inhalte geschützt übermittelt werden können. Zu diesem Zweck wird das sichere HTTPS-Protokoll zwischen Client und API benutzt. Um dem möglichen Datenverlust über eine Internetverbindung entgegenzuwirken, wird der Kommunikation das TCP-Protokoll zugrunde gelegt.
Da die Schnittstelle vermehrt im mobilen Bereich zum Einsatz kommen soll, wird bei der Entwicklung ein spezielles Augenmerk auf die Bandbreite gelegt.

# Inhaltsverzeichnis

# Abbildungsverzeichnis

# Tabellenverzeichnis

# Kapitel 1

# Einleitung

## 1.1 Motivation und Problemanalyse

Die Zentralbibliothek des Forschungszentrums Jülich ist für eine hochwertige Literatur- und Informationsversorgung der Mitarbeiter verantwortlich. Dazu gibt es in der Bibliothek eine Vielzahl von IT-Diensten, denen verschiedene Aufgaben zukommen. So gibt es beispielsweise den Bibliothekskatalog Symphony, die Suchmaschine Solr und ein System für Literaturbestellungen. Zur Nutzung eines Großteils dieser Services existiert bereits ein Browser-Front-End, also eine Web-Anwendung.

In der heutigen Zeit, die vor allem durch Apps geprägt ist, reicht eine einfache Web-Anwendung aber nicht mehr aus, um alle Kunden zufrieden stellen zu können. Deshalb möchte die Bibliothek nun ihr Anwendungsangebot erweitern, um in diesem Bereich dem aktuellen Stand der Technik zu entsprechen. Vorstellbar wären vor allem mobile Anwendungen, die die Nutzung der Bibliothek weiter vereinfachen würden. Interessante Projekte könnten hier das selbständige Ausleihen von Medien durch den Nutzer über eine Barcode-Scanner-App, oder die sprachgesteuerte Nutzung der Bibliotheksfunktionen sein. Darüber hinaus wären auch Systeme vorstellbar, die die Navigation in der Bibliothek verfügbar machen könnten. Die bisherige IT-Landschaft der Bibliothek ist jedoch auf ein solches Vorhaben noch nicht ausgelegt.
Weil sich die Dienste hinsichtlich der Aufgaben, der Ansprache und der Antworten unterscheiden, ist eine stark heterogene Service-Landschaft gegeben, für die es keine einheitliche Kommunikationsmöglichkeit gibt.
Da die einzelnen Services hinter einer Firewall liegen, ist das direkte Ansprechen durch einen nutzenden Client, der von außerhalb des Campus-Netzes zugreifen möchte, ebenfalls nicht trivial.
Das Front-End JuLib eXtended hat diese Probleme zwar für sich schon gelöst, doch ist

diese Lösung ausschließlich für JuLib gültig und kann nicht einfach auf andere Anwendungsprogramme übertragen werden. JuLib kann allerdings als Wrapper genutzt werden, um innerhalb der Zentralbibliothek auf einen Teil der zugrunde liegenden Dienste zuzugreifen. Dafür existiert eine Schnittstelle, die allerdings keinen Standards entspricht und nur bibliotheksintern nutzbar ist.

Um somit Anwendungsentwicklern eine einheitliche Schnittstelle bieten zu können, die auch unabhängig vom Standort der Clients genutzt werden kann, muss eine verständliche und gut dokumentierte API (Application Interface) entwickelt werden, die sich zusätzlich um die Einhaltung der Zugriffsrechte kümmert. Im Folgenden wird genau auf diese speziellen Anforderungen und Zielvorstellungen eingegangen.

## 1.2 Anforderungen und Zielvorstellung

Durch die genannten Probleme entstehen Anforderungen an die zu entwickelnde Schnittstelle. Die wichtigste Eigenschaft ist der einheitliche Zugriff auf verfügbare Bibliotheksinformationssysteme. Kann dieses Ziel nicht erreicht werden, wäre die wichtigste Kernfunktion nicht erfüllt und die Schnittstelle würde keinen Mehrwert zur bestehenden Situation bringen.
Darüber hinaus sollten sich Veränderungen an den zugrundeliegenden Systemen nicht auf die Nutzung der Schnittstelle auswirken. Die Schnittstelle sollte also auch bei Veränderungen abwärtskompatibel bleiben, damit bereits entwickelte Clients ihren Code nicht anpassen müssen, um weiterhin zu funktionieren.
Sollten neue Systeme in der Bibliothek zum Einsatz kommen, müssen diese leicht in die bestehende Schnittstelle eingebunden werden können. Es ergibt sich also die Anforderung der leichten Erweiterbarkeit.
Des Weiteren muss der Datenaustausch auf standardisierten Formaten basieren, um plattform- sowie programmiersprachenunabhängige Kommunikation zu erreichen.
Das konkret verwendete Datenmodell sollte zwischen Schnittstelle und Client dynamisch ausgehandelt werden können, um die optimale Programmiererfreundlichkeit der Schnittstelle gewährleisten zu können.
Da die Schnittstelle vermehrt im mobilen Bereich zum Einsatz kommen soll, ist es wichtig, ein Augenmerk auf die übertragenen Datenmengen zu legen und die begrenzte Bandbreite von Smartphones zu berücksichtigen. Ein guter Eindruck von der Bibliothek fängt schon mit kurzen Wartezeiten am Schalter an, zieht sich aber hin bis zu kurzen Ant-

wortzeiten der angebotenen Websites und Smartphoneanwendungen.
Damit verbunden ist eine sitzungsfreie und zustandslose Kommunikation. Die Schnittstelle, also der Server, sollte es vermeiden, Informationen über die zugreifenden Clients zu speichern. Ließe man diesen Punkt außer Acht, müsste man bei jeder Anfrage zuerst den Zustand des Clients prüfen, um im Anschluss die auf den Zustand passende Antwort zu senden. Im Fall der zustandslosen Kommunikation erhält der Client nach einer Anfrage eine verlässliche, gleichbleibend strukturierte Antwort.
Jede Anfrage muss für sich eigenständig sein und es bedarf keiner Abfrage-Reihenfolge, die von den Clients eingehalten werden muss. Dadurch kann eine Lastenverteilung auf einer verteilten Serverarchitektur vorgenommen werden, die zum einen die Antwortzeit des Servers verkürzt und zum anderen die Ausfallsicherheit des Gesamtsystems verbessert.
Auch wenn ein solches lastenverteilendes System den Rahmen dieser Arbeit vermutlich sprengen würde, ist es jetzt schon wichtig, den Grundstein für ein solches Vorhaben zu legen.
Anders als bei den bereits bestehenden Systemen, ist für die geplante Schnittstelle eine gute Dokumentation vorgesehen. Sie soll Client-Entwicklern die Einarbeitung vereinfachen. Darüber hinaus kann durch die Verwendung von Hypermedia eine Selbstdokumentation der Schnittstelle erreicht werden. Ausgehend von einem Einstiegspunkt werden dem Client die nächsten logischen Schritte aufgezeigt. Dadurch kann der Client eigenständig durch den Service navigieren, ohne statische Links kennen zu müssen [Fie00].
Um die Lizenzbestimmungen von Verlegern einzuhalten, darf die Schnittstelle nur von authentifizierten Clients benutzt werden. Es muss deshalb ein sicherer Weg der Datenübertragung gewährleistet werden.

| Die Schnittstelle sollte: |
| --- |
| **möglichst viele Informationssysteme abdecken** |
| **einen einheitlichen Zugriff gewährleisten** |
| **abwärtskompatibel bleiben** |
| **leicht erweiterbar sein** |
| **Kommunikation durch standardisierte Formate bieten** |
| **geringe Bandbreite nutzen** |
| **zustandslose Kommunikation unterstützen** |
| **leicht verständliche Dokumentation bereitstellen** |
| **nur authentifizierten Nutzern Zugriff bieten** |

Tabelle 1.1: Anforderungen an die zu entwickelnde Schnittstelle der Zentralbibliothek

Durch die Bereitstellung einer solchen Schnittstelle in Form eines Web-Services, der die genannten Anforderungen erfüllt, kann die Kommunikation zwischen bibliothekseigenen Informationssystemen und Client-Anwendungen serialisiert und vereinheitlicht werden. Dadurch verspricht sich die Zentralbibliothek eine einfache Anbindung ihrer Dienstleistungen in neuartigen Anwendungsprogrammen und eine Erweiterung des bestehenden Angebotes. Vor allem im mobilen Umfeld soll die Schnittstelle in Zukunft eingesetzt werden.

# Kapitel 2

# Grundlagen

Dieses Kapitel setzt sich mit den Grundlagen für die Entwicklung einer Web-Schnittstelle auseinander. Dazu wird auf das zugrunde liegende HTTP-Protokoll eingegangen. Die dann folgenden Abschnitte sollen einen Überblick über die Implementationsvarianten möglicher Web-Schnittstellen geben. Dazu wird zuerst eine Einführung in die Grundlagen der Web-Services dargestellt, um darauf aufbauend den REST-Architekturstil zu beleuchten. Im Anschluss werden die Unterschiede zwischen REST und dem Netzwerkprotokoll SOAP betrachtet, wobei auch deren jeweilige Einsatzgebiete zur Sprache kommen.

## 2.1 HTTP

HTTP steht für „Hypertext Transfer Protocol“ und ist ein zustandsloses Protokoll zur Übertragung von Nachrichten über ein Netzwerk.
Es zählt zur Anwendungsschicht des ISO/OSI-Modells und ist das Protokoll, auf dem die Webbrowser-Webserver-Kommunikation aufbaut.

Bei der Kommunikation zwischen Client (z.B. Webbrowser) und Webserver werden zwei verschiedene Nachrichtentypen eingesetzt. Wenn der Client eine Anfrage an den Server verschickt, handelt es sich dabei um einen Request. Sendet der Server wiederum eine Antwort, ist das die Response-Nachricht.
Eine HTTP Nachricht besteht aus einem Header und einem Body. Der Header enthält generelle Informationen über die Art der Nachricht, während im Body die Daten gehalten werden. Zur Verarbeitung der Nachricht wird zuerst der Header interpretiert, um Nachrichtentyp, Datenformat, Kompressionsalgorithmus und Länge der Nachricht auszulesen. Mit diesen gewonnen Informationen kann anschließend der Inhalt des Bodys

analysiert und verarbeitet werden.

| GET / HTTP/1.1 |
| --- |
| Host: example.com |
| Accept: application/json;q=0.9,application/xml;q=0.8 |
| |
| *leerer Body* |

Tabelle 2.1: Aufbau einer HTTP-GET-Request Nachricht

| HTTP/1.1 200 OK |
| --- |
| Content-Type: application/json |
| Content-Length: 98 |
| |
| *Response JSON data...* |

Tabelle 2.2: Aufbau einer HTTP-Response Nachricht

Ein Client könnte nun zum Beispiel eine Ressource (2.3.1) eines Webservers mithilfe einer Request-Nachricht anfragen. Dazu spezifiziert er in der ersten Zeile des HTTP-Headers die eingesetzte HTTP-Methode, den Pfad unter der die Ressource zu finden ist und die Version des verwendeten HTTP-Protokolls. Die zweite Zeile legt die Webadresse des hostenden Systems fest. Nachfolgend können unter anderem akzeptierte Dateiformate und Datenkodierungsalgorithmen festgelegt werden. Eine Leerzeile signalisiert das Ende des Headers und den Beginn des Bodys. Eine zweite Leerzeile bedeutet das Ende einer HTTP Nachricht.
Bei einer normalen Ressourcenanfrage wird die GET-Methode verwendet und der Body der Request Nachricht bleibt leer. Es ist die Methode, die immer dann genutzt wird, wenn im Webbrowser eine URL in die Adresszeile getippt und abgeschickt wird. Neben der am meisten genutzten GET-Methode existieren allerdings noch weitere Methoden.

### 2.1.1 Methoden

HTTP bietet fest definierte Standard-Methoden, denen verschiedene Aufgaben zukommen.

| HTTP Method | Safe | Idempotent |
|---|---|---|
| GET | ✓ | ✓ |
| POST | ✗ | ✗ |
| PUT | ✗ | ✓ |
| DELETE | ✗ | ✓ |
| OPTIONS | ✓ | ✓ |
| HEAD | ✓ | ✓ |

Abbildung 2.1: Die wichtigsten HTTP-Methoden in der Übersicht [Sto13]

Wie in der Abbildung 2.1 zu sehen, legt die Spezifikation des HTTP-Protokolls sechs Hauptmethoden fest.

- **GET** bietet lesenden Zugriff auf eine Ressource und liefert eine Repräsentation der angefragten Ressource im Body der Response-Nachricht. (siehe 2.3.1 für Begriffserklärungen)
- **POST** ändert oder erzeugt eine nicht genauer spezifizierte Ressource.
- **PUT** ändert eine konkrete Ressource, wenn sie bereits vorhanden ist. Sonst erzeugt PUT eine neue Ressource genau auf der URI, der der Aufruf gilt.
- **DELETE** löscht eine Ressource.
- **OPTIONS** liefert die Methoden, die eine Ressource unterstützt.
- **HEAD** liefert nur den Header einer GET-Anfrage.

Mit den Methoden sind verschiedene Eigenschaften verknüpft.
Safe bedeutet hierbei, dass die Verwendung der Methode auf eine Ressource keine Seiteneffekte, wie beispielsweise Änderungen, nach sich zieht. In diese Kategorie der sicheren Methoden fallen GET, OPTIONS und HEAD, da sie nur lesend auf eine Ressource zugreifen. POST, PUT und DELETE verändern Ressourcen. Deshalb werden sie als unsicher eingestuft.

