Masterarbeit

im Studiengang Computerlinguistik

an der Ludwig- Maximilians- Universität München

Fakultät für Sprach- und Literaturwissenschaften

Department 2

Entwicklung eines WEB-basierten Faksimileviewers mit Highlighting von Suchmaschinen-Treffern und Anzeige der zugehörigen Texte in unterschiedlichen Editionsformaten

vorgelegt von
Matthias Lindinger

Betreuer:	Dr. Maximilian Hadersbeck
Aufgabensteller:	Dr. Maximilian Hadersbeck
Bearbeitungszeitraum:	09.03.2015 - 27.07.2015

Erklärung der Selbstständigkeit

Hiermit versichere ich, die vorliegende Arbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt sowie die Zitate deutlich kenntlich gemacht zu haben.

München, den 27.07.2015

Matthias Lindinger

Kurzfassung

In dieser Arbeit wird eine Software vorgestellt, die es möglich macht, eine komplette Edition in Form der Faksimile über den Webbrowser zugänglich zu machen. Diese kommt in der FinderApp WiTTFind zum Einsatz, die am Centrum für Informations- und Sprachverarbeitung an der Ludwig-Maximilians-Universität München entwickelt wurde. Zu jeder Textstelle, welche von der Suchmaschine gefunden wurde, existiert dabei ein Link, über den der Benutzer direkt ins Originaldokument springen und alle Treffer genauestens studieren kann. Neben den Faksimile sind auch die entsprechenden Transkriptionsextrakte in unterschiedlichen Formaten abrufbar.

Diese Arbeit gibt zuerst einen theoretischen Überblick über die Verfahren, wie aus den Bilddokumenten auf Paragraphenebene Koordinaten innerhalb des Bildes extrahiert werden können. Ebenso wird in einem kurzen Abriss vorgestellt, wie Web-Applikationen mit Client-Server-Kommunikation im Allgemeinen arbeiten.

In einem zweiten Teil wird die konkrete Umsetzung der Software erläutert. Dies reicht dabei vom Parsen von XML-Dateien über die Speicherung aller benötigten Daten in einer Datenbank bis hin zur tatsächlichen Darstellung der Koordinaten sowie der Transkriptionen.

Abschließend wird beschrieben, wie der Faksimileviewer konkret in die FinderApp integriert wurde und an welchen Stellen noch Möglichkeiten zur Verbesserung gegeben wären.

Danksagungen

In erster Linie möchte ich mich bei meinem Betreuer, Dr. Maximilian Hadersbeck, bedanken. Er stand mir während der gesamten Bearbeitungszeit mit Rat und Tat zur Seite, hat mich immer unterstützt und in den Besprechungen sehr wertvolle Ideen eingebracht.

Des Weiteren will ich Herrn Alois Pichler meinen Dank aussprechen, da er als zukünftiger Benutzer der entwickelten Software diese testete und ungemein hilfreiches Feedback gab. Außerdem hat er zusammen mit Herrn Hadersbeck die Arbeitsgruppe, in der diese Arbeit entstanden ist, erst ins Leben gerufen, was mich unglaublich freut.

Weitere wichtige Personen, die ebenso ihren Teil zum Gelingen der Arbeit beigetragen haben, sind die Arbeitsgruppenmitglieder Florian Fink und Stefan Schweter. Auch sie haben mir immer wieder wichtigen Input gegeben.

Ein ganz großes Dankeschön gebührt meinem Kommilitonen Roman Capsamun, der im Rahmen seiner Bachelorarbeit sehr wichtige Vorarbeiten geleistet hat. Ohne ihn wäre diese Arbeit in der Form sicher nicht möglich gewesen. An dieser Stelle möchte ich zudem die OCR-Gruppe am Institut erwähnen, welche die von Roman ermittelten Koordinaten für ganze vier Dokumente in einer unfassbaren Geschwindigkeit manuell nachkorrigiert hat.

Zu guter Letzt geht mein Dank an Dr. Josef Rothhaupt, durch dessen Vorstellung, man müsse in der Suchmaschine unbedingt auch das Faksimile sehen, dieser Teilbereich der Arbeitsgruppe erst angestoßen wurde.

Inhaltsverzeichnis

Kapitel 1

Einleitung

1.1 Motivation

Im Forschungsfeld der Digital Humanities stellt sich stets die Frage, wie die edierten Texte am ansprechendsten präsentiert werden. Gerade bei problematischen Editionen, bei der viele Textstellen vom Autor handschriftlich geändert und gestrichen wurden, ist es für die Wissenschaftler sehr wichtig, die gefundenen Textstellen der Suchmaschine im Faksimile zu sehen. Nur hier bekommen die Forscher den wahren Eindruck über den vorliegenden Text und können sogar Editionsfehler finden.

Suchmaschinen, die einen Nachlass durchsuchen und die Ergebnisse in textueller Form darstellen, gibt es viele. Auf der anderen Seite gibt es einige Angebote im Internet, die es dem Nutzer ermöglichen, die Faksimile zu betrachten. Eine Kombination dieser beiden Ansätze existiert bisher allerdings nicht. Das Ziel dieser Arbeit stellt die Entwicklung eines Faksimileviewers dar, der eine Suchmaschine für die zugehörige Edition im Hintergrund hat und die jeweiligen Treffer einer Suchanfrage direkt im digitalisierten Originaldokument darstellt. Zusätzlich soll es dem Benutzer erleichtert werden, durch die Anzeige der Transkription, die der Suchmaschine zugrunde liegt, Fehler in dieser zu finden und zu melden.

1.2 Ludwig Wittgensteins Nachlass[a]

Diese Arbeit bewegt sich stark im Kontext des Nachlasses von Ludwig Wittgenstein. Daher sollen die folgenden Unterpunkte einen groben Überblick über ihn als Person sowie seinen Nachlass und dessen Geschichte geben.

[a] entnommen aus [1]

1.2.1 Ludwig Wittgenstein als Person

Ludwig Wittgenstein war ein österreichisch-britischer Philosoph, geboren im Jahr 1889, der einer Industriellenfamilie entstammt. Seine Beiträge zur Philosophie der Logik können als sehr bedeutsam bezeichnet werden. Viele seiner Werke werden auch heute noch als wichtige Grundlage zweier philosophischer Schulen, des Logischen Positivismus und der Analytischen Sprachphilosophie, bezeichnet.

Im weiteren Verlauf seines Lebens verlegte er seinen Lebensmittelpunkt nach Cambridge und kam dort als Philosoph mit Bertrand Russell und George Edward Moore in Kontakt, bei denen er über sein eigenes Werk *Tractatus* promovierte. Später übernahm er die Nachfolge von Moore und in Folge dessen hielt er zahlreiche Vorlesungen. In diese Zeit fällt unter anderem das Big Typescript, welches ein Konzept eines Buches darstellt. Dieses verwarf er aber sehr schnell wieder.

Nach der Erstellung weiterer Manuskripte und Typoskripte erkrankte er im Jahr 1951 an Krebs und verstarb. Nach seinem Tod wurde sein zweites großes Werk veröffentlicht, *die Philosophischen Untersuchungen*, welches an vielen Textstellen auf das Big Typescript als Quelle verweist. [2]

1.2.2 Sein Nachlass

Der Nachlass wird vor allem an der Universität in Bergen (WAB) verwaltet. Diese Hochschule hat es sich in den letzten Jahren auf die Fahne geschrieben, den Nachlass für die Öffentlichkeit verfügbar zu machen. Dies geschah in Zusammenarbeit mit Oxford University Press.

1.2.3 Die Bergen Electronic Edition (BEE)

Im Angesicht dieses Vorhabens wurde im Jahr 2000 die Electronic Edition veröffentlicht. Die sechs CDs umfassende Sammlung enthielt eine CD mit den bearbeiteten Texten sowie Software und fünf weitere CDs mit den digitalisierten Originalschriftstücken in Form von Bilddateien. [3]

Die Texte Der Nachlass in Textform unterteilt sich in zwei unterschiedliche Versionen: auf der einen Seite die diplomatische, auf der anderen die normalisierte Ausgabe. Erstere bietet dem Leser eine Fülle an das Original betreffende Details, die von Streichungen und Ersetzungen bis hin zu Schreibfehlern reichen. Die normalisierte Version hingegen verzichtet auf gestrichene oder überschriebene Passagen, ebenso wurden Schreibfehler korrigiert und normalisiert. Ludwig Wittgenstein überarbeitete seine Texte auch häufiger und so entstanden verschiedene Varianten ein und derselben Bemerkung. Von diesen ist im Standardfall nur die chronologisch letzte in der normalisierten Fassung zu finden, frühere können bei Bedarf allerdings auch angezeigt werden. Des Weiteren ist auf der CD noch eine Suchmaschine für den kompletten Nachlass enthalten. Hierfür wurde FolioViews verwendet. [4]

Die Faksimile Die Bilddateien entstammen qualitativ hochwertigen Fotografien und wurden digitalisiert. Sowohl die diplomatische als auch die normalisierte Version sind seitenweise mit diesen verknüpft.

1.2.4 Wittgenstein Source & die Bergen Text Edition (BTE)

Wittgenstein Source ist eine Onlineplattform, die es Interessierten erlaubt, auf viele Werke von Wittgenstein in einem angenehmen Rahmen zuzugreifen und darin zu lesen. Zu diesem Zweck wurde die BEE überarbeitet und in von TEI geleitetes XML überführt. Der Vorrat an verwendeten XML-Tags enthält eine ganze Reihe an Möglichkeiten, den Text entsprechend zu gliedern - für diese Arbeit und alle damit verwandten sind insbesondere die Tags <ab> (Absätze) und <s> (Sätze) relevant, da auf ihnen das Highlighting basiert. Die XML-Struktur bildet so den Aufbau von Wittgensteins Dokumenten in Absätzen/Bemerkungen und Sätzen passend ab.

Zu finden sind auf den Seiten abgesehen von Texten (Bergen Text Edition) und Faksimile (Bergen Faksimile Edition) Informationen zu Forschungsgruppen, die sich mit der Weiterentwicklung von Wittgenstein Source beschäftigen. [5]

1.3 Wittgenstein in co-text[b]

Im Jahr 2010 wurde die Arbeitsgruppe *Wittgenstein im co-text* ins Leben gerufen, welche eine Zusammenarbeit des *Centrum für Informations- und Sprachverarbeitung* an der Ludwig Maximilians Universität München und den *Wittgenstein Archives at the University of Bergen* darstellt. Das Ziel dieser Gruppe ist es, eine digitale Bibliothek aus Werken von Ludwig Wittgenstein sowie von ihm gelesener und zitierter Literatur aufzubauen. Außerdem sollen computerlinguistische Werkzeuge entwickelt werden, mit Hilfe derer die Korpora besser im wissenschaftlichen Sinne verarbeitet werden können. Seit dem Jahr 2014 werden diese unter dem Namen $W_{ittgenstein}A_{dvanced}S_{earch}T_{ools}$ zusammengefasst.

Zu diesen Tools zählen unter anderem:

- das Vollformenlexikon WiTTLex, welches eine Untermenge des am CIS entwickelten CISLEX[c] darstellt
- eine lemmatisierte Vorschlagsuche, die auf der Software SIS[d] aufsetzt
- ein Grapheditor, der es erlaubt, lokale Grammatiken zu realisieren (siehe Abbildung 1.1)
- die statistische Suchmaschine Windex

[b] entnommen aus [1]
[c] eines der größten elektronischen Lexika des Deutschen
[d] symmetric index structures

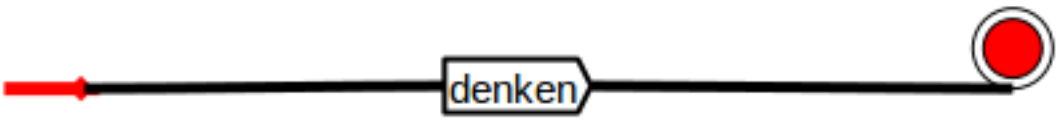

Abbildung 1.1: Beispiel für einen Graphen

1.4 Die Suchmaschine WiTTFind

Die zentrale Komponente der Arbeitsgruppe ist die Suchmaschine WiTTFind. Sie hat einen ganz anderen Ansatz als Windex, da sie regelbasiert arbeitet. Dabei können die Regeln aus verschiedensten Kategorien und sehr komplex aufgebaut sein, wie in Tabelle 1.1 beispielhaft zu sehen ist.

Kategorie	Beispiel	Bedeutung
lexikalisch		Nomen
		Verb in der 3. Person
syntaktisch	[NE]	Eigenname
	[VVFIN]	finites Verb
semantisch		Menschen
		Pflanzen
Modifikatoren	“string”	exakt dieser String
	*	beliebiges Zeichen

Tabelle 1.1: Beispiele für Regelkomponenten

Eine andere Herangehensweise an die Suche wird durch den bereits erwähnten Grapheditor gewährleistet. So kann der Benutzer seine Suchanfrage in Form eines Graphen gestalten, ohne etwas über die unterschiedlichen Modifikatoren zu wissen. An der internen Bedeutung der Anfrage ändert sich nichts, allerdings erlaubt dies einen deutlich intuitiveren Zugang.

Das grundlegende Design der Startseite[e] besteht aus mehreren Bestandteilen,

- einem Header mit Navigationsleiste,
- einer Liste mit allen durchsuchbaren Dokumenten,
- einem Suchschlitz und
- einem freien Bereich, in dem nach einer Anfrage die Treffer dargestellt werden,

wie in Abbildung 1.2 zu sehen ist.