Eine idempotente Methode garantiert, dass das mehrmalige Ausführen der Methode mit denselben Parametern immer den gleichen Endzustand der Ressource erzeugt. Es ist also egal, wie oft beispielsweise ein DELETE auf eine Ressource ausgeführt wird. Der Endzustand ist immer der gleiche - die Ressource ist gelöscht. Genauso idempotent wie DELETE verhalten sich GET, PUT, OPTIONS und HEAD. Die einzige Ausnahme bildet POST, weil weder für das ungezielte Erzeugen noch für das beliebige Ändern einer Ressource eine Garantie gegeben werden kann.
Diese Konventionen sind bei der Entwicklung von Web-Services stets zu beachten, um die Erwartungen, die mit den Methodennamen verknüpft sind, erfüllen zu können.

### 2.1.2 Statuscodes

HTTP sendet auf die genannten Methoden in der ersten Zeile einer Response-Nachricht Status Codes. Dadurch wird der nachfragenden Anwendung über eine dreiziffrige Zahl mitgeteilt, wie die Anfrage ausgegangen ist. Dabei kann schon anhand der ersten Ziffer ausgelesen werden, welcher Kategorie die gesamte Nachricht angehört.

| ***The following are the list of HTTP response status codes*** | | |
|---|---|---|
| Type | Status Codes | Examples |
| Informational | 1xx | 100 Continue, 101 Switching Protocols |
| Success | 2xx | 200 - OK , 201 - Created, 202 Accepted |
| Redirection | 3xx | 300 Multiple Choices, 301 Moved Permanently, 302 - Found |
| Client Error | 4xx | 400 Bad Request, 403 - Forbidden , 404 - Not Found , 422 - Unprocessable Entity |
| Server Error | 5xx | 500 - Internal Server Error , 503 - Service Unavailable |

Abbildung 2.2: Die wichtigsten HTTP-Statuscodes in der Übersicht [Eas15]

## 2.2 Web-Services

Dieses Kapitel klärt die Frage, was ein Web-Service ist, wozu ein Web-Service im Vergleich zu einer Web-Anwendung dient und welche Einsatzgebiete es heutzutage für einen Web-Service gibt.

Ein Web-Service ist eine Schnittstelle zur Kommunikation zwischen einem Server und

Clients. Der Service stellt Dienste zur Verfügung, die von überall abgerufen werden können. Er steht meistens vor umfangreichen IT-Systemen und kapselt die Funktionen durch eine Firewall nach außen. Er kümmert sich um die Berechtigungen und liefert sensible Daten nur an autorisierte Clients.
Im Vergleich zu einer Web-Anwendung steht nicht die Nutzung durch einen Web-Browser, also direkt durch einen Menschen, im Vordergrund. Es wird vielmehr Wert auf die automatische Verwertbarkeit der Server-Antworten gesetzt, als auf grafisch aufbereitete Repräsentationen im Web-Browser. Deshalb läuft die Kommunikation zumeist über standardisierte Formate ab. Neben vielen anderen existierenden sind die beliebtesten Formate XML (Extensible Markup Language [Sel17]) und JSON (JavaScript Object Notation [JSO17]).
Die nach einem geregeltem Muster verlaufende Kommunikation kann dann in speziellen Clients eingebaut werden. Diese Clients können zum Beispiel Smartphone Apps, Desktop-Anwendungen oder auch eingebettete Systeme sein, die im Internet der Dinge einen immer größeren Stellenwert einnehmen.

Vereinheitlicht ein Unternehmen die Schnittstelle zu seinen eigenen Angeboten als Web-Service, erleichtert es somit also die Integration seiner Inhalte in aktuelle Softwaresysteme. Außerdem wird durch diesen Schritt die Portabilität zwischen Geräten und Programmiersprachen ermöglicht und der erreichbare Kundenkreis erweitert.
Wird dann noch eine geeignete Dokumentation zum Web-Service zur Verfügung gestellt, können auch Anbieter anderer Softwaresysteme den Web-Service als Drittanbieter integrieren. Dieses Phänomen ist derzeit beispielsweise bei Twitter zu beobachten. Mithilfe der Twitter-API lassen sich in Drittanbieter-Apps erfolgreich Tweets absetzen [Twi17].

### 2.2.1 Web-API

Das Themengebiet der Web-Services lässt sich nun weiter spezifizieren. Web-Services stellen den Überbegriff zu allen Diensten dar, die über das Internet zum serialisierten Datenaustausch angesprochen werden können. Ein Teilgebiet dieser Thematik sind die sogenannten Web-APIs. API steht für Application Interface und symbolisiert im Zusammenhang mit Web, dass der betreibende Server eine global erreichbare, atomare und ressourcenschonende Schnittstelle anbietet. Die klassische Implementierung einer Web-API findet nach dem Architekturstil REST statt.

## 2.3 Architekturstil REST

Der Representional State Transfer liefert ein Programmierparadigma zur Erstellung von RESTful (REST-konformen) Web-Services, den sogenannten Web-APIs. Dieser Architekturstil wurde im Jahr 2000 von Roy Fielding im Rahmen einer Dissertation begründet [Fie00].
Bei der Kommunikation zwischen Server und Client wird auf die Ausnutzung der vorhandenen Web-Architektur geachtet und sich auf den Ursprung einzelner Web-Komponenten zurück besinnt.
Dazu zählt wieder das HTTP-Protokoll, das in vielen aktuellen Anwendungen nicht optimal genutzt wird und im REST-Sinne voll ausgereizt werden sollte.
Was das konkret bedeutet, findet sich in den folgenden Untersektionen.

### 2.3.1 Ressourcen und Repräsentation

Die Ressourcen bilden das Herzstück einer Web-API. Dieses sind die Informationselemente, die über einen eindeutigen Universal-Resource-Identifier (URI) im World-Wide-Web von überall zugänglich sind und im Idealfall über Jahre hinweg adressierbar bleiben.
Jede Ressource sollte nach dem REST-Stil so atomar sein, dass das Aufrufen der einfachen CRUD-Funktionen (Create, Read, Update & Delete) sinnvoll ist.
Losgelöst von der Ressource ist ihre Repräsentation. Ein Informationselement kann in verschiedenen Darstellungsformen an den Nutzer übertragen werden. Gängige, standardisierte Repräsentationsformate sind neben vielen anderen JSON, XML und HTML. Zu beachten ist, dass immer nur eine Repräsentation einer Ressource zum Nutzer geschickt wird, nie die Ressource selbst. Dadurch bewahrt sich die Ressource vollständige Integrität und Kontrolle über ihren Zustand [Spi17, Kapitel 8.3].

### 2.3.2 Content Negotiation

Das Verfahren zur dynamischen Aushandlung des Repräsentationsformates, in der die Darstellung einer Ressource erfolgen soll, nennt sich Content-Negotiation. Der Client beschreibt in seiner Anfrage an den Server im Accept-Header, welche Repräsentationsformate er akzeptiert. Dabei können den einzelnen Formaten verschiedene Prioritäten zugewiesen werden. Der Server legt, basierend auf dem Accept-Header und den Formaten, die er beherrscht, fest, welche Repräsentation zum Datenaustausch benutzt wird.

### 2.3.3 HATEOAS

HATEOAS ist eine Abkürzung für Hypermedia-As-The-Engine-Of-Application-State und eines der wichtigsten REST-Paradigmen.
Hypermedia bildet den Zustandsautomaten einer Ressource. Was Hypermedia konkret ist und wie der Zustandsautomat funktioniert, findet sich in den folgenden Unterpunkten.

#### Hypermedia

Der Hypermedia-Aspekt einer REST-konformen Schnittstelle erwartet von jeder Repräsentation eine Liste mit URIs, die im Kontext der übertragenen Daten sinnvoll ist. Clients können diese URIs nutzen, um sich, wie Menschen über eine Website, durch den Service bewegen zu können. Sie müssen lediglich einen Einstiegspunkt in den Web-Service kennen. Die übrige Navigation geschieht über dynamisch übertragene URIs.
Der sich daraus ergebende Vorteil ist die Befreiung der Clients von statischen URIs. Dadurch sind Veränderungen an den URIs des Web-Services durchführbar, ohne dass die Clients ein Update benötigen.

#### Hypermedia als Zustandsautomat

Als Zustandsautomat wird ein Knoten-System von Zuständen bezeichnet, zwischen denen es ein Kanten-System von Übergängen gibt. In der folgenden Grafik ist solch ein allgemeiner Zustandsautomat mittels Unified Modelling Language (UML) beschrieben.

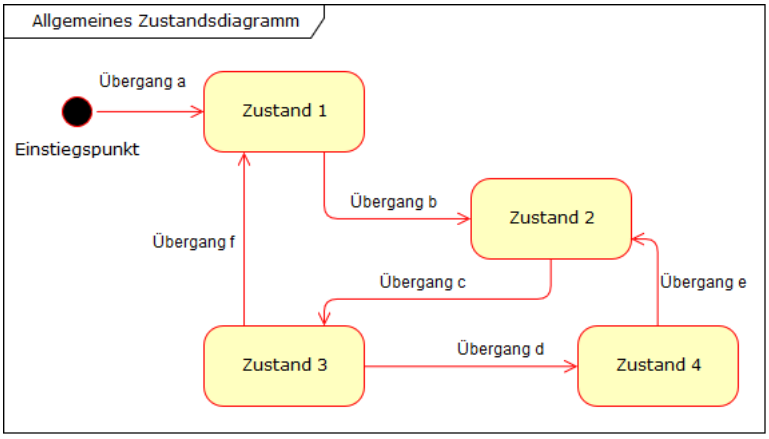


Abbildung 2.3: UML-Diagramm eines allgemeinen Zustandsautomaten

Nun übertragen wir dieses allgemeine Konzept auf die REST-Web-API.

Mit „Zustand“ ist nun eine Ressource der REST-API gemeint. Der Übergang zwischen zwei Zuständen entspricht dem Folgen einer vom Server bereitgestellten URI. Der Einstiegspunkt entspricht der einzigen bekannten URI der Web-API, die als Ausgangspunkt dient.

Die Gesamtheit aller angebotenen Ressourcen und bereitgestellten Übergänge stellt den Zustandsautomaten einer REST-Web-API dar. Für den Server ist es egal, in welcher Reihenfolge die Anfragen durchgeführt werden. Er befindet sich zu keinem Zeitpunkt in einem Zustand.
Die gesamte Zustandshaltung geschieht auf dem Client, indem er dem Ablauf des Zustandsautomaten folgt. Mit diesem Wissen lässt sich nun auch die Herkunft des Wortes REST erschließen. „Representional State Transfer“ steht für die Übertragung des Client-Zustands in der Repräsentation.

Das folgende Beispiel einer fiktiven Web-API zeigt ein einfaches Bestellsystem.

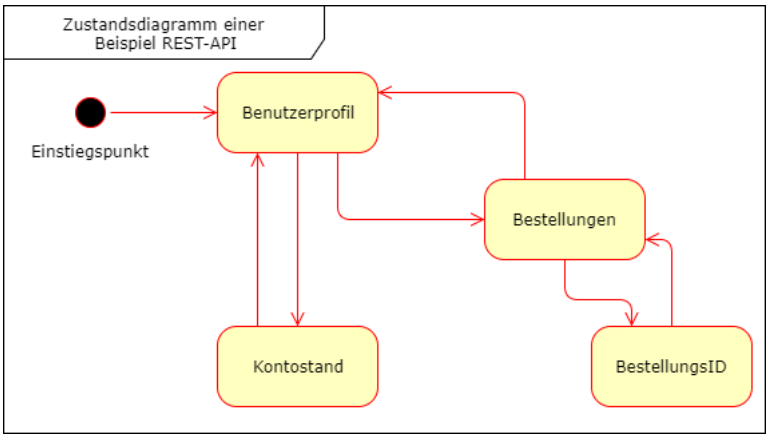


Abbildung 2.4: Beispiel eines konkreten Zustandsautomaten für eine fiktive Web-API

Der Einstiegspunkt führt den Nutzer zum Benutzerprofil. Dort hat dieser zwei Möglichkeiten, weitere Ressourcen zu besuchen. Er kann zu seinem Kontostand, oder zu seinen Bestellungen gelangen. Wählt er die Ressource Bestellungen, könnte er sich im nächsten Schritt eine konkrete Bestellung genauer ansehen.
Die URIs für diese Aktionen erhält der Client über die Repräsentation der jeweiligen Res-

source. Genauso verhält es sich mit den URIs, die ihn wieder eine Ebene zurückführen. Zur Übertragung und Formatierung dieser Links und Relationen innerhalb von Repräsentationen gibt es verschiedene Standards und Möglichkeiten. So existiert beispielsweise die Hypertext Application Language (HAL [Kel13]) für JSON und XML. Sie definiert das „*_links*“-Attribut in Repräsentationen. Darin können Relationen eingetragen werden, die href-Attribute mit Links auf andere Ressourcen enthalten.

Das Problem, das sich für Web-API Entwickler stellt, liegt in der Auswahl des Formats. Es gibt nicht **den** einheitlichen Standard, auf den sich alle Anbieter geeinigt haben [Sto15]. Unternehmen wie zum Beispiel Paypal entwickeln ihren eigenen Standard, der leicht von HAL abweicht [Pay17]. Für jede Verlinkung wird eine konkrete URI, Relation und Methode angegeben. Sie befinden sich in einem Array „links“ in der Repräsentation einer Ressource.

```
{
  "links": [{
    "href":
"https://api.paypal.com/v1/payments/sale/36C38912MN9658832",
    "rel": "self",
    "method": "GET"
  }, {
    "href":
"https://api.paypal.com/v1/payments/sale/36C38912MN9658832/refund",
    "rel": "refund",
    "method": "POST"
  }, {
    "href": "https://api.paypal.com/v1/payments/payment/PAY-
5YK922393D847794YKER7MUI",
    "rel": "parent_payment",
    "method": "GET"
  }]
}
```

Abbildung 2.5: PayPals Umsetzung von HATEOAS

Für Client-Entwickler bedeutet das, dass sie, je nachdem welche API sie ansprechen möchten, mit anderen Bereitstellungsvarianten für Links und Relationen umgehen müssen.

### 2.3.4 Zustandslose Kommunikation

Der Verzicht auf sitzungs- und sessionbasierte Webprogrammierung bringt einige Vorteile mit sich.
So ergibt sich für den gesamten Dienst eine bessere Skalierbarkeit, da sich nicht dieselbe

Serverinstanz um zwei aufeinanderfolgende Anfragen kümmern muss. Der Server interessiert sich für einen Client nur noch für die Dauer einer Anfrage.
Einen Ausweg aus Sitzungen bietet die Umwandlung eines Zustands in eine neue Ressource. Das ist beispielsweise dann der Fall, wenn ein neu angelegter Warenkorb in einem Bestellsystem als Ressource zugänglich gemacht wird.
Durch diese Vorgehensweise werden Daten nicht über einen längeren Zeitraum ungespeichert vorgehalten, sodass auch eine mögliche Serverwartung keinen Datenverlust mit sich bringen würde.

## 2.4 REST versus SOAP

Um die Unterschiede zwischen REST und SOAP genau analysieren zu können, folgt zuerst eine Erklärung von SOAP.

SOAP ist ein Netzwerkprotokoll zur Datenübertragung und seit 1999 ein Standard für Web-Services. Chronologisch gesehen gab es also den SOAP-Ansatz für Web-Services vor dem REST-Ansatz für das Teilgebiet der Web-APIs.
SOAP ist plattform-, programmiersprachen- als auch protokollunabhängig und legt ein XML Schema fest, an das sich Server und Client halten müssen. Dieser Vertrag wird in einer XML-basierten WSDL-Datei (Web-Service Description Language) festgehalten. Darin sind die Parameter und Rückgabewerte der vorhandenen Methoden genau beschrieben und müssen exakt definierten Standards entsprechen.
SOAP ermöglicht durch diese Methoden das Aufrufen von Funktionen auf entfernten Maschinen. Dieses Verfahren nennt sich Remote Procedure Calls (RPCs), da die beiden beteiligten Systeme physikalisch voneinander getrennt sind und der Aufruf über ein Netzwerk geschieht. RPC ist ein wichtiges Message Exchange Pattern, das bei SOAP zum Einsatz kommt. Es existieren jedoch noch weitere.
Die Kommunikation von SOAP erfolgt mittels XML Nachrichten, die zumeist aber nicht zwingend über das HTTP-Protokoll übertragen werden. Der Aufbau der XML Nachricht ist fix festgelegt. Oftmals sind diese Nachrichten sehr lang, obwohl nur eine kurze Information ausgetauscht werden soll. Dadurch ist SOAP meist langsamer als andere Middleware Standards und benötigt eine größere Bandbreite.

Generell besteht eine SOAP Nachricht aus einem XML Gerüst. Darin wird ein soap:Envelope (Briefumschlag) Element definiert, das wiederum einen Header und einen Body enthält.

Während im Header Authentifizierungsinformationen oder andere optionale anwendungsspezifische Inhalte untergebracht werden können, ist der Body für die Haltung der Methodennamen und Übergabeparameter zuständig. Es ergibt sich demnach folgende Struktur:

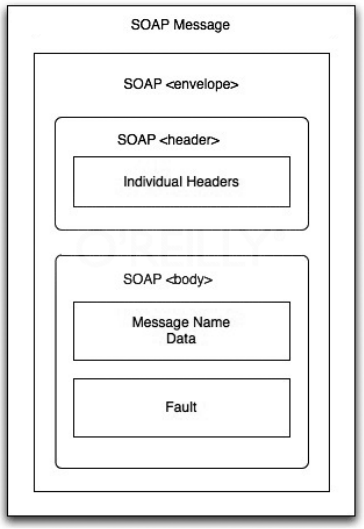


Abbildung 2.6: Struktur einer SOAP Nachricht [Mic09]

Nun folgt ein Vergleich zwischen SOAP und REST anhand eines Beispiels.

Die nachstehende, über HTTP übertragene SOAP-Anfrage eines Clients, stellt einen entfernten Methodenaufruf dar [w3s].

```
POST /InStock HTTP/1.1
Host: www.example.org
Content-Type: application/soap+xml; charset=utf-8

<?xml version="1.0"?>
<soap:Envelope
    xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap
    -encoding">
    <soap:Body
        xmlns:m="http://www.example.org/stock">
        <m:GetStockPrice>
            <m:StockName>Raspberry Pi</m:StockName>
        </m:GetStockPrice>
    </soap:Body>
</soap:Envelope>
```

Abbildung 2.7: SOAP Request

```
HTTP/1.1 200 OK
Content-Type: application/soap+xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>

<soap:Envelope
xmlns:soap="http://www.w3.org/2003/05/soap-envelope/"
soap:encodingStyle="http://www.w3.org/2003/05/soap-encoding">

<soap:Body xmlns:m="http://www.example.org/stock">
  <m:GetStockPriceResponse>
    <m:Price>34.5</m:Price>
  </m:GetStockPriceResponse>
</soap:Body>

</soap:Envelope>
```

Abbildung 2.8: SOAP Response

Mit diesem speziellen SOAP-XML-Format könnte hier der Preis eines Produktes erfragt werden, das sich beispielsweise im Lager eines Versandunternehmens befindet. Dazu

wird auf dem Server die Methode aufgerufen, die unter dem Namensraum http://www.example.org/stock bei GetStockPrice definiert ist. Die Antwort des Servers enthält dann im GetStockPriceRespone Element ein Price Tag.

Die gleiche sinngemäße Abfrage über einen REST-Service könnte wie folgt aussehen.

```
GET /InStock/RaspberryPi/Price HTTP/1.1
Host: www.example.org
Content-Type: application/xml; charset=utf-8
Content-Length: nnn
```

Abbildung 2.9: REST Request

```
HTTP/1.1 200 OK
Content-Type: application/xml; charset=utf-8
Content-Length: nnn

<?xml version="1.0"?>
<Price
    xmlns:i="http://www.w3.org/2001/XMLSchema-instance"
    xmlns="http://schemas.datacontract.org/2004/07/StockAPI.Models.Price">
    34.5
</Price>
```

Abbildung 2.10: REST Response

Eine REST-API würde das Produkt aus dem Lager des Versandunternehmens, in diesem Beispiel den Raspberry Pi, als Ressource anbieten. Eine Subressource könnte das Nomen Price sein. Der Client würde dann lediglich eine GET-Methode auf der URI /InStock/RaspberryPi/Price aufrufen und bekäme eine abgespeckte XML-Nachricht im Body der HTTP-Nachricht zurück.
Wie zu sehen ist, reizen SOAP Web-Services durch ihre Protokollunabhängigkeit das HTTP-Protokoll nicht vollständig aus. Sie steuern ihre Kommunikation ausschließlich über POST Methoden, auch, wenn, wie in diesem Beispiel, nur ein lesender Zugriff stattfindet. Da SOAP nicht zwingend an HTTP gebunden ist, können auch andere Protokolle verwendet werden. Doch auch diese dienen nur als weitere Verpackung für das SOAP:Envelope Element und werden dadurch ebenfalls nicht nach ihrer eigentlichen Intention eingesetzt.
Der REST-Architekturstil versucht hingegen die HTTP-Methoden optimal anzuwenden und die durch die Spezifikation versprochenen Garantien einzuhalten. Dadurch werden kürzere Nachrichten erreicht, die leichter und schneller zu analysieren sind.

SOAP-Services stellen Prozeduren zur Verfügung und geben sie über ein WSDL-Dokument bekannt. Der Client übernimmt dann die Kontrolle und ruft die Prozeduren auf der Servermaschine eigenständig auf.
Das Fehlen eines solchen WSDL-Dokuments bei REST-Services könnte als Nachteil angesehen werden, da der Anwedungsentwickler, der die Schnittstelle implementieren möchte, die URI-Nutzung nicht auf Anhieb verstehen kann. Doch gerade bei diesem Punkt setzt

das Hypermedia-Konzept an. Die Clients benötigen lediglich den Einstiegspunkt zum Service. Die weiteren Ressourcen und verfügbaren CRUD-Methoden werden über das HATEOAS-Konzept dynamisch mitgeteilt.
Darüber hinaus kennen nur die Ressourcen selbst ihre internen Prozeduren. Sie geben lediglich Repräsentationen heraus und entscheiden selbst, wie sich Änderungen an den Repräsentationen auf ihre Ressourcenzustände auswirken [Spi17, Kapitel 8.5]. Dadurch behalten REST-Systeme stets die Kontrolle.
Sowohl REST als auch SOAP koppeln zwei Systeme miteinander, um eine Kommunikation zu ermöglichen. Diese Kopplung ist in beiden Fällen lose, da die Clients nicht direkt von den Implementierungsdetails des Servers abhängig sind. Die entscheidenden Unterschiede, wie die Bereitstellung einer einheitlichen Schnittstelle und der Einsatz von Hypermedia, die der REST-Ansatz im Gegensatz zu SOAP hervorhebt, reduzieren die Kopplung zwischen dem REST-Service und seinen Clients noch weiter [Til15, Kapitel 1.1.1].
In der folgenden Tabelle 2.3 ist der Vergleich noch einmal zusammengefasst.

| | REST | SOAP |
|---|---|---|
| **Datensparsamkeit** | + | - |
| **nutzbare Datenformate** | viele (JSON, XML etc.) | nur XML |
| **Schnittstellenbeschreibung** | fehlt, dafür HATEOAS ab Einstiegspunkt | Web Service Description Language |
| **Service-Aufruf** | Aufruf von URIs<br>Bereitstellung durch HATEOAS | Aufruf durch Message Exchange<br>Patterns wie RPCs |
| **Hypermedia** | + | - |
| **lose Client-Server Kopplung** | ++ | + |

Tabelle 2.3: Vergleich zwischen REST und SOAP auf einen Blick

# Kapitel 3

# Informationssysteme der Zentralbibliothek

In diesem Kapitel werden die einzelnen Informationssysteme der Zentralbibliothek im Zusammenhang mit ihren Anwendungsgebieten vorgestellt. Zudem werden die Schnittstellen, welche diese Informationssysteme für den Datenaustausch zur Verfügung stellen, beschrieben.

## 3.1 Bibliothekssystem Symphony

Das Bibliothekssystem, ein sogenanntes Integrated Library System (ILS), bildet das Backend-Softwaresystem einer Bibliothek. Dem Bibliothekssystem fallen drei wichtige Aufgaben zu.
Als Erstes ist es für die Speicherung der Meta-Daten von Büchern, eBooks, Zeitschriften und anderen angebotenen Medien zuständig. „Meta-Daten“ sind dabei generelle Informationen über ein Medium, beispielsweise Titel, Erscheinungsdatum, Verleger und Autor. Ebenfalls Teil der Meta-Daten sind die Schlagwörter. Das sind die Nomen, die den Inhalt des Mediums am besten beschreiben und die später maßgeblich zum Auffinden des Mediums beitragen.
Die zweite Verantwortlichkeit liegt bei den Benutzerdaten. Das sind die Informationen, die die Nutzer der Bibliothek genau beschreiben. Dazu zählen Profildaten wie Name, Email-Adresse und Organisationseinheit, aber auch Anmeldedaten und ausgeliehene Medien.
Das letzte Verwaltungsgebiet des Bibliothekssystems sind die Bewegungsdaten. Darunter fallen die konkreten Bestandsdaten zu den sich im Umlauf befindenden Exemplaren eines Mediums.
In der Bibliothek des Forschungszentrums kommt das Integrated Library System Symphony der Firma SirsiDynix zum Einsatz [Sir17]. Es verwaltet die erwähnten Aufgabenbereiche in einer relationalen Oracle Datenbank und arbeitet nach dem 3-Schichten

System.

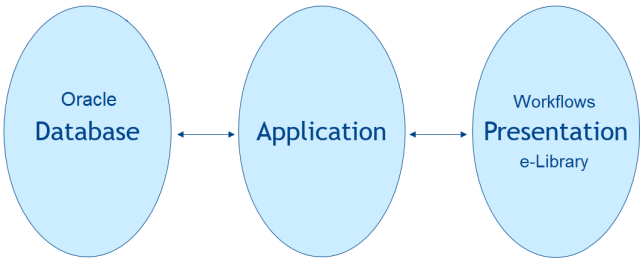


Abbildung 3.1: Aufbau des ILS Symphony

Der Zugriff kann über mehrere Präsentationsprogramme geschehen. Für das Personal der Bibliothek steht eine Client-Software mit dem Namen „Workflows“ zur Verfügung. Mithilfe dieser Software können unter anderem neue Accounts angelegt und Ausleihen verbucht werden.
Im Normalfall würde der Bibliotheksnutzer die für ihn freigeschalteten Funktionen über Symphonys Front-End „e-Library“ erreichen. Die Zentralbibliothek hat dieses Front-End allerdings durch ihre eigene Lösung ersetzt und muss deshalb auf Symphony zugreifen können.
Um Symphony anzusprechen und Transaktionen durchzuführen, kann der in Symphony integrierte API-Server genutzt werden. Er antwortet auf Anfragen, die einem speziellen Muster entsprechen müssen. Für lesende Anfragen existieren beispielsweise die select-Tools, die über eine Eingabe im Terminal angesprochen werden können. Es folgt nun ein Beispiel:

```
selitem -f">20160930<20161101" -h0
```

Mithilfe dieser Anfrage können alle Items aus dem Bibliothekskatalog extrahiert werden, die im Oktober 2016 angelegt wurden und die keine Vormerkungen haben.

Symphony wird auf einem separaten Linux-Server in der Bibliothek gehostet.