[e] zu finden unter diesem Permalink

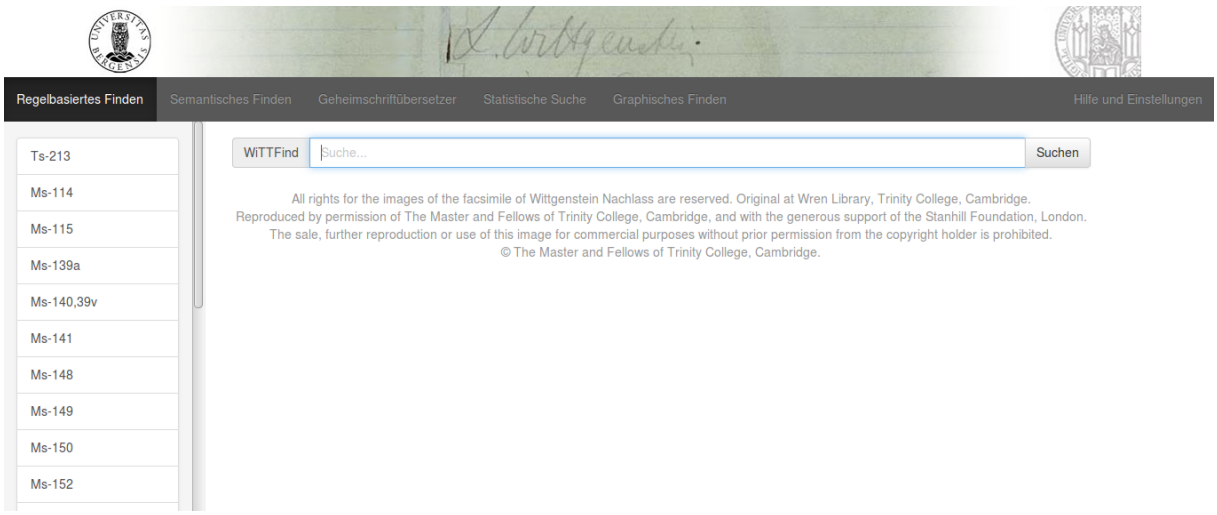

Abbildung 1.2: Startseite von WiTTFind

Sobald eine Suchanfrage erfolgreich durchgeführt wurde, werden alle Treffer in derselben Struktur angezeigt - diese ist in Abbildung 1.3 zu sehen. Es wird der Satz, in dem der Treffer liegt, zusammen mit dem eindeutigen Bezeichner der umfassenden Bemerkung, dem zugehörigen Ausschnitt des Faksimiles und einigen Links dargestellt. Besonders zu achten ist bei den Links auf denjenigen mit dem Titel *Facsimile-Reader*, an dieser Stelle wird das Endergebnis dieser Arbeit referenziert. Im weiteren Verlauf wird auch noch auf das Erstellen des Faksimile-Extrakts näher eingegangen.

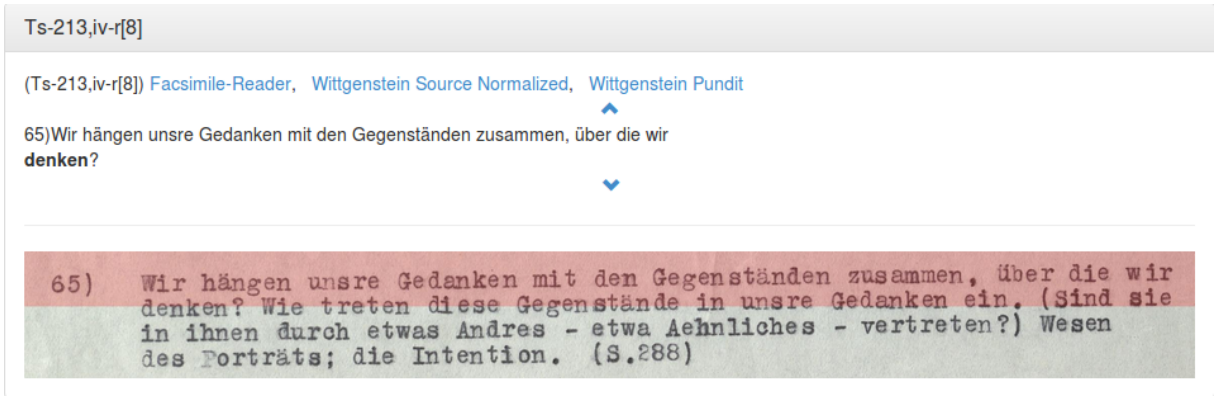

Abbildung 1.3: Beispiel der Trefferanzeige

Kapitel 2

Verwandte Arbeiten

Seit dem Jahr 2013 beschäftigt sich die Arbeitsgruppe neben vielen anderen Thematiken auch mit der Darstellung der Treffer der Suchmaschine WiTTFind im Originaldokument. Im Rahmen dieses Themas sind bereits einige Vorarbeiten zu dieser Arbeit geleistet worden, über welche dieses Kapitel einen groben Überblick gibt.

2.1 Bachelorarbeit Matthias Lindinger [1]

Diese Arbeit zeigt einen ersten Ansatz, wie ein Highlighting von den Treffern einer Suchmaschine im Faksimile realisiert werden könnte. Aus Wittgensteins Nachlass wird nur ein einziges Dokument betrachtet: das Big Typescript, kurz auch Ts-213 genannt. Die Strategie zur Gewinnung der Koordinaten auf Bemerkungsebene, die für ein Highlighting benötigt werden, beinhaltet dabei mehrere Schritte:

- Verarbeitung der Faksimile mit der OCR-Software ABBYY Finereader
- Erstellung eines Lexikons für die jeweilige Bemerkung aus der Transkription
- Extraktion der von der OCR gefundenen Koordinaten für alle erkannten Blockeinheiten auf der Seite
- Ranking der Koordinaten anhand der textuellen Inhalte des OCR-Outputs

Für das Ranking wird eine ganz simple Herangehensweise verwendet: für jeden erkannten Block auf der Seite wird das Verhältnis

im Lexikon der aktuellen Bemerkung enthaltene Wörter / alle Wörter

berechnet. Derjenige Block, der letztendlich den besten Wert aufweist, wird als der am wahrscheinlichsten korrekte angenommen und die zugehörigen Koordinaten zusammen mit dem aktuellen einheitlichen Bemerkungsbezeichner abgespeichert.

Da bei diesem Prozess naturgemäß Fehler passieren, ist die Arbeit um ein Kommandozeilen-Korrekturtool erweitert, welches es dem Benutzer erlaubt, die gespeicherten Koordinaten über die Ansicht im Webbrowser manuell nachzubessern.

2.2 Masterarbeit Claudia Müller [6]

Ziel dieser Arbeit, die auf der vorherigen aufsetzt, ist es, Techniken zu entwickeln, welche zusätzlich zum paragraphenbasierten ein zeilen- oder sogar wortweises Highlighting erlauben. Zu diesem Zweck werden Methoden vorgestellt, die sowohl auf Grund von Parsing des OCR-Outputs als auch der handannotierten Texte eine Berechnung der Koordinaten ermöglichen.

Eine zentrale Erkenntnis ist es, dass eine Kombination von beidem in der Regel am zuverlässigsten funktioniert. Ein äußerst wichtiges Ergebnis, gerade im Hinblick auf zukünftige Arbeiten in dieser Thematik, stellt die erstmalige Verwendung des Levenshtein-Abstandes dar, der in der Computerlinguistik zu verschiedenen Themen verwendet wird. So ist es möglich, Texterkennungsfehler der OCR bis zu einem gewissen Grad zu kompensieren.

2.3 Bachelorarbeit Roman Capsamun [7]

Im Rahmen dieser Arbeit werden die Erkenntnisse und Ergebnisse der beiden zuvor genannten kombiniert und erweitert. Eine wichtige Neuerung stellt die Verwendung der OpenSource-OCR-Software *tesseract* von Google dar, welches ABBYY Finereader ersetzt.

Im Gegensatz zu den bisherigen Ansätzen werden hier die kompletten 5000 Seiten des Nachlasses von Wittgenstein, die frei verfügbar sind, betrachtet und verwendet. Daher kommen neben dem Big Typescript weitere Typoskripte sowie einige Manuskripte hinzu. Bei ersteren werden wie bisher die OCR-Outputs geparsed, mit Hilfe des Levenshtein-Abstandes bewertet und die jeweiligen Koordinaten gespeichert. Für die Manuskripte hingegen produziert OCR in der Regel nur Textmüll - daher werden hierfür Heuristiken vorgestellt, die anhand von Metainformationen über den edierten Text wie die Anzahl der Zeilen, durchschnittliche Zeilenhöhe, leere Zeilen zwischen den Bemerkungen etc. Koordinatenvorschläge berechnen.

Im weiteren Verlauf wird eine Web-Applikation vorgestellt, die es einem Benutzer über eine Weboberfläche erlaubt, die Vorschläge für die Koordinaten zu optimieren und auf einem Server abzuspeichern.

Die in der Arbeit erstellte Software ist Teil einer Pipeline, die vom Faksimile hin zum Highlighting innerhalb der Suchmaschine führt. Diese Pipeline wird in Kapitel 3 näher vorgestellt.

Teil I

Theorie

Die folgenden Kapitel geben einen Einblick auf die theoretischen Grundlagen, welche für die Thematik der Arbeit unerlässlich sind.

Kapitel 3

Vom Faksimile zum Highlighting

Um vom Faksimile mit Hilfe von OCR und einer XML-Edition zum Highlighting im Browser, das sich gut mit CSS lösen lässt, zu kommen, sind mehrere Schritte notwendig.

3.1 Ermittlung der Koordinaten

Zur Realisierung eines Highlightings werden neben eindeutigen Bezeichnern insbesondere Koordinaten der einzelnen Paragraphen innerhalb eines Bildes gebraucht. Zu deren Ermittlung sind in der Arbeitsgruppe *Wittgenstein im co-text* bestimmte Techniken im Einsatz, die sich je nach Dokumentenart unterscheiden.

Auf der einen Seite gibt es Typoskripte, die vorwiegend aus maschinengeschriebenen Texten bestehen. Für diese Art von Dokumenten ist der Vorgang folgendermaßen aufgebaut:

1. Verarbeitung der Faksimile mit einer OCR-Software
2. Extraktion der Koordinaten der Zeilen, die die OCR erkannt hat
3. Extraktion der Texte der Zeilen aus der XML-Datei
4. Alinierung von XML-Text und von OCR-Text
5. Speicherung der absoluten Koordinatenvorschläge für die Paragraphen in einer JSON-Datei

Bei den Manuskripten - handgeschriebenen Dokumenten - gestaltet sich das Verfahren anders. Da sich die OCR in der Regel für handschriftliche Texte nicht eignet, weil hauptsächlich Textmüll erkannt wird. wird dieser Schritt in diesem Fall weggelassen. Daher lässt sich das Vorgehen hier in die folgenden Schritte aufteilen:

1. Extraktion des Textes der Paragraphen
2. Ermittlung der Zeilenanzahl innerhalb des Paragraphen
3. Ergänzung der Ergebnisse um eine Zusatzzeile zwischen den Paragraphen

4. Start mit empirischen Werten
5. Verteilung der ermittelten Werte auf die Gesamtseite
6. Speicherung der absoluten Koordinatenvorschläge für die Paragraphen in einer JSON-Datei

Da bei diesen Vorgängen naturgemäß Fehler auftreten können, wird ein Korrekturtool angeboten, über das Mitarbeiter die gespeicherten Koordinatenvorschläge manuell überarbeiten können. Zu diesem Zweck wird die Sammlung aller Seiten im Dokument in Pakete aufgeteilt. Sobald dies geschehen ist, geben die Helfer über eine Liste an, welche Pakete sie aktuell korrigieren und ob sie bereits fertig sind. Nach Abschluss eines Dokuments werden zuletzt die optimierten Daten wieder in die Sammlung eingespeist.

3.2 Darstellung der Ergebnisse im Browser

Für eine Darstellung im Browser müssen die Koordinaten in einem ersten Schritt in relative Werte, verhältnismäßig zu Breite und Höhe des Faksimile, umgerechnet werden. Dies ist unerlässlich, da sich die Anzeige bei Web-Applikationen stets an den Möglichkeiten des Betrachters orientieren muss - in diesem Fall ist die Auflösung des Bildschirms besonders relevant.

Im Anschluss können HTML-Elemente, die die Koordinaten repräsentieren, zusammen mit dem Faksimile in eine Website eingefügt werden und über die Verwendung von Stylesheets beliebig dargestellt werden. Details zu diesem Thema sind in Kapitel 7 nachzulesen.

Kapitel 4

Web-Applikationen

Dieses Kapitel gibt einen Überblick über verschiedene Arten, eine Web-Applikation zu entwickeln. Die Grundlagen wie die Client-Server-Kommunikation bleiben dabei vom Schema her allerdings identisch. Abbildung 4.1 zeigt ein Beispiel für eine Kommunikation zwischen Client und Server.