## 3.2 Solr

Apache Solr [Apa17a] ist eine als Suchanwendung fungierende Software, die im Bibliothekskatalog der Zentralbibliothek Recherchen durchführen kann. In diesem Katalog sind die Meta-Daten der Medien aus der Zentralbibliothek enthalten. Dazu gehören Bücher, Reports, Dissertationen, eBooks und Zeitschriftenartikel. Die Meta-Daten aus dem Bibliothekssystem sowie die aus den Inhaltsverzeichnissen und Dokumenten extrahierten Texte, werden regelmäßig in den Solr-Server mithilfe der Software SolrMarc [Sol17] eingespeist und im Zuge dessen für die Suche indiziert. Solr baut auf der Open Source Software Lucene [Luc17] auf. Dabei handelt es sich um eine hoch performante und stark skalierbare Informationsabfrage-Bibliothek. Mit ihrer Hilfe kann die Suche von gesamten Dokumenten, Informationen in Dokumenten und Meta-Daten über Dokumente durchgeführt werden [MM10]. Solr bietet eine eigene API zur Kommunikation an [Apa17b].

## 3.3 JuLib eXtended

JuLib eXtended [FJ17b] ist das Discovery Tool der Bibliothek des Forschungszentrums Jülich. Es bietet für die Nutzer die Möglichkeit, nach gedruckten und elektronischen Büchern und Zeitschriften, sowie nach Zeitschriftenartikeln zu suchen. Zu diesem Zweck kann JuLib den Solr-Server ansprechen, der die indizierten Daten aus dem Bibliothekskatalog und anderen Quellen enthält.
Neben der Suchfunktion ermöglicht JuLib die personalisierte Kontenverwaltung. Jeder Nutzer der Zentralbibliothek besitzt ein Konto, das er unter JuLib einsehen kann. Darin können unter anderem die ausgeliehenen Medien betrachtet und bei Bedarf verlängert werden.

Die Homepage wird mit Hilfe einer angepassten Front-End Version des OPACs (Online Public Access Catalog) VuFind [Lib17] ausgeliefert.

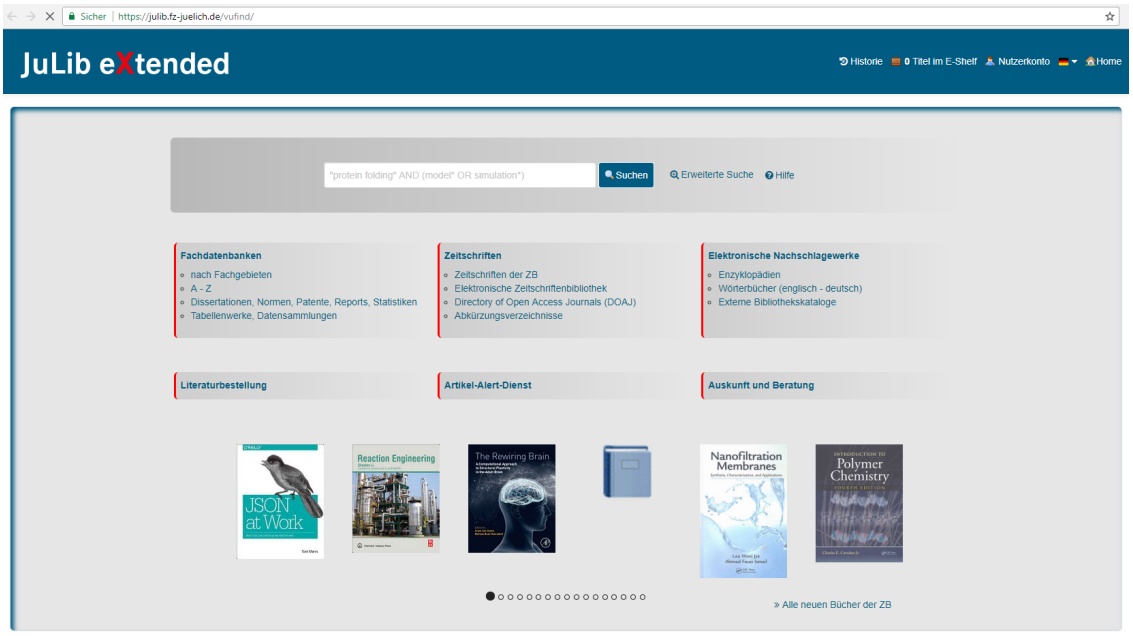


Abbildung 3.2: Front-End von JuLib eXtended

VuFind ersetzt somit das in Symphony integrierte OPAC „e-Library“. Dadurch entsteht ein Mehrwert, da VuFind nicht nur eine bessere Oberfläche, sondern ebenfalls weitere Funktionalitäten bietet. Unter diese bereitgestellten Funktionen fällt die Überprüfung der Verfügbarkeit, sowie die Speicherung von Favoriten, Favoriten-Listen und der Suchhistorie. Zusätzlich können Medien durch den Nutzer verlängert und vorgemerkt werden.

Damit VuFind die personalisierten Funktionen des Bibliothekskatalogs anbieten kann, muss eine Kommunikation mit Symphony stattfinden. Dazu existiert ein VuFind-ILS-Treiber, der Funktionalitäten eines austauschbaren Integrated Library Systems auf ein Interface abbildet [VuF17a]. Ein zweiter Treiber, der ebenfalls durch VuFind zur Verfügung gestellt wird, kümmert sich um die konkrete Ansprache von Symphony und arbeitet direkt auf dem Symphony Server [VuF17b]. Dabei handelt es sich um ein PERL Skript, das die wichtigsten Funktionen und Transaktionen des API-Servers von Symphony als eigene Methoden kapselt. Es kann über parametrisierte HTTP-GET Anfragen angesprochen werden. In der Implementierung des VuFind Interfaces auf der VuFind-Seite wird für jede unterstützte ILS Methode die entsprechende Methode des PERL Skriptes über HTTP aufgerufen.

Es ergibt sich folgender Kommunikationsablauf:

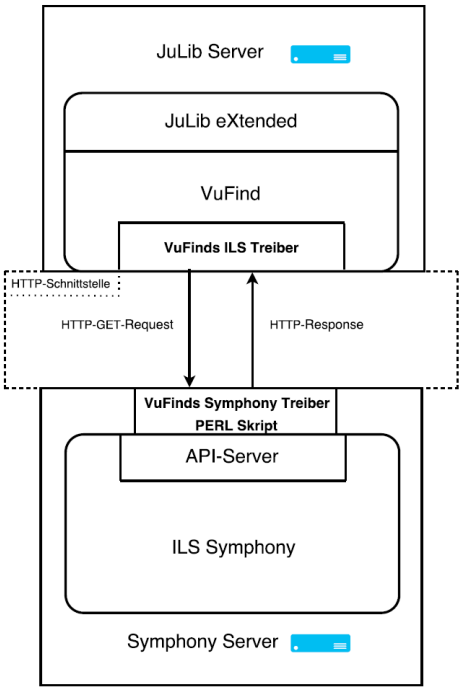


Abbildung 3.3: Kommunikationsablauf zwischen JuLib eXtended und Symphony

Neben dem durch Solr, VuFind und Symphony eingebundenen Funktionsumfang kann über das eigenständige Informationssystem „ZB-Bestellungen (3.5)“ Fachliteratur bestellt werden. Der Zugriff wurde ebenfalls in JuLib implementiert und ist dem Nutzer zugänglich.

Da in JuLib viele benutzerspezifische Funktionen zusammenfließen, bietet JuLib eine eigene Schnittstelle, die ausschließlich die personalisierten Funktionen, wie zum Beispiel Ausleihfristverlängerungen, Favoritenlisten oder Literaturbestellungen, bereitstellt. Sie kann über HTTP-GET Anfragen angesprochen werden und liefert wahlweise HTML oder JSON-Repräsentationen zurück. Eine Suche im Bestand etc. ist somit über die Schnittstelle nicht möglich.

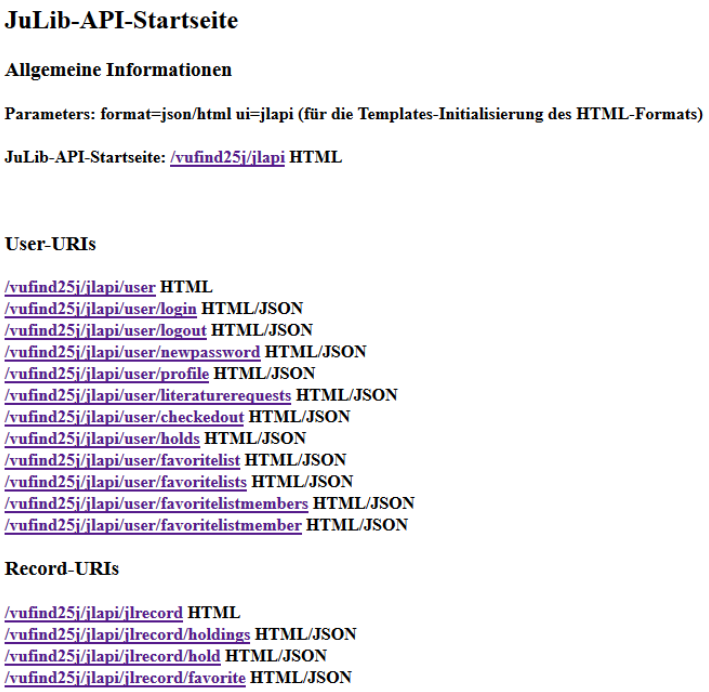


Abbildung 3.4: API von JuLib

Diese Schnittstelle mag auf den ersten Blick wie eine REST-konforme Schnittstelle aussehen, erfüllt jedoch nicht alle Anforderungen.
Die API ist zustandsbehaftet, da sie ein sessionbasiertes Anmeldeverfahren implementiert. Außerdem wird ihre gesamte Kommunikation über HTTP-GET Anfragen abgewickelt, sodass die Spezifikation der HTTP-Methoden nicht eingehalten werden. Darüber hinaus liefert sie Fehlermeldungen nicht über HTTP-Satuscodes, sondern in der Repräsentation, wodurch insbesondere ihre JSON-Repräsentationen sehr verschachtelt sind und sich dadurch nur aufwendig parsen lassen.

## 3.4 EBSCO Discovery Service

Der EBSCO Discovery Service (EDS) bietet Zugriff auf die durch die Bibliothek lizenzierten Zeitschriften. Dabei handelt es sich um einen Service, der extern auf den Servern der Firma EBSCO gehostet wird. Er garantiert modularen Zugriff auf wissenschaftliche Zeitschriftenartikel. Die Ansprache erfolgt über eine eigene REST API [EBS16].

## 3.5 ZB-Bestellungen

„ZB-Bestellungen“ ist ein eigenständiges Informationssystem. Über den sogenannten elektronischen Bestellschein, der mittlerweile in JuLib integriert ist, können die Nutzer ihre Literaturbestellungen aufgeben. Diese werden dann in einer Microsoft SQL Datenbank gespeichert. Die Bearbeitung der Aufträge in der Datenbank kann über eine Desktop-Anwendung durch Mitarbeiter der Zentralbibliothek erledigt werden. Auf diese Datenbank kann man über einen ODBC (Open Database Connectivity) [All17] Treiber aus externen Anwendungen zugreifen.

## 3.6 JuSER

JuSER ist das zentrale Nachweisinstrument der Ergebnisse Jülicher Forschung. In dem System werden die Publikationen der Wissenschaftler eingetragen und Open Access Dokumente im Volltext abgelegt.
Mit JuSER ist es außerdem möglich nach Publikationen zu suchen. Dabei können Sammlungen wie die gesamte Publikationsdatenbank, die Open Access Publikationen, oder die jeweiligen Institutssammlungen durchsucht und Suchfilter angewandt werden.
Als Basis für JuSER dient das vom CERN entwickelte Invenio Digital Library Framework [CER].
Die Eintragungen in JuSER werden regelmäßig vom Solr-Server indiziert, sodass über die Solr-API lesend auf JuSER-Inhalte zugegriffen werden kann. Eine Schnittstelle zur Eintragung von Publikationen in JuSER wird durch eine JuSER Homepage angeboten [FJ17a].

## 3.7 SFX

Ein Open URL Dynamic Link Resolver ist ein Informationssystem, das je nach Lizenzstatus eines Mediums einen Link zum Volltext bereitstellt oder auf ein kontextsensitives Menü verweist, das weitere Instruktionen zur Beschaffung des Volltextes enthält. In der Zentralbibliothek wird als Link-Resolver die Software SFX von der Firma Ex Libris eingesetzt. Der Zugriff auf das System erfolgt über den sogenannten SFX-Button. Dieser ist in vielen internen (JuLib, SFX) sowie externen Webanwendungen (z.B. Web of Science, EBSCOhost, AGRIS, IEEE Xplore und Reaxys) eingebunden. Mittlerweile ist noch nicht geklärt, ob SFX eine strukturierte API anbietet.

## 3.8 Weitere Informationssysteme

Neben den bereits beschriebenen Systemen existieren in der Zentralbibliothek noch weitere Dienste. Zu erwähnen ist das Dokumenten-Management-System (DMS) WIN-DREAM, mit dem die händische Literaturverwaltung vorgenommen wird.
Ein weiteres Tool ist das sogenannte JUelich Inter Library Loan System (JUILL) der Helmholtz-Gemeinschaft Deutscher Forschungszentren. Das Tool ermöglicht den beteiligten Bibliotheken einen automatisierten Austausch von Dokumenten, die von einer Bibliothek gesucht werden und in einer der Partnerbibliotheken bereits zur Verfügung stehen. Der Zugriff erfolgt entweder über Fernleihe, oder über die Webseiten des Anbieters. Ist keine Lieferung durch die Bibliotheken möglich, wickelt das System automatisch den Kauf der Dokumente beim jeweiligen Verlag ab.
Bei Bedarf können diese Systeme, nach einer Analyse ihrer angebotenen Schnittstellen, in die im Rahmen dieser Arbeit zu entwickelnde API mit aufgenommen werden. Vorerst ist ihre funktionelle Abdeckung nicht geplant.

# Kapitel 4

# Konzeption der Schnittstelle

Die im letzten Kapitel dargestellten Informationssysteme der Zentralbibliothek sollen nun durch eine übergeordnete Schnittstelle zusammengefasst werden.
Im Folgenden wird zu diesem Zweck eine Konzeption erarbeitet, nach der die Implementierung erfolgen soll.

Zu Beginn der Entwicklung eines Konzepts steht die Frage, welche Art von Web-Service als Grundlage für die Schnittstelle dienen soll.
Wie bereits unter 2.2, 2.3 und 2.4 analysiert, kommen zwei Varianten der Web-Service Entwicklung in Betracht. Zum einen existiert die Herangehensweise über den Architekturstil REST. Die daraus resultierende Web-API sollte sich dadurch auszeichnen, dass sie die explizite Verwendung der HTTP-Methoden unterstützt, statusbefreit ist, eine ordnerähnliche URI-Struktur aufweist, sowie die Kommunikation über serialisierte Formate, wie beispielsweise JSON und XML, anbietet [Rod15]. Zum anderen wurde die Möglichkeit eines SOAP basierten Web-Services beleuchtet, bei der das Hauptaugenmerk auf dem Aufruf komplexer Methoden auf der Servermaschine liegt.

Die in 1.2 definierten Anforderungen passen sehr gut zu dem REST-Ansatz. Zudem benötigt REST, wie unter 2.4 gezeigt wurde, in vielen Fällen weniger Bandbreite, da die Informationen kompakter übertragen werden können. Gerade für den mobilen Einsatz der Schnittstelle ist dieser Punkt von großer Bedeutung. Aus den genannten Gründen findet die Entwicklung der Schnittstelle nach den REST-Paradigmen statt.

Im Folgenden werden das Interface der Schnittstelle und die Funktionen festgelegt. Anschließend erfolgt die Implementierung.

## 4.1 Ressourcen

Das Ressourcendesign ist für die Schnittstelle von großer Bedeutung. Es fördert die intuitive Nutzbarkeit und den einheitlichen Zugriff.
Für die Dienste der Zentralbibliothek kristallisieren sich drei grundlegende Ressourcen heraus. Die Verwaltung der Nutzerdaten und weiterer nutzerorientierter Funktionen liefert die Hauptressource <user>. Ebenso wichtig ist die <search> Ressource, da für eine Bibliothek die Suche auf verschiedensten Datenbanken von großer Bedeutung ist. Als letzte Hauptressource kommt <record> zum Einsatz. Darunter sind konkrete Bücher über ihre IDs identifizierbar und die zugehörigen Bestandsdaten abrufbar.

Von den Hauptressourcen verzweigen weitere Subressourcen. Nach diesem Konzept ergibt sich eine ordnerähnliche Struktur, die der Anforderung einer REST-API entspricht.

Die konkrete User-Ressource liefert die Daten, die die Bibliothek über den konkreten angemeldeten Nutzer gespeichert hat. Dazu zählen unter anderem Name, Institut, Telefonnummer und Email-Adresse.

Eine Ebene tiefer kommen dann die Subressourcen zum Vorschein. Darunter fällt die Ressource <credentials> für das Ändern von Anmeldedaten, <favoritelists> für die weitere Behandlung von Favoritenlisten, <literaturerequests> zur Einsichtnahme in die eigenen Literaturbestellungen, <checkouts> zur Betrachtung und Verlängerung von Ausleihen und <holds> für die Handhabung von Vormerkungen.
Es ergibt sich also folgende Struktur:


Abbildung 4.1: Ressourcendesign für <user>

Für die Nutzerdaten (user), Anmeldedaten (credentials), Vormerkungen (holds) und Ausleihen (checkouts) muss das Bibliothekssystem Symphony angesprochen werden. Die Bereitstellung von Literaturbestellungen (literaturerequests) erfolgt über das Informati-

onssystem ZB-Bestellungen. Um die Funktionalität der Favoritenlisten zu gewährleisten, ist der Anschluss an VuFind nötig.

Während die anderen Ressourcen nicht tiefer verzweigen, kann <favoritelists> weiter unterteilt werden. Jede Favoritenliste besteht aus Favoriten. Deswegen bietet <favoritelists> die Subressource <favorites> zu einer bestimmten Favoritenliste an. Favoriten einer Favoritenliste bestehen aus Records, die über den untenstehenden Pfad erreichbar sind.


Abbildung 4.2: Ressourcendesign für <favoritelists>

Die Ressource <search> bietet verschiedene Unterressourcen, die teilweise variabel sind:

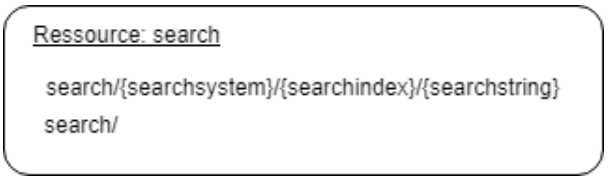


Abbildung 4.3: Ressourcendesign für <search>

< {searchsystem} > spezifiziert dabei die zu durchsuchende Datenbank. Durch die Ersetzung von < {searchsystem} > durch <bilbio>, <juser> oder <eds> kann entweder der Bibliothekskatalog, die Publikationsdatenbank oder der EBSCO Discovery Service ausgesucht werden. < {searchindex} > konkretisiert, nach welchen Daten die Recherche durchgeführt wird. Es kann durch <allfields>, um die Suche nicht einzuschränken, oder durch beispielsweise <title> oder <author> ersetzt werden, um entweder nach konkreten Titeln oder Autoren zu suchen.
Der < {searchstring} > legt abschließend das Gesuchte fest. In dem durch <searchindex> gegebenenfalls spezifizierten Teilgebiet wird dann nach der in <searchstring> enthaltenen Zeichenkette gesucht.
Die Recherche im Bibliothekskatalog und JuSER erfolgt durch Solr. Die Ansprache von

EDS geschieht über die verfügbare REST-API.

Die letzte Hauptressource ist <record>.


Abbildung 4.4: Ressourcendesign für <record>

Mithilfe der Subressource < {recordid} > lassen sich konkrete Datenbankeinträge finden und die dazugehörigen Daten auslesen. Zu diesem Datenbankeintrag kann durch die tiefer gestufte <holdings>-Ressource der konkrete Bestand in der Zentralbibliothek ermittelt werden. An dieser Stelle muss wiederum das Bibliothekssystem angesprochen werden.

## 4.2 HTTP-Methoden

Auf den bereits definierten Ressourcen der Web-API müssen die jeweils sinnvollen HTTP-Methoden aufgerufen werden können. Dazu ergeben sich bestimmte Good-Practice-Regeln, die auch zu den unter 2.1.1 beschriebenen Spezifikationen der HTTP-Methoden konform sind.
Ein GET wird demnach nur für lesende Zugriffe benutzt. Ressourcen, die im Plural formuliert sind, so wie etwa <favoritelists>, geben bei einem GET-Aufruf eine Ergebnisliste zurück.
Eine Plural-Ressource erlaubt die Angabe einer Identifikations-ID als Subressource, um ein konkretes Objekt der Liste auswählen zu können.
Für das konkrete Ändern und Erzeugen von Ressourcen, die keine Listenressourcen sind, wird PUT benutzt. Eine weitere Voraussetzung ist, dass das mehrmalige Ausführen der PUT-Methode, im Vergleich zur einmaligen, keine zusätzlichen Auswirkungen hat. Dadurch wird die idempotente Eigenschaft sichergestellt.
In allen anderen Fällen, in denen Ressourcen erzeugt oder geändert werden, aber keine Garantien gegeben werden können, findet die Verwendung von POST statt. Vor allem findet POST Gebrauch, wenn ein Objekt zu einer Liste hinzugefügt werden soll und die ID-Zuweisung automatisch durch die API stattfindet.
Die DELETE-Methode wird bei dieser Schnittstelle nur für das Löschen von ID-Ressourcen

eingesetzt. Es folgt eine Tabelle, in der die jeweiligen unterstützten Methoden zu den verschiedenen Ressourcen gelistet sind:

| HTTP-Methoden: | **GET** | **POST** | **PUT** | **DELETE** |
|---|---|---|---|---|
| user | ✓ | | | |
| user/credentials | | | ✓ | |
| user/favoritelists | ✓ | ✓ | | |
| user/favoritelists/{listid} | ✓ | | ✓ | ✓ |
| user/favoritelists/{listid}/favorites | ✓ | | | |
| user/favoritelists/{listid}/favorites/{recordid} | | | ✓ | ✓ |
| user/holds | ✓ | | | |
| user/holds/{itemid} | | | ✓ | ✓ |
| user/literaturerequests | ✓ | | | |
| user/checkouts | ✓ | ✓ | | |
| search/{searchsystem}/{searchsubject}/{searchstring} | ✓ | | | |
| record/{recordid} | ✓ | | | |
| record/{recordid}/holdings | ✓ | | | |

Tabelle 4.1: Mapping der HTTP-Methoden auf die Ressourcen der Web-API

## 4.3 Repräsentationen

Als unterstützte Repräsentationsformate für die Web-API der Zentralbibliothek dienen XML und JSON. Die gängigen Programmiersprachen bieten eine gute Unterstützung dieser Formate an, sodass durch ihre Verwendung die Programmiersprachenunabhängigkeit der REST-API erreicht werden kann. Das Repräsentationsformat muss außerdem auf Basis des Accept-Headers dynamisch zwischen Server und Client ausgehandelt werden können.
Folgende Repräsentationen werden dann beispielsweise von der <user> Ressource nach einer erfolgreichen GET-Anfrage zurückgeliefert.

JSON-Repräsentation von <user>

```
{
      ”firstname”: ”Jannis (Hr.)”,
      ”lastname”: ”Konopka”,
      “address1”: “ZB”,
      ”phone”: ”1587”,
      “email”: ”j.konopka@fz-juelich.de”,
      ”group”: ”FZJ”,
      ”library”: “ZB”,
      “username”: ”320806”
}
```

Tabelle 4.2: JSON Repräsentation

XML-Repräsentation von <user>

```
<UserProfile
      xmlns:i=”http://www.w3.org/2001/XMLSchema-instance”>
      <firstname> Jannis (Hr.)</firstname>
      <lastname>Konopka</lastname>
      <address1>ZB</address1>
      <phone>1587</phone>
      <email>j.konopka@fz-juelich.de</email>
      <group>FZJ</group>
      <library>ZB</library>
      <username>320806</username>
</UserProfile>
```

Tabelle 4.3: XML Repräsentation

Um die HATEOAS-Aspekte in der Schnittstelle umzusetzen, wird nach der unter 2.3.3 beschriebenen PayPal-Konzeption vorgegangen. Dadurch erfolgt eine Bereitstellung von Verlinkungen, die pro Verlinkung jeweils URI, Relation und HTTP-Methode enthält.

Das Beispiel für die Ressource <user> sieht wie folgt aus:

```
"Links": [
    {
      "Href": "user",
      "Rel": "self",
      "Method": "GET"
    },
    {
      "Href": "user/credentials",
      "Rel": "updateCredentials",
      "Method": "PUT"
    },
    {
      "Href": "user/favoritelists",
      "Rel": "getFavoriteLists",
      "Method": "GET"
    },
    {
      "Href": "user/checkouts",
      "Rel": "getCheckouts",
      "Method": "GET"
    },
    {
      "Href": "user/literaturerequests",
      "Rel": "getLiteratureRequests",
      "Method": "GET"
    },
    {
      "Href": "user/holds",
      "Rel": "getHolds",
      "Method": "GET"
    }
  ]
```

## 4.4 Statuscodes

Auftretenden Fehlern bei der Nutzung der Schnittstelle durch Clients wird mit verschiedenen Fehlermeldungen begegnet. Erfolgreiche Anfragen werden ebenfalls als solche kommuniziert. Zu diesem Zweck kommen die HTTP-Statuscodes zum Einsatz.
Immer wenn eine Anfrage erfolgreich verarbeitet und ein passendes Ergebnis erzeugt werden konnte, bekommt der Client den Statuscode 200 übermittelt. Wurde mithilfe eines POSTs eine neue Ressource kreiert, erhält der Client über den Code 302 die Mitteilung, dass sein neuer Eintrag gefunden und somit korrekt erzeugt wurde. Über den in der Server-Antwort enthaltenen Location-Header kann der Client zu der neu generierten Ressource weitergeleitet werden (redirect).
Ist eine angeforderte Ressource nicht vorhanden, liefert die API den Code 404 - Not Found. Wird im Rahmen eines POSTs oder PUTs ein fehlerhafter Body übermittelt, quittiert dies der Code 400 - Bad Request.