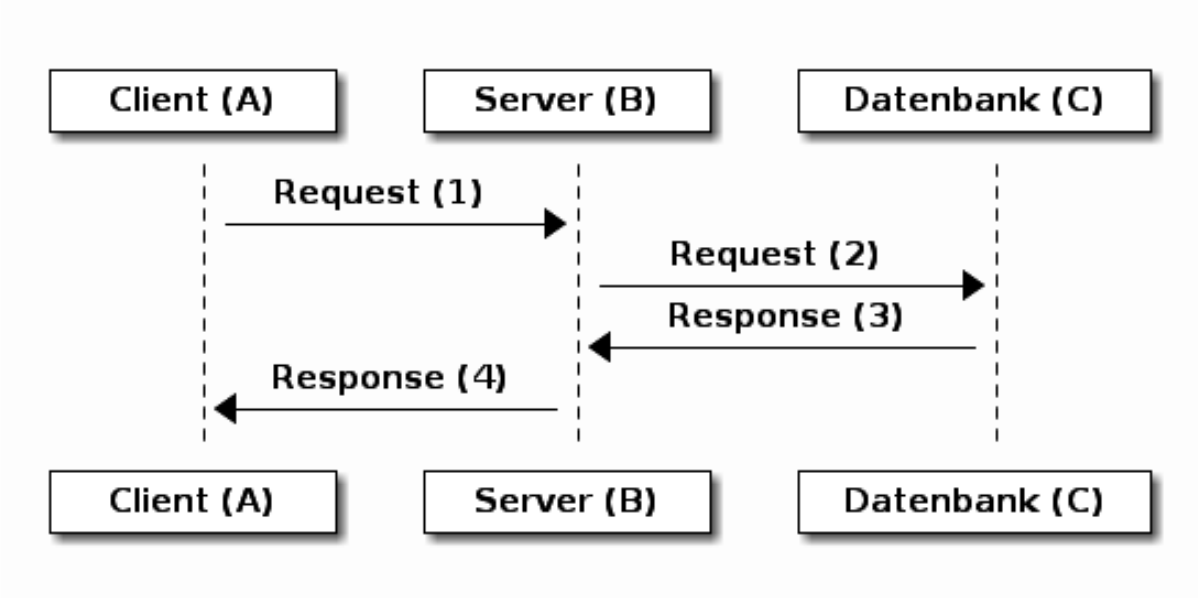

Abbildung 4.1: Client-Server-Kommunikation [8]

Anfragen an den Server können dabei nach dem HTTP-Protokoll verschiedene Methoden haben, für diese Arbeit sind ausschließlich `GET` und `POST` relevant. Auf Serverseite ist wiederum anhand von Routen festgelegt, welche Aktion bei einem Request ausgeführt werden soll. [8]

4.1 Das LAMP-Softwarepaket

Eine Möglichkeit, Web-Applikationen mit dynamischen Webseiten zur Verfügung zu stellen, stellt das Softwarepaket - oder auch häufig als Stack bezeichnet - LAMP dar. Beim Namen handelt es sich um ein Akronym, welches die Anfangsbuchstaben der beteiligten Komponenten zusammenfasst:

- Linux
- Apache Webserver
- MySQL Datenbank
- PHP Skriptsprache

> Diese Programmkombination definiert im Sinne einer Software-Distribution eine Infrastruktur, in deren Rahmen dynamische Webseiten und -anwendungen entwickelt und bereitgestellt werden können.[a]

Geprägt wurde das Akronym LAMP vom Heise-Autor Michael Kunze. Die Einzelkomponenten können allerdings auch durch Alternativen ersetzt werden, z.B. Linux durch Windows, Apache durch nginx, MySQL durch Postgresql und PHP durch Perl oder Ruby. Daher existieren ebenfalls andere Kombinationen, wie WAMP (mit Windows) oder MAMP (mit MacOS). Ein äußerst bekannter Bezeichner, der häufig im Internet zu finden ist, ist XAMPP - hier beschreibt das X die Unabhängigkeit von Plattformen und das doppelte P steht für ein Zusammenwirken von PHP und Perl. Es kann problemlos auch auf einem Heimrechner installiert und verwendet werden. [9]

Eine schematische Übersicht über das Zusammenspiel der Komponenten des Softwarepakets ist in Abbildung 4.2 zu sehen. Die einzigen Spezialisierungen in dieser Struktur sind diejenigen Stellen, an denen der Skriptinterpreter einerseits mit dem Webserver und andererseits mit der Datenbank kommuniziert.

Sobald eine Anfrage nun die Netzwerkkarte erreicht, entscheidet das Betriebssystem anhand der Portnummer, welches Programm zur Verarbeitung vorgesehen ist - in diesem Fall der Apache Webserver - und leitet die Anfrage an die entsprechende Stelle weiter. Bezieht sich die Anfrage auf statische Daten, wie HTML-Dateien oder JPG-Bilder, werden diese aus dem Dateisystem geholt und zurückgeschickt. Im Fall von dynamischen Daten, die hier durch den PHP-Skriptinterpreter erzeugt werden, wird dieser gestartet und erstellt den HTML-Code. Wenn zusätzliche Daten aus der Datenbank benötigt werden, stellt PHP eine Verbindung her und extrahiert sie. Sobald dieser Vorgang abgeschlossen ist, werden die erzeugten Inhalte an der Client übermittelt. [9]

Handelt es sich um einen POST-Request, sollen in der Regel nur Daten gespeichert werden. Hier wird nach Abschluss nur ein Statuscode an den Client zurückgegeben werden.

[a] entnommen aus [9]

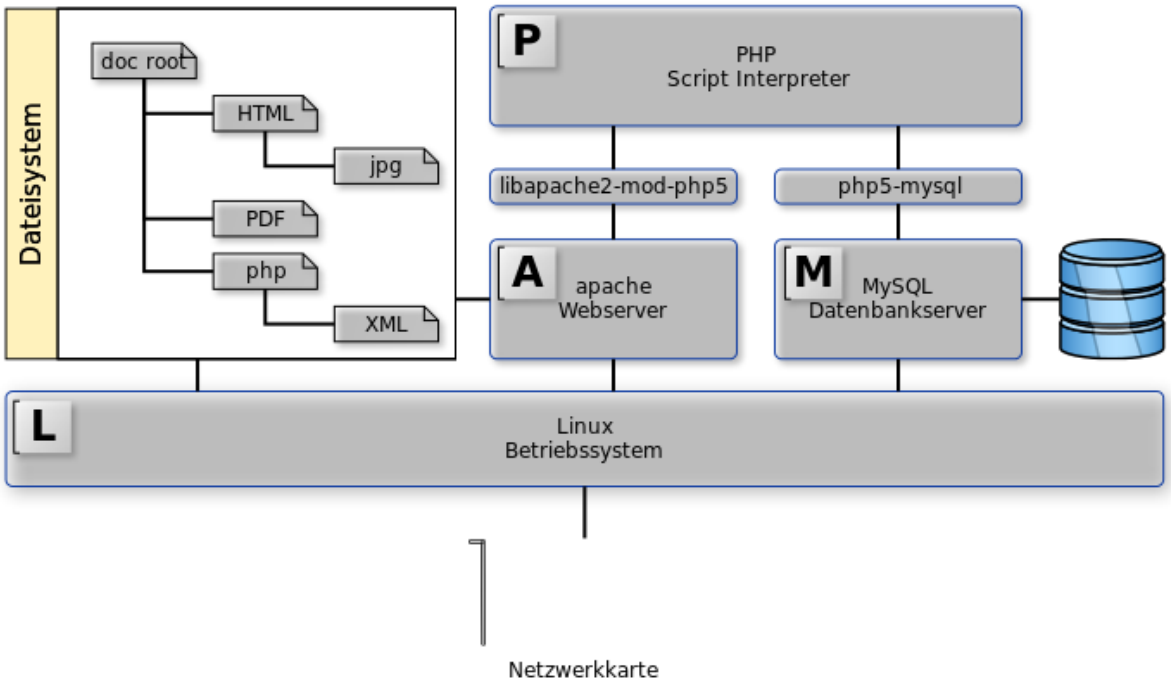

Abbildung 4.2: Strukturierung des LAMP Stacks [9]

4.2 Der MEAN-Stack

Beim MEAN-Stack handelt es sich um eine Ansammlung von Frameworks, die es einem Programmierer erlauben, moderne Webseiten zu erstellen. Von klassischen Webseiten unterscheidet es sich insofern, dass immer nur Teilbereiche neu geladen werden. Aus diesem Zusammenhang heraus werden derartige Applikationen auch oftmals als Single Page Web-Applications, kurz SPA, bezeichnet. [8]

Ein wichtiger Unterschied zu klassischen Webseiten, wie in Kapitel 4.1 beschrieben, besteht darin, dass viel Arbeit, die normalerweise auf dem Server erledigt wird, auf den Client, d.h. den Browser des Benutzers, ausgelagert wird. Dadurch sind die Webseiten deutlich performanter und der Server wird merklich entlastet.

Wie auch beim LAMP-Softwarepaket handelt es sich beim MEAN-Stack ebenso um ein Akronym. Hierbei steht jeder Buchstabe für eine Komponente, diese werden in den folgenden Unterpunkten näher erläutert.

4.2.1 Node

Die zentrale Komponente des MEAN-Stack ist Node. Dieses ermöglichst es einem Programmierer, einen eigenen Webserver zu implementieren. Dies lässt sich bereits mit wenigen Zeilen erreichen, wie im Snippet 4.1[b] dargestellt. In diesem Beispiel wird ein Server erstellt, der auf Port 1337 läuft und auf alle Anfragen mit demselben Ergebnis antwortet:

- Statuscode 200, erfolgreicher Request,
- Content-Type plain text,
- String “Hello World”

```
1 var http = require('http');
2 http.createServer(function (req, res) {
3   res.writeHead(200, {'Content-Type': 'text/plain'});
4   res.end('Hello World\n');
5 }).listen(1337, '127.0.0.1');
```

Listing 4.1: Beispiel für einen Node-Server

Da auf einem Webserver in der Regel bereits eine Apache-Instanz auf Port 80 läuft, muss für den Node-Server ein separater Port angegeben werden - im Beispiel wäre er dann über Port 1337 erreichbar.

[b] Snippet entnommen aus [10]

4.2.2 Express

Bei Express handelt es sich um ein Mini-Framework, das direkt auf Node aufsetzt und insbesondere die Definition von Routes enorm erleichtert. [8]

Das Snippet 4.2 ist in folgende Schritte aufgeteilt:

- Einbindung des Express-Moduls
- Erstellung der Express-Variable und einer weiteren Variable zum Speichern von Daten
- Definition einer Route exampleA für einen GET-Request, req.query enthält dabei die übergebenen Parameter
- Definition einer Route exampleB für einen POST-Request, req.body enthält dabei die übergebenen Parameter

```
1  var express = require('express');
2  ...
3  var app = express();
4  var value;
5  ...
6  app.get('/exampleA',function(req,res){
7    value = req.query.xyz;
8    res.header('Content-Type', 'text/plain');
9    res.header('Charset', 'utf-8');
10   res.send('Hello World!');
11 });
12 ...
13 app.post('/exampleB',function(req,res){
14   value = req.body.xyz;
15   res.header('Content-Type', 'text/plain');
16   res.header('Charset', 'utf-8');
17   res.send('Hello World!');
18 });
```

Listing 4.2: Beispiel für Express-Routes

4.2.3 Die Datenbank MongoDB

Für viele Web-Applikationen wird neben den anderen Komponenten auch eine Datenbank benötigt, um Daten zu speichern und wieder abzufragen. Im Gegensatz zum LAMP-Stack verwendet der MEAN-Stack die MongoDB. Der Name leitet sich von *humounguous*, zu deutsch *riesig*, ab. Bei dieser Datenbank handelt es sich um eine NoSQL- bzw. dokumentenorientierte

Datenbank, welche die Daten in JSON[c]-Objekten ablegt. Insbesondere durch diesen Fakt ist sie hervorragend für das Zusammenspiel mit den restlichen Komponenten geeignet, die Javascript-basiert programmiert sind. [8]

Bei der Installation der MongoDB auf dem Rechner wird das Kommandozeilentool *mongo* mitgeliefert, mit dem sich Datensätze anlegen, verändern, löschen und durchsuchen lassen. Das Ergebnis der Operationen in Snippet 4.3 sind alle Datensätze der Datenbank *foo* in der Collection *bar*, deren Attribut *ab* den Wert *xy* haben. Falls kein Datensatz zur Anfrage passt, wird ein leeres Array zurückgegeben. Für den Fall, dass mindestens ein oder mehrere passende Datensätze existieren, enthält das Rückgabe-Array diese. Wie zu sehen ist, bekommt jedes eingefügte Objekt neben allen Attributen außerdem ein internes Attribut *_id*.

```
1 $ mongo
2 > use foo
3 > db.getCollection("bar").find({"ab":"xy"})
4 > { "_id" : ObjectId("5591b8040c0f440aad28e7e8"), "ab" : "xy", "cd" :
      "z" }
```

Listing 4.3: Beispiel-Query für die MongoDB

Die Query-Syntax gestaltet sich in der Regel folgendermaßen:

```
1 {<Attribut>:<gewünschter Wert>}
```

Listing 4.4: Allgemeine Syntax einer Query für die MongoDB

Neben dieser einfachen Art der Anfrage existieren noch eine Reihe von Operatoren, mit denen sich die Query beliebig spezifizieren lässt. Eine Auflistung ist unter [11] zu finden.

4.2.4 Angular

AngularJS stellt die clientseitige Komponente des MEAN-Stack dar. Damit lassen sich dynamisch Inhalte der Website darstellen. Für diese Arbeit ist es allerdings nicht weiter relevant.

4.2.5 Jade

Ein weiteres Tool, das zwar nicht direkt zum MEAN-Stack dazugehört, aber häufig in dessen Kontext verwendet wird, ist Jade. Dabei handelt es sich um ein Template-System, mit dem sich HTML deutlich kompakter und übersichtlicher schreiben lässt. Über Variablen lässt sich der entstehende Code dynamisch gestalten. [12]

Dieses Snippet[d] in Jade

[c] Kurzform für JavaScript Object Notation
[d] Beispiel entnommen aus [13]

```
doctype html
html
  head
    title my jade template
  body
    h1 Hello #{name}
```

wandelt sich bei Übergabe von

```
{"name": "user"}
```

in

```
<!DOCTYPE html>
<html>
  <head>
    <title>my jade template</title>
  </head>
  <body>
    <h1>Hello user</h1>
  </body>
</html>
```

um.