## 4.5 Zugriffsrechte

Weil die angebotenen Services der Zentralbibliothek lizenzrechtlichen Bestimmungen unterliegen, darf die gesamte Verwendung der Schnittstelle nur für Nutzer der Bibliothek mit vorhandenem Nutzeraccount möglich sein. Daher muss eine Authentifizierung des anfragenden Clients geschehen. Um zustandslos zu bleiben, ist es erforderlich, bei jeder Anfrage zu prüfen, ob die zugreifende Clientanwendung zur Nutzung berechtigt ist, oder nicht.
Der Schutz gegen unautorisierten Zugriff auf eine REST-API kann schon durch die Verwendung einer Kombination aus der im HTTP-Protokoll implementierten Basic-Authentification und TLS (Transport Layer Security) erreicht werden [Til15, Kapitel 11.5]. Dabei wird eine Zeichenkette bestehend aus username:password Base64-kodiert im Authorization-Header einer HTTP-Request-Nachricht eingetragen.

```
Beispiel: username=zb, password=api
Base64(zb:api) wird zu emI6YXBpDQo=

Der Authorization-Header sähe wie folgt aus:
Authorization: Basic emI6YXBpDQo=
```

Da diese Verschlüsselung jedoch umkehrbar ist, muss zusätzlich sogenannten Man-In-The-Middle-Angriffen vorgebeugt werden. Bei solch einer Attacke schaltet sich ein potenzieller Angreifer zwischen zwei Kommunikationspartner und kann dadurch Nachrichten mitlesen und gegebenenfalls verändern.
Um dem entgegenzuwirken, kommt das um TLS erweiterte HTTP-Protokoll „HTTPS“ zum Einsatz. Über ein Server-Zertifikat wird dabei die Echtheit der Kommunikationspartner garantiert. Die versandten Request- und Response-Nachrichten sind dadurch auf der Transportebene asynchron verschlüsselt.
Da die Anmeldedaten bei der Verwendung von Basic-Authentification und TLS weder in der Parameterliste übertragen, noch von einem zwischenliegendem Softwaresystem ausgelesen werden können, sind sie gegen die Einsichtnahme Fremder geschützt. Gleiches gilt für sonstige übertragene Daten. Somit können die Lizenzbestimmungen eingehalten werden.
Die an die API übertragenen Anmeldedaten werden zur Authentifizierung eingesetzt. Zu diesem Zweck kann eine Anfrage an die JuLib-API gestellt werden, die wiederum das Symphony-System anspricht. Es wird ein Session-Cookie generiert, der im Rahmen der serverseitigen Verarbeitung der aktuellen Client-Anfrage weiterhin genutzt werden kann. Nach der gesamten Verarbeitung der Anfrage wird der Cookie gelöscht und die Anmeldung erlischt.

## 4.6 Versionierung und Erweiterbarkeit

Um die in den Anforderungen gelistete Abwärtskompatibilität zu gewährleisten, wird ein Versionierungssystem eingeführt. Dazu kann eine Versionsressource vor die bekannten Hauptressourcen geschaltet werden. Als generelle Heimatressource dient die <api>-Ressource. Hinter ihr folgt die Version, die bei <v1> beginnt und nach gravierenden Änderungen an der API hochgezählt werden kann. Auch bei neueren Versionen bliebe der einmal entwickelte Client zur Vorgängerversion funktionstüchtig.
Es ergibt sich folgender Ressourcenaufbau.

```
api/{version}/{user/search/record}/...
```

Sollte das bestehende Angebot an Ressourcen erweitert werden, ist das kein Problem. Es lassen sich einfach neue Ressourcen generieren, sogar ohne die Versionsnummern ändern zu müssen.

Die Erweiterbarkeit und Abwärtskompatibilität ist also durch diesen beschriebenen Ansatz gewährleistet.

## 4.7 Datenminimierung

Bei manchen Anfragen kann es zu erhöhter Datenübertragung kommen. Das ist insbesondere dann der Fall, wenn eine Durchsuchung von Datenbanken nach allgemeinen Begriffen statt findet. Um auch bei solchen Anfragen nur eine möglichst geringe Datenmenge zu übertragen, kann ein Konzept zur Paginierung eingesetzt werden, das die Ergebnismenge seitenweise limitiert.
Der Client soll die Möglichkeit haben, selbst zu bestimmen, wie viele Ergebnisse pro Seite angezeigt werden sollen. Das Link-Element der Antwort enthält dann die verfügbaren URIs, um zur nächsten oder vorherigen Seite zu kommen. Die Angabe der Ergebnisanzahl pro Seite und der aktuellen Seitennummer erfolgt über die Parameterliste. Sie ist für die Ergebnisfilterung optimal einsetzbar.

Ein Beispiel könnte dann wie folgt aussehen:

```
search/biblio/allfields/python&perpage=5?page=1
```

Dies soll auch gleichzeitig der Standard sein, falls keine Angaben zur Einschränkung der Ergebnismenge vorhanden sind, um nicht unnötig große Datenmengen zu übertragen. Wie zu sehen ist, enthält das Links-Array dann einen Eintrag, um zur nächsten Seite zu kommen. Dadurch kann der Client selbstständig durch die Ergebnisse navigieren.

```
"Links": [
    ...
    {
      "Href": "search/biblio/allfields/python&perpage=5?page=2",
      "Rel": "next",
      "Method": "GET"
    }
    ...
]
```

Ab einer Seite größer als 1 wird zusätzlich zur "*next*" Relation ebenfalls eine "*previous*" Relation zur Verfügung gestellt, um die vorherige Seite erreichbar zu machen.

Durch die Etablierung dieses Paginierungssystems, kann die übertragene Datenmenge effektiv reduziert werden. Davon profitieren vor allem mobile Endgeräte.

## 4.8 Dokumentation

Die Dokumentation eines Application Interfaces ist von großer Bedeutung, damit der Service von anderen Entwicklern in Anwendungen integriert werden kann. Gerade bei der Dokumentation ist es wichtig, auf automatische Erzeugung zu achten, da kleine Fehler zu falschen Anfragen führen würden, die nicht unbedingt mit geeigneten Fehlermeldungen quittiert würden.
Deshalb wird für die Dokumentation dieser Schnittstelle das Tool Swagger [Sof17] eingesetzt. Swagger ist ein populäres Werkzeug zum Beschreiben und Dokumentieren von REST-APIs. Die ursprüngliche Swagger-Specification heißt seit 2015 OpenAPI-Specification und wird von der OpenAPI Initiative unter der Linux Foundation weiterentwickelt [Ini17]. Ende Juli 2017 wurde Version 3.0 veröffentlicht, die auf der 2.0 Spezifikation aufbaut. Das Ziel der OpenAPI Initiative ist die möglichst genaue Abbildung eines REST-Services in einer maschinen -und menschenlesbaren JSON- bzw. YAML-Datei. Weil die Initiative von mehreren großen Unternehmen gesponsert wird, ist abzusehen, dass das Beschreiben von REST-APIs nach und nach zu einem Industriestandard vereinheitlicht wird [Sma17].
Da die OpenAPI-Spezifikation programmiersprachenunabhängig ist und eine deklarative Ressourcenbeschreibung bietet, können Clients, ohne die Implementierung des Servers kennen zu müssen, den Service verstehen und konsumieren. Genau bedeutet das, dass die Spezifikation zu jedem URI-Muster die unterstützten HTTP-Methoden, eine Beschreibung und die Parameter festlegt. Zusätzlich können die verwendeten Medientypen, Statuscodes und JSON Schemata für die Antworten angegeben werden [Til15, Kapitel 12.4.3].
Es existieren für viele Sprachen Swagger-Implementierungen zur Umsetzung der entwickelten Schnittstelle in die einheitliche Beschreibungsdatei. Swagger stellt weitere Werkzeuge wie das Swagger UI (User Interface), den Swagger Editor und einen Codegenerator zur Verfügung [Swa17]. Das User Interface bereitet zu einer REST-API eine HTML Client-Anwendung auf und liefert somit eine automatisch generierte interakti-

ve Dokumentationsquelle. Mithilfe des Swagger-Editors können spezifikationskonforme YAML-Datei geschrieben werden. Diese Möglichkeit verfolgt den Contract-First Ansatz. Der Codegenerator kann wiederum zu einer spezifikationskonformen JSON- oder YAML-Datei automatisch Software Development Kits (SDKs) für Clients und sogenannte Serverstubs (Implementationslose Serverrümpfe mit vorgefertigten Schnittstellen) generieren. Über diese Herangehensweise kann nach dem Top-Down Prinzip Servercode für REST-APIs automatisch generiert werden.

# Kapitel 5

# Implementierung der Schnittstelle

Für die Implementierung können viele Techniken verwendet werden. Aktuelle Standards sind beispielsweise das auf PHP basierende Slim- [JL17], oder das auf Ruby basierende Sinatra-Framework [Sin17]. Eine andere Möglichkeit besteht in der Nutzung des ASP.NET Web API 2 Moduls, mit dem sich direkt in Microsofts integrierter Entwicklungsumgebung Visual Studio Web-APIs entwickeln lassen. Ein weiterer Microsoft-Weg führt über das Windows Communication Framework (WCF).
Im Folgenden wird das Hauptaugenmerk auf ASP.NET Web API 2 gelegt, da es besser auf die Implementierung von REST-APIs ausgelegt ist.

Zur Entwicklung der Schnittstelle wird Visual Studio 2015 Professional und die ASP.NET Version 4.0.30319 benutzt. Die Programmiersprache ist C#. Die Entwicklungsumgebung ist ein 64-Bit Windows 8.1 Pro Rechner mit acht Gigabyte RAM und vier 3.2 GHz Intel Prozessorkernen.

Der programmatische Aufbau entspricht dem Model-View-Controller (MVC) Konzept.

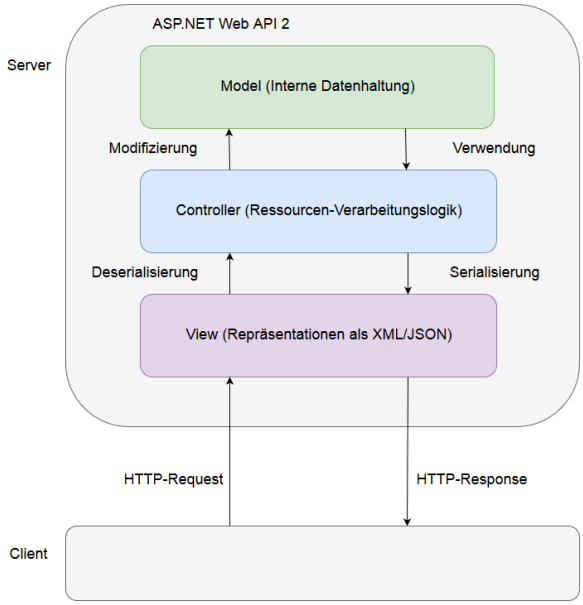


Abbildung 5.1: MVC-Model der API

Die Controller enthalten die einzelnen HTTP-Methoden mit ihrer jeweiligen Implementierung. Sie verwenden die im Model definierten serialisierbaren Datenstrukturen. Der Controller liefert indirekt die Instanz einer Model-Klasse an den Client. Das ASP.NET Modul kümmert sich um die Umsetzung der Instanz in eine XML- oder JSON-Repräsentation. Die Auswahl wird gemäß Content Negotiation auf Basis des Accept-Headers der Request-Nachricht getroffen. Es findet also keine explizite Programmierung der View statt.

## 5.1 Controller

Für die Schnittstelle der Zentralbibliothek gibt es neun Controller. Jeder einzelne deckt sein semantisches Aufgabengebiet ab.
Es ergibt sich die folgende Zuordnung:

| Controller | Funktion |
|---|---|
| Checkouts | Ausleihen betrachten und verlängern |
| Credentials | Zugangsdaten ändern |
| FavoriteLists | Favoritenlisten betrachten, anlegen, ändern und löschen; Favoriten betrachten, hinzufügen löschen und ändern |
| Holds | Vormerkungen betrachten, anlegen und löschen |
| LiteratureRequests | Literaturbestellungen betrachten |
| Record | Ein bestimmtes Dokument betrachten, konkreten Bestand in der Zentralbibliothek betrachten |
| Search | Verschiedene Suchergebnisse von Anfragen an Bibliothekssystem, JuLib und EDS betrachten |
| Start | Einstiegspunkt bieten |
| UserProfile | Nutzerprofil betrachten |

Tabelle 5.1: Die verschiedenen Controller der Web-API

Jeder Controller ist von der Klasse ApiController abgeleitet und implementiert die verschiedenen HTTP-Methoden als eigene Methoden, im Folgenden "Aktionen" genannt. Dabei ist die Namensgebung sehr wichtig. Beginnt eine Aktion mit "Get" wird damit die Zuweisung der HTTP-GET Methode signalisiert. Die weiteren Aktionen können analog generiert werden.
Wenn die Web-API eine Anfrage empfängt, reicht sie diese an die entsprechende Aktion weiter. Dabei wird nach dem Konzept der Routing-Tabelle verfahren, wobei eine Route einer Kombination aus URI und HTTP-Methode entspricht. Jede Controller-Aktion ist mit einem Eintrag aus dieser Tabelle verknüpft. Die Eintragung einer Aktion kann entweder automatisch über eine Verknüpfung aus der Default-Route und dem Controllernamen, oder explizit durch Angabe eines Route-Attributs erfolgen.
Im folgenden Beispiel wird die Eintragung explizit vorgenommen:

```
[Route("api/v1/user/credentials", Name = "ChangeCredentials")]
public IHttpActionResult PutCredentials(NewCredentials
    newCredentials){...}
```

Listing 5.1: Zu sehen ist die Aktion PutCredentials aus dem CredentialsController. Sie verwendet explizites Routing unter Angabe des Route-Attributs.

Über den Rückgabetyp *IHttpActionResult* können verschiedene Statuscodes als Antwort generiert werden. Dazu helfen die Methoden *Ok*() (Code 200) und *BadRequest*("It failed to change your credentials!") (Code 400). Die gezeigte PutCredentials-Aktion erhält zur weiteren Bearbeitung eine Instanz der Klasse *NewCredentials*. Diese Model-Klassen werden unter 5.2 genauer behandelt.

## 5.2 Model

Die Klassen des Models repräsentieren die Ressourcendaten. Um die Serialisierung verfügbar zu machen, enthält jede Klasse des Models das *DataContract* Attribut. Jede serialisierbare Eigenschaft dieser Klasse ist durch das *DataMember* Attribut gekennzeichnet.
Für die Klasse *NewCredentials* ergibt sich die folgende Strukturierung:

```
[DataContract]
public class NewCredentials
{
    [DataMember(Name = "username")]
    public string Username { get; set; }

    [DataMember(Name = "old_password")]
    public string OldPassword { get; set; }

    [DataMember(Name = "new_password")]
    public string NewPassword { get; set; }
}
```

Listing 5.2: Zu sehen ist die Klasse NewCredentials. Sie kommt beim Ändern der Zugangsdaten zum Einsatz.