Im Zusammenspiel mit Node lässt sich eine in Jade-Syntax verfasste Eingabe problemlos in für den Browser lesbaren HTML-Code umwandeln. Dies soll Snippet 4.5 visualisieren.

```
var app = express();
...
app.set('view engine','jade');
...
app.get('/hello',function(req,res){
  res.render('index',{"name":"user"});
});
```

Listing 4.5: Beispielcode zum Rendern eines Jade-Templates

4.3 Gegenüberstellung[e]

Wenn LAMP- und MEAN-Stack miteinander verglichen werden, stellt sich sehr schnell heraus, dass es dabei kein *entweder oder* gibt. Durch die Flexibilität der beiden Pakete lassen sich Komponenten aus dem einem mit solchen aus dem anderen kombinieren. Es lässt sich so theoretisch ein neuer Stack mit dem Namen LAMP definieren, wobei der Buchstabe *M* nicht mehr für MySQL steht, sondern für MongoDB. Genauso ist es auch andersherum möglich, eine neue Zusammenstellung namens MEAN festzulegen, deren Komponente *M* für MySQL steht.

Einen sehr großen Vorteil des MEAN-Stack stellt dar, dass alle Bestandteile auf der Basis von Javascript arbeiten. Daher ist es nicht nötig, für die Entwicklung einer Applikation, die auf MEAN basiert, weitere Programmiersprachen zu erlernen.

Kritiker bemängeln häufig, dass die MongoDB für kleine wie mittlere Projekte funktionieren mag - sobald die Benutzeranzahl allerdings in die Hundert-Tausende steigt, reicht sie nicht mehr aus. Das stellt aber in keiner Weise ein Problem dar, da problemlos auch eine MySQL-Datenbank verwendet werden kann.

[e] vergleiche [14]

Teil II

Praxis

In den nachfolgenden Kapiteln wird die praktische Implementation des Faksimileviewers näher beleuchtet - dies reicht von von der grundlegenden Architektur bis zu Details wie der Anzeige der edierten Texte.

Kapitel 5

Allgemeines zum Projekt

Die Applikation ist Teil der WAST-Gruppe im CIS-Gitlab und unter folgender URL zu finden:

https://gitlab.cis.uni-muenchen.de/wast/quadroreader

Außerdem ist das Repository auf der beigelegten CD im Ordner *quadroreader* enthalten.

5.1 Git

Sie ist folglich mit Git versioniert und nach dem sehr bekannten *git branching model* aufgebaut, welches hier genauer studiert werden kann.

5.2 Verzeichnisstruktur

Die Ordnerstruktur des Repository gestaltet sich dabei folgendermaßen:

```
1  -- bower_components     # enthält alle clientseitigen Abhängigkeiten
2                          # wird im Deploy-Prozess erstellt
3  -- data                 # enthält die Index-Files im JSON-Format
4  -- facsimile            # enthält die Faksimile
5                          # muss dazugelinkt werden
6  -- node_modules         # enthält alle serverseitigen Abhängigkeiten
7                          # wird im Deploy-Prozess erstellt
8  -- public               # enthält alle Dateien für die Darstellung
9     -- images            # Bilddateien
10    -- javascripts       # eigene clientseitige Javascriptdateien
11    -- stylesheets       # CSS-Dateien
12 -- tools                # enthält zusätzliche Hilfsskripte
13 -- transcription        # enthält die XML-Transkriptionen
```

```
14                          # muss dazugelinkt werden
15 -- views                 # enthält alle Jade-Templates
16
17 bower.json               # Liste aller clientseitigen Abhängigkeiten
18 package.json             # Liste aller serverseitigen Abhängigkeiten
19 wittreader-server.js     # der WiTTReader-Server
```

Das Faksimile-Verzeichnis muss dabei nach diesem Schema aufgebaut sein:

```
1 -- wittgenstein
2    -- <Dokumentname> (bspw. Ts-213)
3       -- high_density_jpg
4       -- high_density_small
```

Beim Transkriptionsverzeichnis ist die Struktur irrelevant, da beim Speichern der XML-Daten in die Datenbank rekursiv nach allen Dateien, deren Name auf *NORM.xml* oder *DIPL.xml* endet, gesucht wird - der genaue Ablauf wird in Kapitel 8 erläutert.

5.3 Abhängigkeiten

Um einen Deploy-Vorgang erfolgreich vorzunehmen, wird die folgende Software benötigt:

- make
- git
- nodejs
- mongodb
- curl
- imagemagick
- cpanm

Des Weiteren müssen folgende Perl-Module installiert sein:

- MongoDB
- File::Spec
- JSON::PP
- Data::Compare
- XML::Twig

5.4 Docker

Für die Verwendung der Applikation auf dem eigenen Rechner ist im Repository ein *Dockerfile*[a] enthalten, das einen Container mit aller benötigter Software repräsentiert. Mit dem Befehl **make docker** kann dieser gebaut und gestartet werden. Um die Faksimile und die Transkriptionen im Container verfügbar zu haben, gibt es die beiden Mountpoints *FACSIMILE_MOUNT* und *TRANSCRIPTION_MOUNT*.

Über den Aufruf von

```
1 make FACSIMILE_MOUNT=/absolute/path/to/the/facsimile
    TRANSCRIPTION_MOUNT=/absolute/path/to/the/transcription/files
    docker
```

werden die angegebenen Verzeichnisse an der richtigen Stelle im Container eingebunden, die Daten in die Datenbank eingepflegt und der Deploy-Vorgang angestoßen.

[a] genauere Informationen auf der Docker-Website

Kapitel 6

Implementation als Node.js-Applikation

Dieses Kapitel widmet sich der konzeptionellen Architektur, die dem Faksimileviewer zugrunde liegt. Dabei handelt es sich um eine Client-Server-Architektur - auf der Serverseite kommt hierbei das Framework *Node.js* zusammen mit einigen Modulen zum Einsatz, auf der Clientseite wiederum wird pures Javascript in Kombination mit der Bibliothek *jQuery* verwendet.

6.1 Der Server

Der Server benutzt neben Node.js auch noch eine Reihe von weiteren Modulen, welche in der Datei *package.json* (siehe Snippet 6.1) unter dem Key *dependencies* zusammengefasst sind. Es handelt sich dabei um Key-Value-Paare von Modulname und Versionsnummer.

```
12 "dependencies": {
13   "body-parser": "~1.12.0",
14   "cors": "*",
15   "express": "~4.12.2",
16   "gitlab": "*",
17   "gm": "*",
18   "jade": "~1.9.2",
19   "morgan": "~1.5.1",
20   "mongodb": "*"
21 }
```

Listing 6.1: Die Abhängigkeiten des Servers

Im Server selbst, der in der Datei *wittreader-server.js* zu finden ist, werden die angegebenen Abhängigkeiten mit einem `require`-Statement geladen (siehe Snippet 6.2). Danach stehen alle Module zur Verfügung.

```
4  var express = require('express');
5  var http = require('http');
6  var path = require('path');
7  var logger = require('morgan');
8  var bodyParser = require('body-parser');
9  var mongoClient = require('mongodb').MongoClient;
10 var imageProcessor = require('gm').subClass({imageMagick: true});
11 var cors = require('cors');
```

Listing 6.2: Import der Abhängigkeiten

Von ganz zentraler Bedeutung ist Express.js, um die Definition der Routes zu erleichtern - wie das genau funktioniert, ist bereits in Kapitel 4.2.2 erklärt. Die definierten Routes werden Stück für Stück in den kommenden Kapiteln näher beleuchtet.

Mit dem Express-Module wird über den Befehl **var app = express();** eine Applikation definiert, die in Verwendung auf Serveranfragen so reagiert, wie es festgelegt wurde. Danach lassen sich noch einige Einstellungen vornehmen, unter anderem

- das Setzen des Ports, über **app.set('port',process.env.PORT || 3000);**
- das Setzen des Verzeichnisses für die Weboberflächen, über **app.set('views',path.join(__dirname, 'views'));**
- das Setzen der Engine zum Rendern der Oberflächen, über **app.set('view engine','jade');**
- das Setzen der von außen zugänglichen Dateipfaden, über **app.use(express.static(path.join(__dirname, 'public')));** (wird als erstes Argument ein String übergeben, wird als Oberverzeichnis dieses genommen statt Root /)

Zum Abschluss wird mittels

```
285 http.createServer(app).listen(app.get('port'),function(){
286   console.log('Express server listening on port ' + app.get('port'));
287 });
```

ein Server angelegt, der auf dem vorher angegebenen Port läuft.

6.1.1 Verbindung zur Datenbank

Um eine Verbindung zur Datenbank herzustellen, wird das Node-Modul *mongodb* hergenommen. Die Syntax für diesen Prozess ist in Snippet 6.3 dargestellt. Dabei kann der Port, auf dem die Datenbank läuft, vom Benutzer über Setzen der Environment-Variable **MONGOPORT** angepasst werden. Darauf geht Kapitel 6.1.3 näher ein.

```
29 var database;
30 mongoClient.connect('mongodb://localhost:'+(process.env.MONGOPORT ||
       27017)+'/wittreader', function(err, db) {
31   database = db;
32 });
```

Listing 6.3: Herstellen der Verbindung zur Datenbank

Nach dem Starten des Servers hält die Variable database den Kontakt zur MongoDB aufrecht und kann über die Ausführung von

```
63 app.use(function(req,res,next){
64   req.db = database;
65   next();
66 });
```

in allen definierten Routes verwendet werden.

6.1.2 Speicherung der Daten

Zusätzlich zu den Koordinaten und den Transkriptionen, die in der Datenbank abgelegt und von dort wieder abgefragt werden - darauf gehen die Kapitel 7 und 8 näher ein - existieren noch weitere Daten, die ebenfalls in der Datenbank *wittreader* innerhalb der Collection *serverData* gespeichert werden. Diese wird mit dem Code in Snippet 6.4 initialisiert, es wird eine Regel erstellt, die jeden Datensatz eine Stunde nach Erstellung wieder automatisch entfernt, und ein Datensatz für Demonstrationszwecke gespeichert. Unter anderem für diese Aufgabe existiert das Make-Target **database−insert**, welches daneben auch noch andere Dinge erledigt.

```
1  db.createCollection("serverData");
2  db.getCollection("serverData").createIndex({"createdAt":1},{
       expireAfterSeconds:3600});
3  db.getCollection("serverData").insert(
4    {
5      "dataId":"demo",
6      "documentId":"Ts-213",
7      "hits": {"Ts-213":{"Ts-213,i-r":["Ts-213,i-r[1]"],"Ts-213,31r":["
       Ts-213,31r[2]"]},"Ms-114":{"Ms-114,60r":["Ms-114,60r[5]"]}},
8      "startSiglum":"Ts-213,i-r",
9      "facsimileDensity":"high_density_jpg",
10     "scaleFactor":1
11   }
12 );
```

Listing 6.4: Initlialisierung der Collection zur Speicherung von externen Daten

Zu jedem Datensatz wird eine eindeutige ID angegeben, um später wieder darauf zugreifen zu können, und das Erstelldatum gespeichert. Letztere Information ist elementar, damit die Datensätze nach einer definierten Zeitdauer wieder aus der Datenbank entfernt werden können. Im Einzelnen handelt es sich um die Variablen

- **documentId**, den Bezeichner des ausgewählten Dokuments
- **hits**, die von der Suchmaschine gefundenen Treffer (Bemerkungssiglen unterteilt nach Dokument und Seite, Beispiel siehe Snippet 6.6)
- **startSiglum**, das zuerst anzuzeigende Siglum
- **facsimileDensity**, die aktuell vom Benutzer gewählte Auflösung der Bilder
- **scaleFactor**, die aktuell vom Benutzer gewählte Skalierung des Readers

welche von extern gespeichert werden. Dazu wird die Route in Snippet 6.5 verwendet. **hits** und **startSiglum** müssen hierbei nicht zwingend angegeben werden - für diesen Fall sind beide einfach leer. Besonders wichtig ist an dieser Stelle die Verwendung des Moduls *cors*,[a] welches es auch externen Aufrufern ermöglicht, diese Route anzufragen:

```
78 app.post('/data',cors(),function(req,res){
79   var dataId = req.body.dataId;
80   var db = req.db;
81   var collection = db.collection("serverData");
82   collection.findOne({"dataId":dataId},function(e,serverData){
83     var data = {
84         "dataId":dataId,
85         "documentId":req.body.documentId,
86         "hits":(typeof req.body.hits === "undefined" ? {} : JSON.
    parse(req.body.hits)),
87         "startSiglum":req.body.siglum || "",
88         "facsimileDensity":req.body.density || "high_density_jpg",
89         "scaleFactor":req.body.scaleFactor || 1,
90         "createdAt":new Date
91     };
92     if(serverData){
93       collection.update({"dataId":dataId},data);
94     }else{
95       collection.insert(data);
96     }
97   });
```

[a] eine Abkürzung für *cross origin resource sharing*

```
98 });
```

Listing 6.5: Route zur Speicherung von externen Daten

```
{
    "Ts-213": {
        "Ts-213,i-r": [
            "Ts-213,i-r[1]"
        ],
        "Ts-213,31r": [
            "Ts-213,31r[2]"
        ]
    },
    "Ms-114": {
        "Ms-114,60r": [
            "Ms-114,60r[5]"
        ]
    }
}
```

Listing 6.6: Beispielstruktur der Suchmaschinen-Treffer

Daneben gibt es noch weitere Variablen, die intern definiert sind:

- **factor**, das Verhältnis von Breite zu Höhe der Faksimile
- **index**, die Indexfiles, welche ein Mapping *Seitensiglen <-> interne Seitennummer* je Dokument enthalten
- **originRectoX**, die X-Koordinate, von der aus die Koordinaten auf den rechten Seiten zu rechnen sind

6.1.3 Verwaltung des Servers

Zur Verwaltung des Servers existiert ein Makefile, welches Targets für unterschiedliche Aufgaben enthält - das wichtigste hierbei ist **deploy**, welches folgendermaßen aufgebaut ist:

```
deploy: $! check build start
    $(SILENT) $(cmd_check_running) \
    && (echo WiTTReader server: started successfully at port $(PORT); exit 0) \
    || (echo WiTTReader server: starting server failed.; exit 8)
```

Es hat wiederum verschiedene Voraussetzungen, genauer gesagt

- **check**, testet auf die Verfügbarkeit benötigter Software
- **build**, installiert client- und serverseitige Abhängigkeiten über *npm*[b] und *bower*[c]
- **start**, startet den Server nach erfolgreicher Abarbeitung der beiden vorhergehenden Aufgaben

In diesem Kontext besteht die Möglichkeit, den Port für die Applikation sowie den Port, auf dem die Datenbank läuft, zu definieren. In diesem Fall sähe der Aufruf dann so aus:

```
1 make PORT=<application port> MONGOPORT=<database port> deploy
```

6.2 Die Views

Das letztendliche Erscheinungsbild der Applikation ist über unterschiedliche Views geregelt, denen jeweils ein Jade-Template zugrunde liegt und die über entsprechende Routes im Server gerendered werden. Dabei sind die zentralsten die Startseite und der tatsächliche Faksimileviewer, welche über GET-Requests an

```
1 app.get('/',function(req,res){
2   res.render('index',{title:"Quadro Reader"});
3 });
```

und

```
1 app.get('/reader',function(req,res){
2   res.render('reader');
3 });
```

in HTML konvertiert und dargestellt werden.