Da die beschriebene Klasse den Übergabetyp der PutCredentials-Aktion bildet, hat der Nutzer die Möglichkeit, seine Anmeldedaten auf zwei Arten zu ändern. Im Body der HTTP-Request-Nachricht kann er entweder eine JSON- oder XML-Variante nach den folgenden Schemata übermitteln. Die Web-API konvertiert den Body dann automatisch in eine Instanz der *NewCredentials* Methode.

```
{
  "username": "string",
  "old_password": "string",
  "new_password": "string"
}
```

```
<?xml version="1.0"?>
<NewCredentials>
  <username>string</username>
  <old_password>string</old_password>
  <new_password>string</new_password>
</NewCredentials>
```

Während *NewCredentials* für eingehende Nachrichten verwendet wird, gibt es andere Klassen, die für die Beschreibung von ausgehenden Nachrichten genutzt werden. Zu

diesem Zweck kann der Rückgabewert von Controller-Aktionen beliebige Klassen des Models enthalten. Die Serialisierung nach JSON oder XML findet dann aufgrund des Accept-Headers gemäß Content-Negotiation statt.

## 5.3 Authentifizierung

Für die in der Konzeption geforderten Basic Authentifizierung kommt das BasicAuthJulibHttpModule zum Einsatz. Es implementiert das Interface IHttpModule.
Die Klasse ist in der Konfigurationsdatei der Web-API unter *runForAllRequests* angegeben und wird deshalb vor jeder Anfrage ausgeführt. So kann garantiert werden, dass der aktuelle Nutzer für die Nutzung der Schnittstelle autorisiert ist.

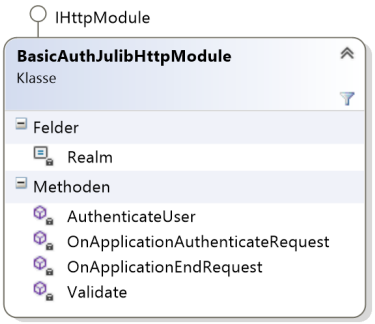


Abbildung 5.2: Authentifizierungsmodul für die Schnittstelle

Zuerst wird in *OnApplicationAuthenticateRequest* überprüft, ob der Authorization-Header in der aktuellen Anfrage vorhanden ist und nach dem richtigen Schema (4.5) befüllt wurde. Anschließend findet in *AuthenticateUser* die Entschlüsselung der Base64-kodierten Zeichenkette, sowie die Trennung der Anmeldedaten am Trennzeichen ” : ” statt. Es folgt die Überprüfung der Anmeldedaten durch die Methode *Validate*. Sie stellt eine Anfrage an die JuLib-API und interpretiert aus der JSON-Antwort, ob die Anmeldung erfolgreich war. Im Fall einer erfolgreichen Anmeldung wird der Session-Cookie aus der JuLib-API in den *Set-Cookie* Header der aktuellen Anfrage geschrieben und ist dadurch in einer nachfolgenden Controller-Aktion abrufbar.
War die Anmeldung nicht erfolgreich, setzt die *OnApplicationEndRequest* Methode den Statuscode 401 *Unauthorized*. Zusätzlich wird der Header *WWW-Authenticate* auf die Zeichenkette Realm *REST-API der Zentralbibliothek Jülich* gesetzt, um der Client-Anwendung ein erneutes Übertragen von Anmeldedaten zu ermöglichen.

## 5.4 Dokumentationstool Swagger

Zur Dokumentation der Schnittstelle wird Swagger in der Version 2.0 eingesetzt. Dank der Implementierung Swashbuckle [Mor17] für das ASP.NET Web-API 2 Modul kann die Dokumentationsautomatisierung auch für die Schnittstelle der Zentralbibliothek eingesetzt werden. Zudem bietet das NuGet-Paket in der Version 5.6.0 ein eigenes Swagger User Interface, das nach Einbindung des Pakets unter [SERVERNAME]/swagger zur Verfügung steht. Darin existiert für jeden Controller des MVC-Models ein Swagger-Knoten, der die implementierten HTTP-Methoden enthält.

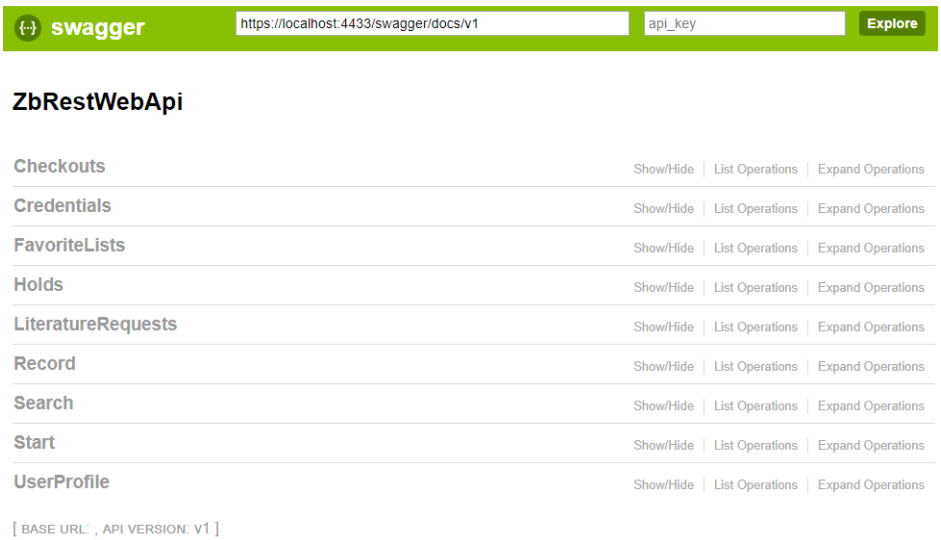


Abbildung 5.3: Die einzelnen Controller in Swagger

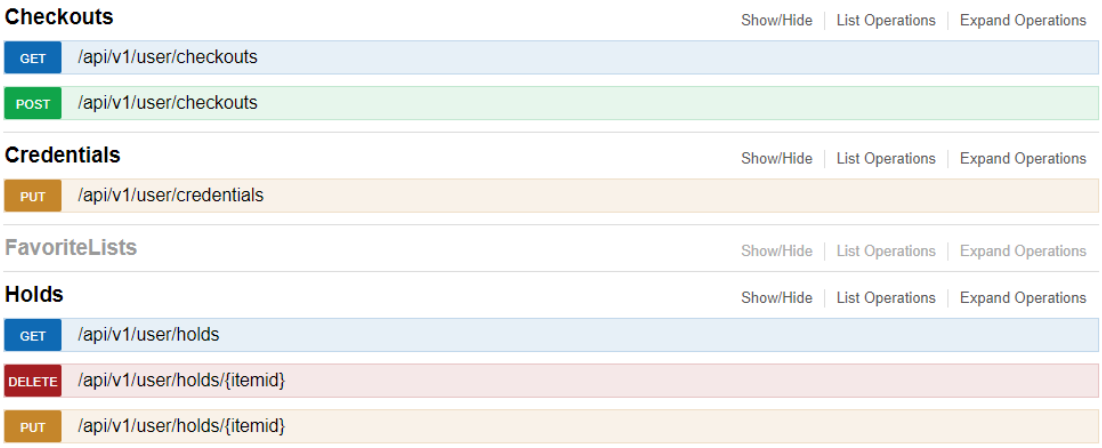


Abbildung 5.4: Swagger Controller für User-Funktionen

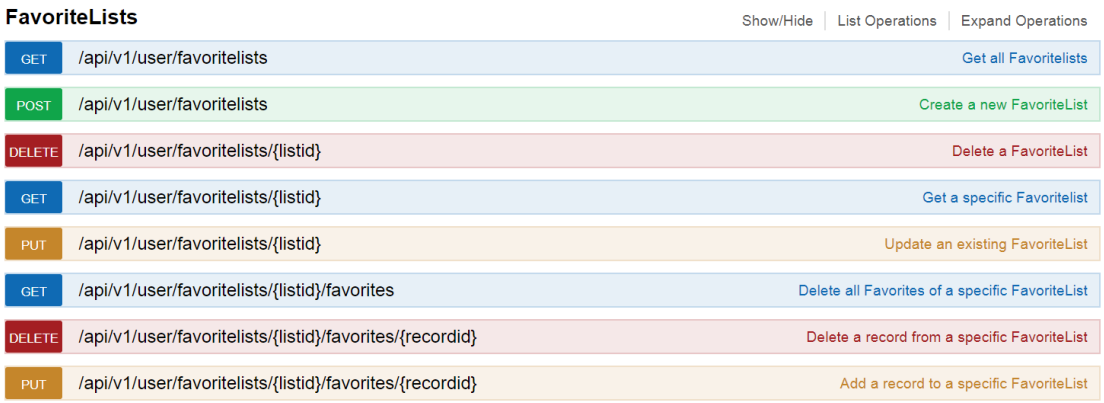


Abbildung 5.5: Swagger Controller für das Verwalten von Favoritenlisten

Um das Verhalten der API zu testen, können die Methoden direkt über das User Interface ausgeführt werden. Es fungiert also als erste Client-Anwendung der Schnittstelle. Zur Nutzung kann bei einem GET-Request zuerst das Repräsentationsformat ausgewählt und anschließend die Anfrage ausgeführt werden. Bei einem PUT oder POST wird zusätzlich eine Beispiel-Form des Bodys der Request-Nachricht bereitgestellt. Bei Bedarf kann diese mit eigenen Werten befüllt und über die „*Try it out!* Schaltfläche“ zum Server geschickt werden. Im Anschluss wird dem Nutzer die Response-Nachricht präsentiert. Anhand des nächsten Beispiels kann die Funktionsweise demonstriert werden.

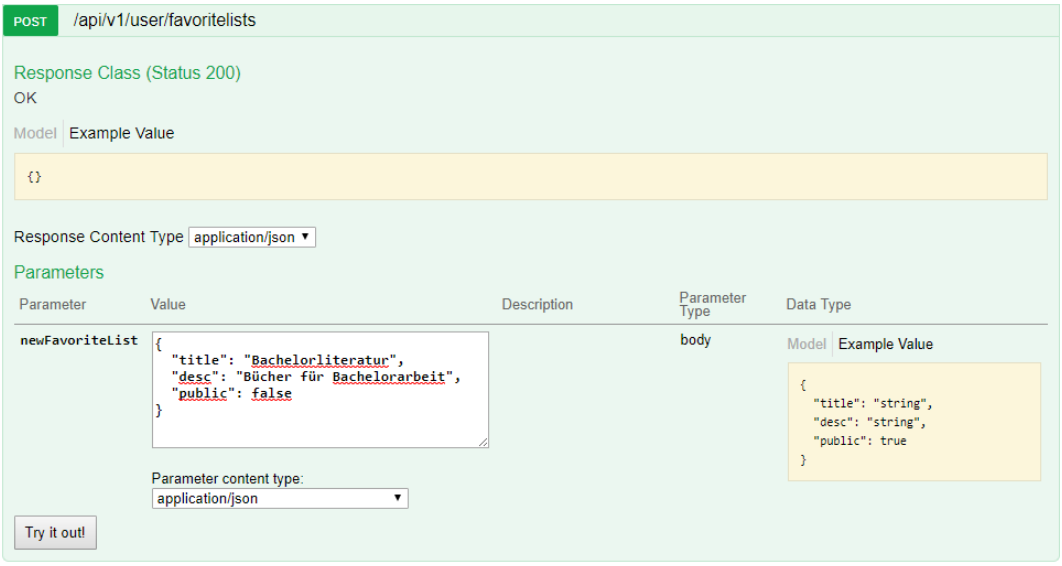


Abbildung 5.6: Swagger-Request zur Erstellung einer neuen Favoritenliste

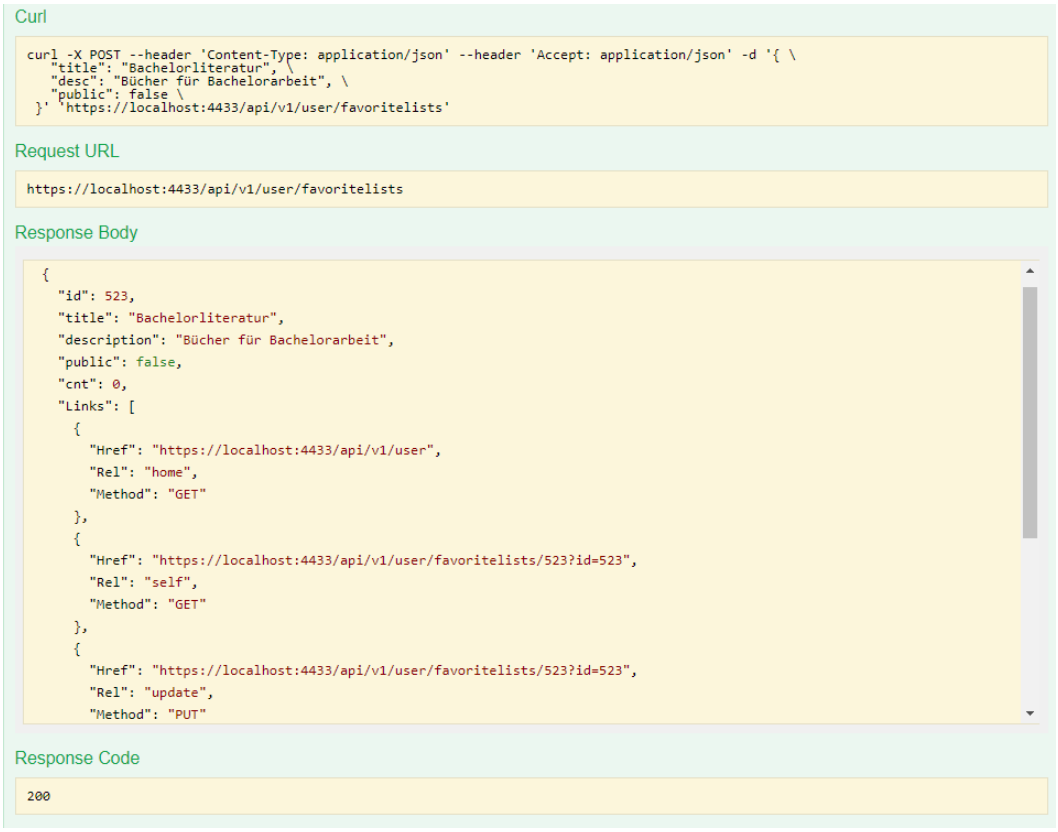


Abbildung 5.7: Swagger-Response auf die Erzeugung einer neuen Favoritenliste

## 5.5 Hosting

Das lokale Hosting der Web-API erfolgt durch Microsofts IIS (Internet Information Service) in der Version 8.5. Darin können neben vielen anderen Einstellungen auch die Authentifizierungsart, der Anwendungsport und das ausschließliche Erlauben des HTTPS Protokolls eingestellt werden.
Da für die lokalen Tests HTTPS und der Port 4433 verwendet werden, lautet der Pfad zur Web-API: https://localhost:4433/api/...
In der Zentralbibliothek befindet sich ein Microsoft Server, der ebenfalls den Internet Information Service anbietet. Auf diesem kann die REST-API nach vollständiger Implementierung produktiv gehostet werden.