6.3 Der Client

Nach erfolgreichem Laden der View werden auf Clientseite verschiedene Dinge ausgeführt. Was die Startseite angeht (festgelegt in *public/javascripts/mainppage.js*), handelt es sich dabei nur darum, dass beim Auslösen des *mousedown*-Events auf den Dokumentencovern - zu sehen in Abbildung 6.1 - ein POST-Request mit der jeweiligen Dokument-ID und der aktuell eingestellten Auflösung an den Server geschickt wird.

[b] kurz für Node Package Manager, Verwaltungssystem für Node-Module
[c] Paketmanager für clientseitige Javascript-Libraries

Abbildung 6.1: Startseite des Faksimileviewers

Im Falle des Readers wird zuerst der vorher genannte eindeutige Bezeichner aus der URL entnommen, siehe Snippet 6.7. Hierbei befindet sich die ID am Ende der URL, nach dem Zeichen #. Beispielsweise wäre in der URL

http://localhost:3000/reader#das-ist-die-datenid

der Teil **das-ist-die-datenid** der Bezeichner für die Daten.

```
67 this.setDataId = function(){
68   dataId = window.location.hash === "" ? "demo" : window.location.
       hash.substring(1);
69   window.location.hash = "";
70 }
```

Listing 6.7: Definition der ID zu den zugehörigen Daten

Nach diesem Schritt wird eine kurze Anleitung angezeigt, falls das nicht bereits vom Benutzer deaktiviert wurde. Im Anschluss werden über einen GET-Request mit der ID als Parameter die Daten, welche auf Serverseite abgelegt wurden, abgefragt. Dies ist im Snippet 6.8 im Quelltext dargestellt, das aus der Datei *public/javascripts/wittReader.js* entnommen ist.

```
20 $(function(){
21   WiTTReader.setDataId();
22   if(localStorage.getItem("hideNextStart") === null){
23     WiTTReader.showHelpBox();
24   }
25   WiTTReader.setReaderObject();
26   WiTTReader.getVariables();
```

```
27 });
```

Listing 6.8: Funktionalität nach erfolgreichem Laden

Falls ein Problem beim Laden des Seitenindex aufgetreten ist, wird eine entsprechende Fehlermeldung angezeigt. Im Erfolgsfall werden die zurückgelieferten Daten in die internen Variablen gespeichert und die Initialisierung gestartet. Diese lässt sich in vier Phasen unterteilen:

1. Anwendung von CSS-technischen Anpassungen (Funktion *applyLayoutStyles*)
2. Initialisierung der turn.js-Library, im Quellcode dargestellt in Snippet 6.9
3. Erstmalige Anwendung der Funktion *resize* zur Anpassung des Layouts an die Dimensionen des Browserfensters
4. Verknüpfung von HTML-Elementen mit der entsprechenden Funktionalität (Funktionen *applyButtonBindings*, *applyKeyBindings*, *applyFullscreenBindings* und *applyChangeBindings*)

```
134 readerObject.turn({
135   display: "double",
136   pages: pages,
137   page: siglumIndex[startSiglum],
138   acceleration: true,
139   gradients: !$.isTouch,
140   elevation: 50,
141   height: height,
142   corners: {
143     backward: ['bl'],
144     forward: ['br'],
145     all: ['bl', 'br']
146   },
147   when:{
148     turning: function(...),
149     turned: function(...)
150   }
151 });
```

Listing 6.9: Initialisierung von turn.js

Besonders wichtig sind die Funktionen, die beim Auslösen der *turnjs*-Events *turning* und *turned* aufgerufen werden und die Blätterfunktion erlauben. Dabei enthält das *turn.js*-Objekt alle Seiten von 1 bis n, die dem Reader hinzugefügt wurden. Für den Fall, dass eine Seitennummer noch nicht enthalten ist, wird sie über die Funktion **addPage** - abgebildet in Snippet 6.10 - neu hinzugefügt. Zu diesem Zweck wird ein neues DIV-Element mit bestimmten Klassen, einer ID sowie einer Sanduhr als einzigem Inhalt angelegt. Nach einer gewissen Ladezeit wird dann die

Sanduhr entfernt und mithilfe der Grafiklibrary *Raphaël.js* ein neuer Canvas erzeugt, auf dem das Faksimile, die Zoombuttons und eventuell die Highlighting-Boxen abgelegt werden. Dieser Vorgang wird für jede Seite im aktuellen Scope, d.h. die zwei sichtbaren Seiten sowie jeweils zwei Seiten links und rechts, durchgeführt.

```
338   function addPage(page, reader){
339     var facsimilePage = $("<div />", {"class": "page "+((page%2==0) ?
      "odd" : "even"), "id": "page-"+page}).html('<i class="loader"></i
     >');
340     reader.turn("addPage", facsimilePage, page);
341     setTimeout(function(){
342       facsimilePage.html("");
343       facsimilePage.attr("documentid",documentId);
344       facsimilePage.attr("density",facsimileDensity);
345       var paper = Raphael(facsimilePage.attr("id"));
346       papers[page] = paper;
347       paper.setViewBox(0,0,width,height,true);
348       paper.setSize('100%','100%');
349       paper.image("/facsimile?documentId="+documentId+"&density="+
     facsimileDensity+"&page="+pageIndex[page]+"&width="+width+"&height
     ="+height,0,0,width,height);
350       var pageHits = hits[pageIndex[page]];
351       if(typeof(pageHits) != "undefined"){
352         insertHighlighting(paper,page,pageHits);
353       }
354       insertZoomButtons(facsimilePage,page);
355     }, 100);
356   }
```

Listing 6.10: Hinzufügen von Seiten zum Reader

Für die Funktionalitäten des Dokument- und Auflösungswechsels sowie die Zoommöglichkeit - genaueres dazu in Kapitel 10 - ist diese Vorgehensweise auch entscheidend.

Um nun im aktuellen Dokument zu blättern, gibt es zwei verschiedene Möglichkeiten. Zum einen können die Pfeilbuttons, welche auf der linken und rechten Seite des Readers zu finden sind, verwendet werden. Zum anderen besteht die Option, über Drücken der Tasten a und d auf der Tastatur zur vorherigen bzw. nächsten Seite zu springen. Daneben kann der Benutzer der Software ebenso durch alle Treffer hindurch blättern - dies funktioniert auf der einen Seite über die entsprechenden Buttons in der Menüleiste, auf der anderen Seite wiederum über die Tastatur mittels der Buchstaben p für den vorhergehenden und n für den nächsten Treffer.

Kapitel 7

Highlighting im Faksimile

Dieses Kapitel beschreibt die benötigten Einzelschritte, um ein Highlighting der Suchmaschinen-Treffer in der Weboberfläche zu ermöglichen. Das reicht vom Ablegen der notwendigen Informationen in der Datenbank über das Erstellen einer Treffersammlung bis hin zur tatsächlichen Anzeige dieser Daten innerhalb der Faksimile-Ansicht.

7.1 Speicherung der Koordinaten in der Datenbank

Das Herzstück des Highlightings stellt die Datenbank dar, in welcher die Koordinaten zusammen mit weiteren Informationen abgelegt werden. Bevor diese eingepflegt werden, befinden sie sich in den Indexfiles, die auch den Seitenindex enthalten und im Verzeichnis *data* untergebracht sind. Dabei existieren für vier Dokumente - die Manuskripte Ms-114 und Ms-115 sowie die Typoskripte Ts-213 und Ts-310 - Koordinaten auf Bemerkungsebene, für das Ts-213 gibt es darüber hinaus auch Satzkoordinaten.

7.1.1 Bemerkungskoordinaten

Die Struktur der Bemerkungskoordinaten, welche in der jeweiligen JSON-Datei unter dem Haupt-Key *coordinates* stehen, sieht wie in Snippet 7.1 dargestellt aus. Dabei stellen die Kommazahlen für z.B. *left* eine Angabe dar, ab welcher Prozentzahl der kompletten Breite des Faksimile die Bemerkung beginnt. Über die Information *page* lässt sich der Datensatz einer Seiten-ID zuordnen - *type* gibt wiederum an, dass es sich um Koordinaten für eine Bemerkung handelt, so lässt es sich von anderen Datensätzen mit Satzkoordinaten unterscheiden.

```
1 {
2   "coordinates" : {
3     "height" : 0.14,
4     "left" : 0.02,
5     "top" : 0.07,
```

```
6       "width" : 0.75
7     },
8     "page" : "Ms-114,100r",
9     "siglum" : "Ms-114,100r[1]",
10    "type" : "remark"
11  }
```

Listing 7.1: Beispiel für Bemerkungskoordinaten

7.1.2 Satzkoordinaten

Bei den Koordinaten auf Satzebene gestaltet sich der Aufbau im Grunde sehr ähnlich, es kommt eine laufende Satznummer hinzu und das Attribut *type* unterscheidet sich dementsprechend.

```
1   {
2     "coordinates" : {
3       "height" : 0.04,
4       "left" : 0.35,
5       "top" : 0.04,
6       "width" : 0.25
7     },
8     "page" : "Ts-213,i-r",
9     "sentID" : 1,
10    "siglum" : "Ts-213,i-r[1]",
11    "type" : "sentence"
12  }
```

Listing 7.2: Beispiel für Satzkoordinaten

7.1.3 Vorgehen

Um die Koordinaten letztlich in die Datenbank einzupflegen, enthält das Makefile ein Target namens **coordinates-insert**. Dieses startet das Perl-Skript *saveCoordinatesToDB.perl* im Unterordner *tools* mit folgenden Argumenten:

- dem Datenverzeichnis, in dem die Indexfiles mit den Koordinaten liegen (in der Regel *data*)
- einer Liste aller Dokument-IDs

Mittels

```
18 my $client = MongoDB::MongoClient->new(host => 'localhost', port =>
       27017);
19 my $database = $client->get_database('wittreader');
```

wird zuerst die Verbindung zur Datenbank hergestellt. Danach werden für jedes Dokument die Koordinaten extrahiert, die passende Collection mit dem Namen *-coordinates* erzeugt und über die Sammlung der Bemerkungskoordinaten iteriert. Für jeden Datensatz wird mit

```
42 $entry = $collection->find_one({"siglum" => $_->{"siglum"}});
43 if(!$entry){
44   $collection->insert($_);
45   say LOGGING "INSERT new siglum ".$_->{"siglum"};
46 }else{
47   if(!Compare($_->{"coordinates"},$entry->{"coordinates"})){
48     $collection->update({"siglum" => $_->{"siglum"}},{'$set' => {"
       coordinates" => $_->{"coordinates"}}});
49     say LOGGING "UPDATE siglum ".$_->{"siglum"};
50   }else{
51     say LOGGING "NOTHING TO BE DONE for siglum ".$_->{"siglum"};
52   }
53 }
```

geprüft, ob mit dem aktuellen Siglum bereits ein Eintrag existiert. Ist dies nicht der Fall, wird er angelegt. Falls ja, wird auf Unterschiede zwischen den Koordinaten untersucht und der Eintrag entweder aktualisiert oder übersprungen. Für jeden Fall wird zudem eine entsprechende Meldung in ein Logfile geschrieben.

7.2 Ermittlung der Treffer von WiTTFind

Alle Treffer, die bei einer Anfrage von der Suchmaschine WiTTFind gefunden werden, werden in einem XML-File mit einem Namen, der sich aus der IP des Benutzers, einem Zeitstempel sowie dem Suffix client*out* zusammensetzt, gespeichert. Der Aufbau der Einzeltreffer in dieser Datei sieht folgendermaßen aus:

```
1 <hit path="/srv/www/vhosts/wittfind/htdocs/data/Ts-213_OA_NORM-tagged
      .xml" n="Ts-213,iii-r[8]_1" f="Ts-213,iii-r" abnr="54" satznr="128
      " no="0">
2   <match pos="3" id="10383" tag="VVFIN" token="denkt" gc="+V+refl+tr"
       ic=":2mGi:3eGi" />
3 </hit>
```

Zum Sammeln der Hits wird die Funktion **getHits** in Snippet 7.3 verwendet. Diese extrahiert aus dem HTML-Output der Suchmaschine den Pfad zum Hitsfile und iteriert über alle *hit*-Tags. In jedem Durchlauf werden zuerst das Siglum und die Dokumenten-ID ermittelt, was durch passendes Splitten der XML-Attribute *n* und *path* geschieht. Sollte ein Siglum bereits über einen anderen Treffer bearbeitet worden sein, wird es ignoriert. Im gegenteiligen Fall wird das Siglum am speziellen Stichwort *et*[a] aufgespalten. Für jedes der resultierenden Einzelsiglen wird nun noch die zugehörige Seiten-ID erstellt und die Kollektion entsprechend erweitert.

```
1  var hitsFile = $(".bemerkungen:first").attr("hitsfile");
2  $.get(hitsFile, function(data){
3    var siglen = [];
4    var siglum;
5    var siglumParts;
6    var page;
7    var docId;
8    var docIdMatcher;
9    hitCounter = {};
10   hits = {};
11   $(data).find("hit").each(function(){
12     siglum = $(this).attr("n").split(/_/)[0];
13
14     /*
15      * Special case: Ms-140,39v would be Ms-140 with .split(/,/)[0]
16      * Instead, use path: /mnt/wittfind-web/data/Ms-140,39v_OA_NORM-
    tagged.xml
17      * 1. Split on _ -> /mnt/wittfind-web/data/Ms-140,39v
18      * 2. Split on / + pop() -> Ms-140,39v
19      */
20     docId = $(this).attr("path").split(/\//).pop().split(/_/)[0];
21     docIdMatcher = new RegExp("^"+docId);
22     if(!hits[docId]){
23       hits[docId] = {};
24     }
25     if(!hitCounter[docId]){
26       hitCounter[docId] = 0;
27     }
28     hitCounter[docId]++;
29     if($.inArray(siglum, siglen) == -1){
30       siglen.push(siglum);
31       siglumParts = siglum.split(/et/);
32       for(var i=0; i<siglumParts.length; i++){
33         if(!siglumParts[i].match(docIdMatcher)){
34           siglumParts[i] = docId+","+siglumParts[i];
```

[a] über das Stichwort *et* werden Bemerkungen markiert, die sich über mehrere Seiten erstrecken

```
35          }
36          page = siglumParts[i].split(/\[/)[0];
37          if(!hits[docId][page]){
38            hits[docId][page] = [];
39          }
40          hits[docId][page].push(siglumParts[i]);
41          hits[docId][page] = $.unique(hits[docId][page]);
42          hits[docId][page].sort();
43        }
44      }
45    });
46  })
```

Listing 7.3: Erstellung der Treffer-Sammlung

Angenommen die XML-Datei enthält folgende Treffer:

```
1  <hit path="/srv/www/vhosts/wittfind/htdocs/data/Ts-213_OA_NORM-tagged
       .xml" n="Ts-213,iii-r[8]_1" f="Ts-213,iii-r" abnr="54" satznr="128
       " no="0">
2    <match pos="3" id="10383" tag="VVFIN" token="denkt" gc="+V+refl+tr"
       ic=":2mGi:3eGi" />
3  </hit>
4  <hit path="/srv/www/vhosts/wittfind/htdocs/data/Ts-213_OA_NORM-tagged
       .xml" n="Ts-213,6a-r[5]et7r[1]_1" f="Ts-213,6a-r" abnr="189"
       satznr="460" no="1">
5    <match pos="3" id="10383" tag="VVFIN" token="denkt" gc="+V+refl+tr"
       ic=":2mGi:3eGi" />
6  </hit>
7  <hit path="/srv/www/vhosts/wittfind/htdocs/data/Ms-114_OA_NORM-tagged
       .xml" n="Ms-114,7r[1]_1" f="Ms-114,7r" abnr="7" satznr="90" no="2"
       >
8    <match pos="3" id="10383" tag="VVFIN" token="denkt" gc="+V+refl+tr"
       ic=":2mGi:3eGi" />
9  </hit>
```

Dann sähe die Treffer-Sammlung so aus:

```
1  {
2      "Ts-213": {
3          "Ts-213,iii-r": [
4              "Ts-213,iii-r[8]"
5          ],
6          "Ts-213,6a-r": [
```

```
7                "Ts-213,6a-r[5]"
8            ],
9            "Ts-213,7r": [
10               "Ts-213,7r[1]"
11           ]
12       },
13       "Ms-114": {
14           "Ms-114,7rr": [
15               "Ms-114,7r[1]"
16           ]
17       }
18   }
```

7.3 Darstellung im Browser

Um die Treffer final darzustellen, wird für jede neu eingefügte Seite im Reader getestet, ob es auf dieser Hits gibt. Falls ja, werden sie mit der Funktion **insertHighlighting** in Snippet 7.4 in den Canvas der Seite eingefügt. Zuerst werden dazu alle Koordinaten für die aktuelle Seite über die im Server definierte Route */coordinates* abgefragt und darüber iteriert. Für jedes Siglum wird gecheckt, ob es auch in den Treffern vorkommt. Ist dies der Fall, wird mit den jeweiligen Dimensionen ein Rechteck auf den Canvas gelegt, welches über die CSS-Klasse *highlightingHit* eine Farbe sowie eine Durchsichtigkeit erhält. Falls es sich um ein Siglum handelt, das nicht in der Treffersammlung existiert, wird ebenso eine Box - hier mit der Klasse highlihtingNoHit - eingefügt, die im Gegensatz zu den Treffern vorerst unsichtbar ist. Erst beim Überfahren mit der Maus wird sie sichtbar.

Beim Wert für *left* wird zudem noch ein Overlapping-Faktor aufaddiert, falls es sich um eine rechte Seite handelt - bei Typoskripten ist dieser *gleich 0*, bei Manuskripten ist er in der Regel *größer 0*, da die zugehörigen Faksimile aus Doppelseiten entstanden sind und nicht in der Mitte, sondern mit einem gewissen Spielraum, den dieser Faktor angibt, zerschnitten wurden. Daneben erhält jede solche Box Informationen dazu, zu welchem Dokument sie gehört und welches Siglum sie repräsentiert. Sehr wichtig ist hierbei auch die Definition derjenigen Aktion, die ausgeführt wird, sobald der Benutzer auf die Box klickt. Dies beleuchtet Kapitel 8 näher.

```
1  function insertHighlighting(paper,page,pageHits){
2    $.get("/coordinates?documentId="+documentId+"&page="+pageIndex[page
     ], function(pageCoordinates){
3        $.each(pageCoordinates, function(index,coordinates){
4          var x = Math.round((coordinates.coordinates.left+((page%2)
     ==1?originRectoX:0))*width);
5          var y = Math.round(coordinates.coordinates.top*height);
6          var w = Math.round(coordinates.coordinates.width*width);
```

```
7           var h = Math.round(coordinates.coordinates.height*height);
8           var highlightingBox = paper.rect(x,y,w,h).click(function(){
    WiTTReader.showTranscription(this); });
9           if($.inArray(coordinates.siglum,hits[pageIndex[page]]) != -1)
    {
10            highlightingBox.node.setAttribute("class","highlightingHit"
    );
11          }else{
12            highlightingBox.node.setAttribute("class","
    highlightingNoHit");
13          }
14          highlightingBox.node.setAttribute("documentid",documentId);
15          highlightingBox.node.setAttribute("siglum",coordinates.siglum
    );
16          highlightingBox.node.setAttribute("page",page);
17        });
18    });
19  }
```

Listing 7.4: Einfügen der Treffer in die Faksimile-Ansicht

Im Browser sieht das Endergebnis dann so aus:

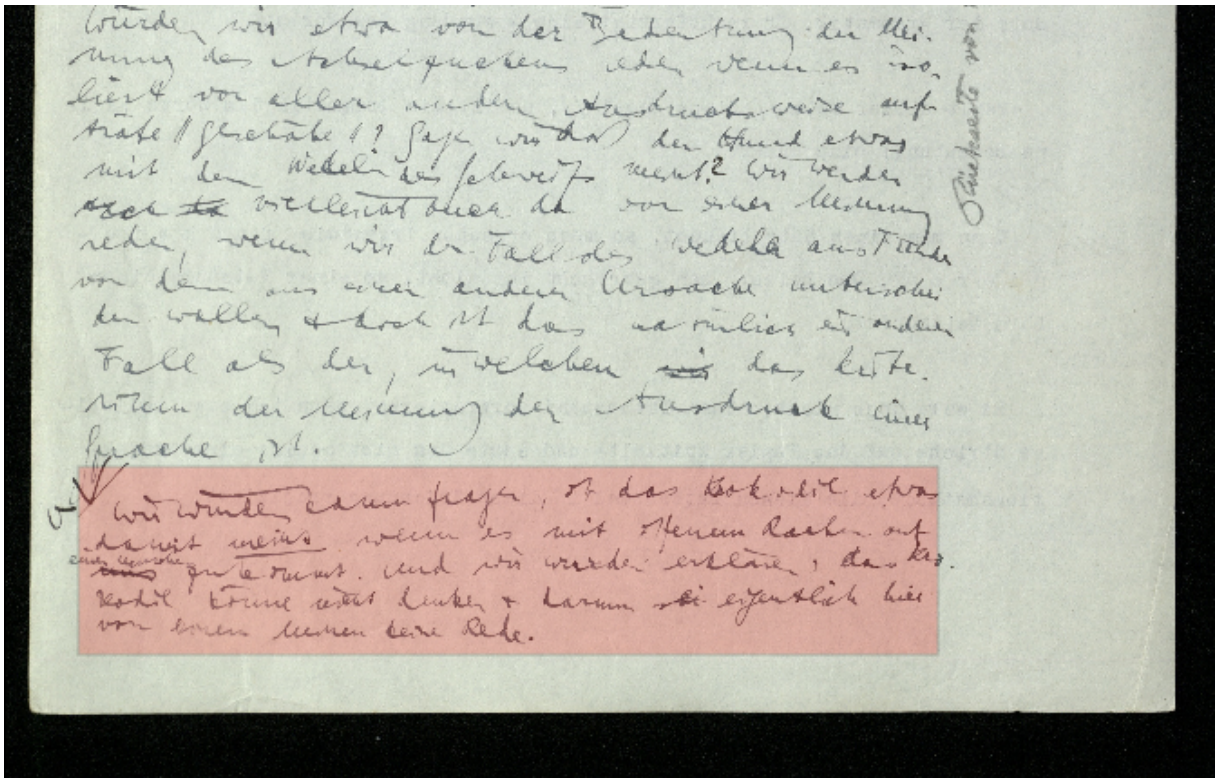

Abbildung 7.1: Highlighting im Faksimile

Kapitel 8

Darstellung des XML-Codes

8.1 Extraktion der Editionsfragmente

Um die unterschiedlichen Transkriptionen auf Bemerkungsebene - in Snippet 8.1 ist ein Beispiel dafür dargestellt - im Browser darstellen zu können, müssen die XML-Dateien geparsed und die Fragmente daraus extrahiert werden.

```
1 <ab n="Ts-213,i-r[2]" ana="abnr:2">
2   <s n="Ts-213,i-r[2]_1" ana="facs:Ts-213,i-r abnr:2 satznr:2">1)Das
      Verstehen, die Meinung, fällt aus unsrer Betrachtung heraus.</s>
3   <s n="Ts-213,i-r[2]_2" ana="facs:Ts-213,i-r abnr:2 satznr:3">(<abbr
      type="abb">S.</abbr>1)</s>
4 </ab>
```

Listing 8.1: Beispiel für eine Bemerkung im XML-Code

Für diese Aufgabe ist im Makefile das Target **transcription-insert** definiert, welches das Perl-Skript *saveTranscriptionToDB.perl* - zu finden im Unterordner *tools* - mit den Dateipfaden der normalisierten und diplomatischen XML-Files als Argument startet.

Dieses durchläuft alle übergebenen Argumente und stellt durch

```
23 my $isNormFile = 1;
24 if(/DIPL/){
25   $isNormFile = 0;
26 }
```

zuerst fest, ob es sich beim aktuellen Dokument um ein normalisiertes oder ein diplomatisches handelt. Das ist eminent wichtig, da in beiden Fällen unterschiedliche Dinge zu erledigen

sind. Im Anschluss wird ein XML-Twig[a] erstellt, siehe Snippet 8.2, und über den Parameter *twig_handlers* festgelegt, was geschehen soll, sobald der Parser auf einen *ab*-Tag stößt, welcher eine Bemerkung repräsentiert. Zum Abschluss wird das Parsing der aktuellen Datei gestartet.

```
31 $xmlTwig = XML::Twig->new(
32   twig_handlers => {
33     ab => sub { if($isNormFile){ handleNormRemarks(@_,$collection); }
     else{ handleDiploRemarks(@_,$collection); } }
34   },
35   pretty_print => 'indented'
36 );
37 $xmlTwig->parsefile($_);
```

Listing 8.2: Erstellung des XML-Twig

8.2 Speicherung in der Datenbank

Zum Speichern der Fragmente des XML-Codes wird in einem ersten Schritt, wie in Snippet 7.1.3 dargestellt, eine Verbindung zur Datenbank hergestellt und die zum Dokument gehörende Collection ausgewählt. Danach wird durch das Parsen für jeden *ab*-Tag die entsprechende Funktion aufgerufen, welche das Tag-Element sowie die vorher gewählte Collection[b] als Parameter erhält.