## 5.6 Bandbreitentests

Die Schnittstelle wurde unter Berücksichtigung des Einsatzes im mobilen Umfeld entwickelt. Für dieses Einsatzgebiet wird im Folgenden die Funktionsfähigkeit getestet, indem die Bandbreite limitiert und die Latenz erhöht werden. Aus den Ergebnissen kann dann abgeleitet werden, ob das Nutzungserlebnis der Schnittstelle maßgeblich von dem zugrundeliegenden Übertragungsdienst abhängt.

| Profile Name | Download | Upload | Latency |
|---|---|---|---|
| | kb/s | kb/s | ms |
| Add Cancel | optional | optional | optional |
| Offline | 0 kb/s | 0 kb/s | 0ms |
| GPRS | 50 kb/s | 20 kb/s | 500ms |
| Regular 2G | 250 kb/s | 50 kb/s | 300ms |
| Good 2G | 450 kb/s | 150 kb/s | 150ms |
| Regular 3G | 750 kb/s | 250 kb/s | 100ms |
| Good 3G | 1.5 Mb/s | 750 kb/s | 40ms |
| Regular 4G | 4.0 Mb/s | 3.0 Mb/s | 20ms |
| DSL | 2.0 Mb/s | 1.0 Mb/s | 5ms |
| WiFi | 30 Mb/s | 15 Mb/s | 2ms |

Abbildung 5.8: Auswahl verschiedener mobiler Übertragungsdienste, die für die Simulation zur Verfügung stehen

In der Grafik 5.8 sind die standardisierten Verbindungstypen der mobilen Datenkommunikation gelistet. Mit ihnen gehen Downloadgeschwindigkeit, Uploadgeschwindigkeit

und Latenz einher. Dank der Developer-Tools, die in Googles Browser Chrome verfügbar sind, können Anfragen über eine limitierte Bandbreite mit der dem Medium entsprechenden Latenzzeit simuliert werden.
Getestet werden die drei Verbindungstypen Ethernet, Regular 4G und GPRS (General Packet Radio Service). Dazu wird immer der Mittelwert aus 10 Abfragen gebildet.

| | **Datengröße** | **Ethernet** | **Regular 4G** | **GPRS** |
|---|---|---|---|---|
| user | 1.2 KB | 2,74 s | 3,32 s | 3,71 s |
| user/favoritelists | 3,2 KB | 2,44 s | 2,5 s | 3,04 s |
| search/biblio/allfields/python | 1.0 KB | 2,89 s | 2,82 s | 2,91 s |

Tabelle 5.2: Messergebnisse

Es ist zu erkennen, dass die API nur geringe Datenmengen im unteren Kilobyte-Bereich verschickt. Bei der Übertragung über die Mobilfunknetze fällt außerdem die Latenzzeit ins Gewicht.
Der Grafik kann außerdem entnommen werden, dass die einzelnen Informationssysteme verschiedene Antwortzeiten haben. Eine Abfrage der Favoritenlisten ist schneller durchgeführt als die der Nutzerdaten, weil VuFind schneller angesprochen werden kann als das Bibliothekssystem Symphony. Das erfüllt die Erwartungen, denn zwischen API und Bibliothekssystem befindet sich eine Abstraktionsebene mehr als zwischen API und VuFind.

Generell sind die Geschwindigkeiten zwischen den Übertragungstypen nicht gravierend unterschiedlich. Das Nutzungserlebnis hängt demnach nicht maßgeblich von der Wahl des Endgerätes ab.

# Kapitel 6

# Fazit und Ausblick

## 6.1 Zusammenfassung

Ziel dieser Arbeit ist es, einen Überblick über die theoretischen Grundlagen verschiedener Web-Service-Konzepte zu gegeben und deren verschiedene Funktionsweisen zu erklären. Dazu wurden die unterschiedlichen Facetten des Architekturstils REST analysiert und mit dem Netzwerkprotokoll SOAP vor allem hinsichtlich der mobilen Nutzbarkeit verglichen. Auf Basis dieser Diskussion fand eine Auswahl für die Schnittstelle der Zentralbibliothek statt, deren Anforderungen und zugrundeliegenden Systeme ebenfalls beleuchtet wurden. Nach dem Entwurf einer konzeptionellen Annäherung konnte die geforderte Schnittstelle erfolgreich implementiert werden. Die zu Beginn definierten Rahmenbedingungen wurden eingehalten. Insbesondere die Dokumentation konnte durch die Bereitstellung des effizienten Tools Swagger gewährleistet werden. Auf der Grundlage von Bandbreitentests wurde gezeigt, dass die ausgetauschten Datenmengen überschaubar und regulierbar sind. Mobile Geräte sind demnach bei der Nutzung der Schnittstelle nicht maßgeblich durch ihre limitierte Bandbreite benachteiligt.

## 6.2 Ausblick

Der gewährleistete Funktionsumfang reicht von der individuell konfigurierbaren Suche in verschiedenen Datenbanksystemen über den Abruf des Bestandsstatus konkreter Medien bis hin zu der Verwendung personalisierter Dienstleistungen, wie beispielsweise dem Anlegen neuer Favoritenlisten, Vormerkungen und dem Verlängern ausgeliehener Bücher.

Trotzdem können noch nicht alle Informationssysteme in ihrem vollem Funktionsumfang angesprochen werden. Es ist noch nicht möglich, neue Literaturbestellungen aufzu-

geben und Medien über die Schnittstelle auszuleihen. Die Ausarbeitung der Schnittstelle hinsichtlich dieser Punkte ist für die Zukunft ebenso geplant, wie die Einbindung neuartiger Systeme.
Darunter fällt ein mögliches Navigationssystem, das über Bluetooth Hotspots, sogenannte Beacons, die aktuelle Position eines Bibliotheksnutzers bestimmen kann. Durch die Einbindung dieses Systems in die REST-API könnte die mobile Navigation innerhalb der Bibliothek ermöglicht werden. Ein weiteres Projekt ist die Selbstausleihe über eine Barcode-Scanner App.
Wie zu sehen ist, existieren einige zukünftige Anwendungsgebiete in der Zentralbibliothek, besonders im mobilen Bereich. Ziel für die Zukunft sind mehrere mobile Client-Anwendungen, die die verschiedenen Aspekte der entwickelten Schnittstelle nutzen und einen Mehrwert zu den bestehenden Systemen ermöglichen.

# Literaturverzeichnis


[All17] Alliance, Open D.: *Open Database Connectivity*. http://www.opendatabasealliance.com/, 2017. – [Online; Entnommen: 10-August-2017]

[Apa17a] Apache: *Apache Solr*. https://lucene.apache.org/solr/, 2017. – [Online; Entnommen: 2-August-2017]

[Apa17b] Apache: *Apache Solr Wiki*. https://wiki.apache.org/solr/, 2017. – [Online; Entnommen: 2-August-2017]

[CER] CERN: *Invenio Digital Library Framework* . http://invenio-software.org/, . – [Online; Entnommen: 27-Juli-2017]

[Eas15] Easy, Selenium: *Find out broken links on websites* . http://www.seleniumeasy.com/sites/default/files/pictures/selenium/Httpstatuscodes.png, 2015. – [Online; Entnommen: 26-Juli-2017]

[EBS16] EBSCO: *EDS API Documentation* . http://edswiki.ebscohost.com/EDS_API_Documentation, 2016. – [Online; Entnommen: 10-August-2017]

[Fie00] Fielding, Roy: *Representational State Transfer (REST)* . https://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm, 2000. – [Online; Entnommen: 26-Juli-2017]

[FJ17a] Forschungszentrum Jülich, Zentralbibliothek: *JuSER* . http://juser.fz-juelich.de/?ln=de, 2017. – [Online; Entnommen: 3-August-2017]

[FJ17b] Forschungszentrum Jülich, Zentralbibliothek: *JuLib eXtended*. https://julib.fz-juelich.de/vufind/, 2017. – [Online; Entnommen: 3-August-2017]

[Ini17] Initative, OpenAPI: *OpenAPI*. https://www.openapis.org/about, 2017. – [Online; Entnommen: 16-August-2017]

[JL17] Josh Lockhart, Rob A. Andrew Smith S. Andrew Smith: *Slim Framework*. https://www.slimframework.com/, 2017. – [Online; Entnommen: 2-August-2017]

[JSO17] JSON: *JSON*. http://www.json.org/, 2017. – [Online; Entnommen: 18-August-2017]

[Kel13] Kelly, Mike: *HAL - Hypertext Application Language*. http://stateless.co/hal_specification.html, 2013. – [Online; Entnommen: 10-August-2017]

[Lib17] Library, Villanova University's Falvey M.: *VuFind - Search. Discover. Share.* . http://vufind.org, 2017. – [Online; Entnommen: 2-August-2017]

[Luc17] Lucene: *Apache Lucene Core*. https://lucene.apache.org/core/, 2017. – [Online; Entnommen: 2-August-2017]

[Mic09] Microsoft: *Introducing Windows Communication Foundation: Accessible Service-Oriented Architecture* . https://i-msdn.sec.s-msft.com/dynimg/IC400808.png, 2009. – [Online; Entnommen: 26-Juli-2017]

[MM10] Micheal McCandless, Otis G. Erik Hatcher H. Erik Hatcher: *Lucene in Action* . Manning Publications Co., 2010. – ISBN 978–1–933988–17–7. – Kapitel 1.2

[Mor17] Morris, Richard: *Swashbuckle - Swagger for WebApi 5.6.0* . https://www.nuget.org/packages/Swashbuckle, 2017. – [Online; Entnommen: 31-Juli-2017]

[Pay17] PayPal: *The REST APIs and HATEOAS*. https://developer.paypal.com/docs/api/hateoas-links/, 2017. – [Online; Entnommen: 10-August-2017]

[Rod15] Rodriguez, Alex: *RESTful Web services: The basics* . https://www.ibm.com/developerworks/library/ws-restful/index.html, 2015. – [Online; Entnommen: 29-Juli-2017]

[Sel17] SelfHtml: *XML*. https://wiki.selfhtml.org/wiki/XML, 2017. – [Online; Entnommen: 18-August-2017]

[Sin17] Sinatra: *Sinatra Framework*. http://www.sinatrarb.com/, 2017. – [Online; Entnommen: 2-August-2017]

[Sir17] SirsiDynix: *SirsiDynix Symphony*. http://www.sirsidynix.com/products/sirsidynix-symphony, 2017. – [Online; Entnommen: 2-August-2017]

[Sma17] SmartBear: *SmartBear Assumes Sponsorship Of Swagger API Open Source Project*. https://smartbear.com/news/news-releases/sponsorship-of-swagger/, 2017. – [Online; Entnommen: 16-August-2017]

[Sof17] Software, SmartBear: *Swagger* . https://swagger.io/, 2017. – [Online; Entnommen: 09-August-2017]

[Sol17] SolrMarc: *SolrMarc*. https://code.google.com/archive/p/solrmarc/, 2017. – [Online; Entnommen: 4-August-2017]

[Spi17] Spichale, Kai: *API-Design* . dpunkt.verlag, 2017. – ISBN 978–3–86490–387–8

[Sto13] Stoleru, Diana: *Rest* . https://blog.4psa.com/wp-content/uploads/Blog.Rest_.jpg, 2013. – [Online; Entnommen: 25-Juli-2017]

[Sto15] Stowe, Michael: *Hypermedia: The Good, the Bad, and the Ugly*. https://www.slideshare.net/mikestowe/hypermedia-the-good-the-bad-and-the-ugly, 2015. – [Online; Entnommen: 10-August-2017]

[Swa17] Swagger: *Swagger Codegen*. https://swagger.io/swagger-codegen/, 2017. – [Online; Entnommen: 16-August-2017]

[Til15] Tilkov, Eigenbrodt: *REST und HTTP* . dpunkt.verlag, 2015. – ISBN 978–3–86490–120–1

[Twi17] Twitter: *Twitter Developer Documentation REST APIs*. https://dev.twitter.com/rest/public, 2017. – [Online; Entnommen: 18-August-2017]

[VuF17a] VuFind: *Interface des VuFind Ils Treiber*. https://vufind.org/wiki/development:plugins:ils_drivers, 2017. – [Online; Entnommen: 3-August-2017]

[VuF17b] VuFind: *Konkreter VuFind Treiber für das ILS Symphony*. https://vufind.org/wiki/community:ils_support_list?s%5b%5d=unicorn, 2017. – [Online; Entnommen: 3-August-2017]

[w3s] W3SCHOOLS: *XML Soap* . https://www.w3schools.com/xml/xml_soap.asp, . – [Online; Entnommen: 26-Juli-2017]