Im Falle eines normalisierten Files handelt es sich dabei um die Funktion **handleNormRemarks** (zu sehen in Snippet 8.3). Zuerst wird ein Datensatz für das aktuelle Element angelegt und alle aus dem *n*-Attribut ausgelesenen Einzelsiglen - bei einer Bemerkung über mehrere Seiten gibt es mehrere durch *et* verknüpfte - in einer Variablen gespeichert. In der Folge wird geprüft, ob es für die Sammlung von Siglen, welche momentan bearbeitet werden, bereits einen Eintrag in der Datenbank gibt. Ist dies der Fall, wird das Element schlicht übersprungen. Falls nicht, werden zuerst alle Einzelsiglen unter dem Key *siglen* im Datensatz gespeichert. Dann wird der XML-Code des Elements extrahiert, in einem Preprocessing-Schritt alle führenden Zeilenumbrüche entfernt sowie die Einrückung angepasst und an Newline gesplittet unter dem Key *norm* abgelegt. Zuletzt wird der Datensatz in die Datenbank eingefügt.

```
41 sub handleNormRemarks{
42   my($twig,$remark,$collection) = @_;
43   my $siglumData = {};
44
45   my $siglum = $remark->{'att'}->{'n'};
46   my @siglumParts = split(/,|et/,$siglum);
47   my $docId = shift(@siglumParts);
48
```

[a] siehe Modul-Dokumentation auf CPAN

[b] der Name der Collection setzt sich aus der Dokument-ID und dem Suffix "-transcription" zusammen

```
49    my $entry = $collection->find_one({"siglen" => $docId.",".
       $siglumParts[0]});
50    if(!$entry){
51      $siglumData->{"siglen"} = [];
52      foreach(@siglumParts){
53        push(@{$siglumData->{"siglen"}},$docId.",".$_);
54      }
55
56      my $normXml = $remark->sprint;
57      $normXml =~ s/^\n//;
58      $normXml =~ s/^(\s+)//;
59      my $initialSpaces = $1;
60      $normXml =~ s/(\n)$initialSpaces/$1/g;
61      @{$siglumData->{'norm'}} = split(/\n/,$normXml);
62
63      $collection->insert($siglumData);
64    }
65  }
```

Listing 8.3: Handling der Bemerkungen einer normalisierten Datei

Für diplomatische Dateien sieht das Vorgehen etwas anders aus - dafür wird die Funktion **handleDiploRemarks** verwendet, die in Snippet 8.4 dargestellt ist. Sie extrahiert ebenso zu Anfang die Einzelsiglen, danach folgen dieselben Vorverarbeitungsschritte. Am Schluss wird auf der Collection eine Update-Operation ausgeführt, die den Datensatz, der zur Siglengruppe passt, um einen Key *diplo* erweitert und dort den XML-Code ablegt.

```
67  sub handleDiploRemarks{
68    my($twig,$remark,$collection) = @_;
69
70    my $siglum = $remark->{'att'}->{'n'};
71    my @siglumParts = split(/,|et/,$siglum);
72    my $docId = shift(@siglumParts);
73
74    my $diploXml = $remark->sprint;
75    $diploXml =~ s/^\n//;
76    $diploXml =~ s/^(\s+)//;
77    my $initialSpaces = $1;
78    $diploXml =~ s/(\n)$initialSpaces/$1/g;
79    my @diploXmlLines = split(/\n/,$diploXml);
80
81    $collection->update({"siglen" => $docId.",".$siglumParts[0]},{'$set
       ' => {"diplo" => \@diploXmlLines}});
82  }
```

Listing 8.4: Handling der Bemerkungen einer diplomatischen Datei

8.3 Darstellung im Browser

Im Browser dargestellt werden die XML-Fragmente, sobald der Benutzer in der Reader-Ansicht auf eine der Highlighting-Boxen klickt. Daraufhin wird die Funktion **showTranscription** aufgerufen, welche alle benötigten Parameter aus den Eigenschaften der Box herausliest und mit diesen einen GET-Request an die Route **transcriptionViewer** auf dem Server schickt. Die Antwort wird in eine *div*-Box in der HTML-Struktur des Readers geladen und angezeigt.

Auf dem Server ist die Route folgendermaßen angelegt:

```
180 app.get('/transcriptionViewer',function(req,res){
181   var db = req.db;
182   var documentId = req.query.documentId;
183   var siglum = req.query.siglum;
184   var page = req.query.page;
185   var collection = db.collection(documentId+"-transcription");
186   var normTranscription;
187   var diploTranscription;
188   collection.findOne({"siglen": siglum}, {"norm": 1, "diplo":1, "_id"
       : 0},function(e,transcription){
189     normTranscription = transcription && transcription.norm ?
      transcription.norm.join("\n") : "No transcription found: norm";
190     diploTranscription = transcription && transcription.diplo ?
      transcription.diplo.join("\n") : "No transcription found: diplo";
191     res.render('transcriptionViewer',{"documentId":documentId,"
      density":req.query.density,"siglum":siglum,"page":page,
192                                       "normTranscription":
      normTranscription,"diploTranscription":diploTranscription});
193   });
194 });
```

Dazu werden am Anfang die übergebenen Parameter ausgelesen sowie in Variablen gespeichert und danach in der Datenbank diejenige Collection ausgewählt, die zum angeforderten Dokument gehört. In dieser wird in der Folge der Datensatz gesucht, dessen Siglen-Array das benötigte Siglum enthält, und die jeweiligen normalilisierten sowie diplomatischen Fragmente werden zurückgeliefert. Die Ergebnisse werden dann über den Rendering-Befehl an die zugehörige View weitergereicht und mittels

```
23 pre#norm.tab-pane.active(role='tabpanel')
24   code
25     |#{normTranscription}
26 pre#diplo.tab-pane(role='tabpanel')
27   code
28     |#{diploTranscription}
```

in die entstehende HTML-Struktur als Code-Block eingebaut. Dabei sind die unterschiedlichen Formate über Buttons auswählbar.

Zusätzlich zum XML-Code wird außerdem noch das jeweilige Faksimile-Extrakt mit

```
23 img(class='facsimileExtract' src='/facsimileExtractor?documentId=' +
     documentId + '&density=' + density + "&siglum=" + siglum + "&page=
     " + page)
```

eingefügt. Diese Funktionalität wird in Kapitel 10 näher beschrieben.

Das Endergebnis gestaltet sich dann so, wie in Abbildung 8.1 gezeigt.

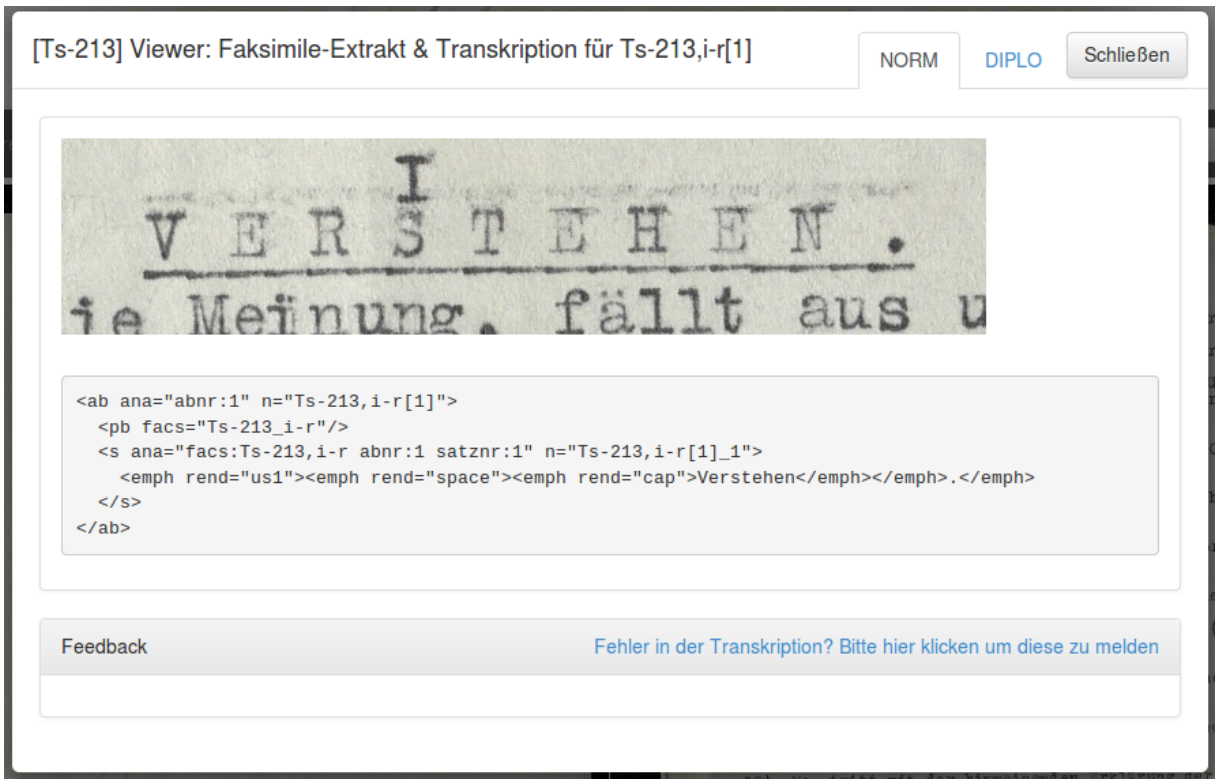

Abbildung 8.1: Darstellung der Transkription

Kapitel 9

Integration einer Feedback-Funktion

Da bei der Erstellung einer Edition von Zeit zu Zeit auch Transkriptionsfehler auftreten, ist in der Ansicht, welche in Abbildung 8.1 dargestellt ist, zusätzlich eine Feedbackfunktion enthalten. Diese ermöglicht es dem Benutzer, die vorrangig Philosophen sein werden, Fehler im XML-Code zu melden. Diese Funktionalität ist so implementiert, dass beim Abschicken automatisch ein Issue im CIS-Gitlab erstellt und dem zuständigen Maintainer zugewiesen wird.

Sobald der User einen Fehler entdeckt und auf den Link geklickt hat, um diesen zu melden, wird ein Formular - zu sehen in Abbildung 9.1 - eingeblendet. In dieses muss eine Email-Adresse und eine Fehlerbeschreibung eingetragen werden. Zum Schutz vor Bots ist zudem vorgesehen, dass eine zufällig erstellte Figur nachgezeichnet werden muss, bevor das Formular abgeschickt werden kann - dafür wird die Javascript-Library *MotionCAPTCHA*[a] verwendet.

Beim Klick auf den Submit-Button wird dann ein POST-Request an der Server geschickt, der die Interaktion mit Gitlab abwickelt - bei erfolgreichem Erstellen der Meldung wird im Anschluss eine dementsprechende Meldung angezeigt, bei einem Fehler wird auch dies gemeldet.

Auf Seite des Servers wird zur Verbindung das Modul *gitlab* verwendet, das mit Hilfe von

```
273 var gitlab = require('gitlab')({
274   url:    'https://gitlab.cis.uni-muenchen.de',
275   token: 'foobar'
276 });
```

eine Verbindung zum CIS-Gitlab herstellt. Der Parameter *token* steht hierbei für eine eindeutige ID des Benutzers, der das Issue anlegt.

Nachdem alle benötigten Informationen für das Issue, im Einzelnen

- der Titel,

[a] Informationen zu finden auf GitHub

Abbildung 9.1: Formular für Feedback zur Transkription

- die Beschreibung,
- die ID der Person, der das Issue zugeordnet werden soll und
- die Labels,

zusammengetragen wurden, wird über den Aufruf von

```
278 gitlab.issues.create(123,{title:shortdescription,assignee_id:
      assigneeId,description:description,labels:labels.toString()},
      function(issueStatus){
279   if(typeof issueStatus === "object"){
280     res.send({"status":"success"});
281   }else{
282     res.send({"status":"error"});
283   }
284 });
```

ein Request an das Gitlab gesendet, um das Issue zu erstellen. Dabei repräsentiert der erste Parameter die ID des Projekts, in dem das Issue angelegt werden soll. Zum Abschluss wird ein entsprechender Status zurückgesendet.

Kapitel 10

Weitere Funktionalitäten

Dieses Kapitel erläutert einige weitere kleine Features, die der Reader bietet. Diese sind größtenteils über die Menüleiste, die in Abbildung 10.1 dargestellt ist, zu verwenden.

Abbildung 10.1: Die Menüleiste des Readers

10.1 Anzeige einer Benutzeranleitung

Sobald die Faksimile-Ansicht gestartet wurde, wird für den Benutzer eine kurze Anleitung eingeblendet. Dort ist erklärt, wie das Blättern durch die Seiten und die Treffer über Tastaturkommandos funktioniert. Des Weiteren besteht die Möglichkeit, die Anzeige der Anleitung für den nächsten Start über eine Checkbox zu deaktivieren. Für diesen Zweck kommt das Konzept des *localStorage* zum Einsatz. Die Box lässt sich aber auch problemlos wieder standardmäßig aktivieren, indem sie über einen Klick auf den Button *Hilfe* in der Menüleiste geöffnet und in der Ansicht ein entsprechendes Häkchen gesetzt wird.

10.2 Wechsel des Dokuments

Eine weitere Funktion, die dem User zur Verfügung steht, ist der Wechsel des Dokuments. Genauso wie die Suchmaschine mehrere Dokumente durchsucht, ist es auch möglich, für alle diese die zugehörigen Faksimile zu betrachten. Um einen Wechsel vorzunehmen, muss nur in der Liste der Dokumente, welche sich auch in der Menüleiste befindet, ein anderes ausgewählt werden. Danach wird über die Funktion **handleDocumentChange** ein GET-Request an die Route **documentChange** gesendet und als Parameter die ID der Daten sowie die ID des gewählten Dokuments mitgeschickt.

Auf dem Server werden in der Folge die entsprechenden Daten und deren Treffer-Sammlung geladen. Falls es keine Treffer geben sollte, bleibt die Sammlung schlicht leer. Zuletzt werden die ermittelten Treffer, der passende Seitenindex sowie der Overlapping-Faktor übermittelt.

Nach erfolgreichem Abschluss des Request werden die internen Variablen mit den neuen Werten befüllt und dann durch

```
327 var newPage = keys[0] || 2;
328 if(newPage == readerObject.turn("page")){
329   readerObject.trigger("turning");
330 }else{
331   readerObject.turn("page",newPage);
332 }
333 readerObject.turn("pages", pages);
```

auf die Seite, auf der der erste Treffer zu finden ist, geblättert und das Kontingent an Seiten angepasst. Danach wird das in Kapitel 6.3 bereits genannte Event *turning* relevant. Dieses wird folglich ausgelöst und über die Ausführung von

```
166 if(currentPage.attr("documentid") != documentId){
167   $("svg image",currentPage)
168     .attr("href","/facsimile?documentId="+documentId+"&density="+
      facsimileDensity+"&page="+pageIndex[page]+"&width="+width+"&height
      ="+height);
169   $(".highlighting[documentId!='"+documentId+"']",currentPage).hide()
      ;
170   var pageHits = hits[pageIndex[page]];
171   if(typeof(pageHits) != "undefined"){
172     if($(".highlighting[documentid='"+documentId+"']",currentPage).
      length > 0){
173       $(".highlighting[documentid='"+documentId+"']",currentPage).
      show();
174     }else{
175       insertHighlighting(papers[page],page,pageHits);
176     }
177   }
178   currentPage.attr("documentId",documentId);
179 }
```

der Link zu den Faksimile auf das neue Dokument angepasst. Zudem werden diejenigen Highlighting-Boxen, die nicht zur neuen Dokument-ID passen, unsichtbar gemacht und eventuelle neue Boxen entweder neu angelegt oder wieder angezeigt.

10.3 Wechsel der Faksimileauflösung

Die Auflösung der Faksimile kann über einen Klick auf den Text, der die aktuelle Einstellung anzeigt, zwischen *HD good quality* und *HD medium quality* hin- und hergewechselt werden. Dazu wird die Funktion **updateDensity** benutzt, die einerseits den Text ändert und andererseits wiederum das *turning*-Event anstößt. Auch hier wird der Pfad zum Faksimile entsprechend angepasst.

10.4 Zoom ins Faksimile

Um die Faksimile besonders genau zu untersuchen und zu studieren, ist im Reader neben vielen anderen Möglichkeiten auch die Option implementiert, ins Bild hinein zu zoomen. Zur konkreten Realisierung wurde auf die jQuery-Bibliothek *jQuery.panzoom*[a] zurückgegriffen. Damit die Zoom-Funktion vom Benutzer verwendet werden kann, sind auf jeder Seite des Readers mit Hilfe der Funktion **insertZoomButtons** - abgebildet in Snippet 10.1 - drei unterschiedliche Buttons eingebunden, die verschiedene Aktionen der Bibliothek auslösen und in Tabelle 10.1 dargestellt sind.

```
399 function insertZoomButtons(facsimilePage,page){
400   var pageId = facsimilePage.attr("id");
401   var zoomButton;
402   $.each({"In":["+","zoom",false],"Out":["-","zoom",true],"Reset":["
      ","resetZoom",true]}, function(key,value){
403     zoomButton = $("<button />",{"class":"btn btn-default zoomButton
     zoomButton"+((page%2==0)?"Left":"Right"),"page":pageId});
404     zoomButton.addClass("zoom"+key+"Button").html(value[0]).attr({"f"
     :value[1],"o":value[2]});
405     zoomButton.attr("onclick",'$($(this).siblings()[0]).panzoom($(
     this).attr("f"),($(this).attr("o")==="true")?true:false);');
406     facsimilePage.append(zoomButton);
407   });
408 }
```

Listing 10.1: Funktion zum Einfügen der Zoom-Buttons

Kategorie	Funktion	Bedeutung
	zoomIn	hineinzoomen

[a]Details sind hier nachzulesen

Kategorie	Funktion	Bedeutung
	zoomOut	herauszoomen
	zoomReset	Zoom zurücksetzen

Tabelle 10.1: Arten von Zoombuttons

10.5 Anzeige im Fullscreen-Modus

Eine Funktion, die gerade im Hinblick auf kleine Bildschirme wichtig ist, stellt der Fullscreen-Mode dar. Auch dafür existiert ein Button in der Menüleiste. Bei einem Klick auf diesen wird der Browser in den Vollbild-Modus gesetzt und zudem der Header des Readers ausgeblendet.

10.6 Anpassung der Skalierung

Ebenfalls stellt eine Möglichkeit für Benutzer mit kleinen Bildschirmen dar, dass sich der Reader, welcher sich in der Breite defaultmäßig am Bildschirm ausrichtet, skalieren lässt. Zu diesem Zweck gibt es in der Menüleiste eine Liste von Prozentangaben, die von 50% bis 100% reichen und es so ermöglichen, den Reader den eigenen Ansprüchen entsprechend zu verkleinern.

10.7 Erstellung der Faksimile-Extrakte

Eine letzte Funktionalität, die der Server beherrscht, ist das Erstellen eines Faksimile-Ausschnitts abhängig von den Koordinaten. Um dies zu gewährleisten, reicht ein GET-Request an die Serverroute **facsimileExtractor**, dem als Parameter die Dokument-ID, die Auflösung und das jeweilige Siglum angehängt werden.

Falls das Faksimile, von dem ein Ausschnitt erstellt werden soll, nicht gefunden wird, erscheint für den Benutzer eine Fehlermeldung. Im Erfolgsfall werden zuerst die zum Siglum gehörenden Koordinaten aus der Datenbank abgefragt.

Danach wird die Funktion **crop** aus dem Modul *gm*[b] aufgerufen, welche das Faksimile mit den ermittelten Koordinaten über das Programm *ImageMagick*[c] zuschneidet. Zuletzt wird das Ergeb-

[b] weitere Informationen sind im GitHub zu finden
[c] mehr Details können auf der Website gefunden werden

nis in einen JPG-Buffer umgewandelt und zurückgeschickt. Diese Operationen sind in Snippet 10.2 dargestellt.

```
221 image.crop(sizes.w,sizes.h,sizes.l,sizes.t)
222 .toBuffer('JPG',function(manipulationError,buffer){
223   if(manipulationError) return handle(manipulationError);
224   res.header('Content-Type','image/jpeg');
225   res.send(buffer);
226 });
```

Listing 10.2: Zuschneiden eines Faksimile

Kapitel 11

Integration in die Suchmaschine

Dieses Kapitel befasst sich mit der Integration des Faksimileviewers in die Suchmaschine WiTTFind, diese besteht aus drei Schritten.

11.1 Integration als Submodul

Der erste Punkt auf dem Weg zum Einbau der Software stellt die Einbindung als Submodul dar. Da auch das Hauptprojekt als Git-Repository verwaltet wird, bietet sich hier die Verwendung eines *git submodule* an. Die ausgewählten Submodule werden dabei über eine versteckte Datei mit dem Namen **.gitmodules** definiert. Mit Eintragung von

```
13 [submodule "components/quadroreader"]
14   path = components/quadroreader
15   url = git@gitlab.cis.uni-muenchen.de:wast/quadroreader.git
```

in diese Datei wird das Repository des Faksimileviewers als neues Untermodul angegeben, welches im Unterverzeichnis *components/quadroreader* abgelegt wird.

Falls nun Änderungen am Faksimileviewer vorgenommen werden, muss immer ein Komponenten-Update gemacht werden, damit diese auch im Hauptprojekt verfügbar sind. Dies funktioniert über die Abarbeitung der folgenden Punkte:

1. ins Komponentenverzeichnis wechseln, beispielsweise mit **cd components/quadroreader**
2. gegebenenfalls ein **git pull** ausführen, falls es neue Branches gibt
3. den gewünschten Stand mit **git checkout <gewünschter Branch>** auschecken
4. wieder ins Hauptverzeichnis wechseln
5. die Änderungen über **git commit -m <Nachricht> -o components/quadroreader** committen

11.2 Integration in die Deploy-Pipeline

Damit der Faksimileviewer auch tatsächlich verwendet werden kann, wird er in die Deploy-Pipeline des Hauptprojekts eingebaut. Um dies zu gewährleisten, ist in dessen Makefile ein Target namens **deploy-component-quadroreader** enthalten, welches in Snippet 11.1 dargestellt ist.

```
26 deploy-component-quadroreader: $! $(TO)/components/quadroreader/
       facsimile \
27                                   $(TO)/components/quadroreader/
       transcription
28     +$(MAKE) SILENT="@" QUIET="2>&1 >/dev/null" --no-print-directory
       -C components/quadroreader database-insert
29     +$(MAKE) SILENT="@" QUIET="2>&1 >/dev/null" --no-print-directory
       -C components/quadroreader deploy
```

Listing 11.1: Maketarget zum Deploy des Faksimileviewers

Durch die beiden Voraussetzungen werden Links zum Faksimile- und Transkriptionsordner im Komponentenverzeichnis erzeugt. Die weiteren Aufrufe starten einerseits das Einfügen der Daten in die Datenbank und stoßen andererseits den Deployvorgang des Faksimileviewers an. Nach erfolgreichem Abschluss ist dann der Reader unter Port 3000 erreichbar.

11.3 Integration in die Trefferanzeige

Nachdem in der Folge eine Suchanfrage abgeschickt wird, werden - wie in Kapitel 7.2 bereits beschrieben - deren Treffer gesammelt. Daneben wird die Trefferanzeige, welche in Abbildung 1.3 zu sehen ist, aufgebaut. Unter anderem werden hierbei auch der Link zum Faksimileviewer sowie das Faksimile-Extrakt, welches ebenfalls einen Link mit demselben Ziel darstellt, erstellt. Das Ziel besteht dabei aus dem aktuellen Host (im Falle von WiTTFind in der Regel http://wittfind.cis.lmu.de), dem Port 3000 und der Route **reader**. Daran angehängt wird noch der Name der XML-Datei, aus der die Treffer extrahiert wurden. Dies wird später zur Speicherung und Abfrage der Daten benötigt.

Sobald der Benutzer nun auf einen der beiden Links klickt, werden beim Drücken der Maustaste die Daten - bestehend aus der Dokument-ID sowie dem Siglum des jeweiligen Links, den Treffern sowie der aktuell eingestellten Auflösung - gesammelt und per POST-Request an die Serverroute **data** geschickt. Dies geschieht über den Codeblock in Snippet 11.2.

```
81 $("a.facsimileReader, a.facsimileExtract").bind("mousedown", function
       (){
82   var doc = $(this).attr("siglum").split(/,/)[0];
83   doc = _.find(_.keys(hitCounter), function(currentDoc) {
84     return currentDoc.indexOf(doc) == 0;
```

```
85    });
86    var serverData = {};
87    serverData.dataId = hitsFile.split(/\//)[1].replace(/.xml$/,"");
88    serverData.documentId = doc;
89    serverData.hits = JSON.stringify(hits);
90    serverData.siglum = $(this).attr("siglum").split(/\[/)[0];
91    serverData.density = localStorage.getItem(c_density_name);
92    $.post("http://"+window.location.host+":3000/data", serverData,
       function(data){
93      console.log("POST request was sent successfully.");
94    }).fail( function(xhr, textStatus, errorThrown) {
95      console.log(xhr.responseText);
96      console.log(textStatus);
97    });
98  });
```

Listing 11.2: Senden der Daten an den Server

Zum Abschluss wird der Reader mit den entsprechenden Daten gestartet.

Kapitel 12

Ausblick

Im Rahmen dieser Arbeit wurde ein Faksimileviewer entwickelt, der sowohl ein Highlighting von Paragraphen im Faksimile als auch die Anzeige der zugehörigen edierten Texte in verschiedenen Formaten ermöglicht. Erweiterungsmöglichkeiten - insbesondere Features, die aus Zeitmangel nicht mehr realisiert werden konnten - gibt es dabei an mehreren Stellen.

12.1 Erweiterung zu einer Stand-Alone-Applikation

Um die Software tatsächlich zu einer komplett eigenständigen Anwendung auszubauen, muss sie um einen Suchschlitz ergänzt werden. Über diesen sollte es möglich sein, Suchanfragen an die Suchmaschine zu versenden und die Ergebnisse direkt innerhalb des Faksimileviewers darzustellen.

12.2 Integration in die Weboberfläche der Suchmaschine

Ein weiteres Vorhaben, welches aus Zeitgründen nicht umgesetzt werden konnte, war eine konsequente Einarbeitung des Faksimileviewers in die Weboberfläche von WiTTFind. Dabei sollte die Software als Komponente für eine weitere Suchart angeboten werden, was so aussehen könnte, wie in Abbildung 12.1 dargestellt.

12.3 Einbau des Satzhighlighting

Ebenso zeitbedingt nicht eingebaut werden konnte das Highlighting auf Satzebene. Die Koordinaten liegen für das Ts-213 vor, sie müssten in die Datenbank eingepflegt sowie entsprechend wieder ausgelesen und dargestellt werden.

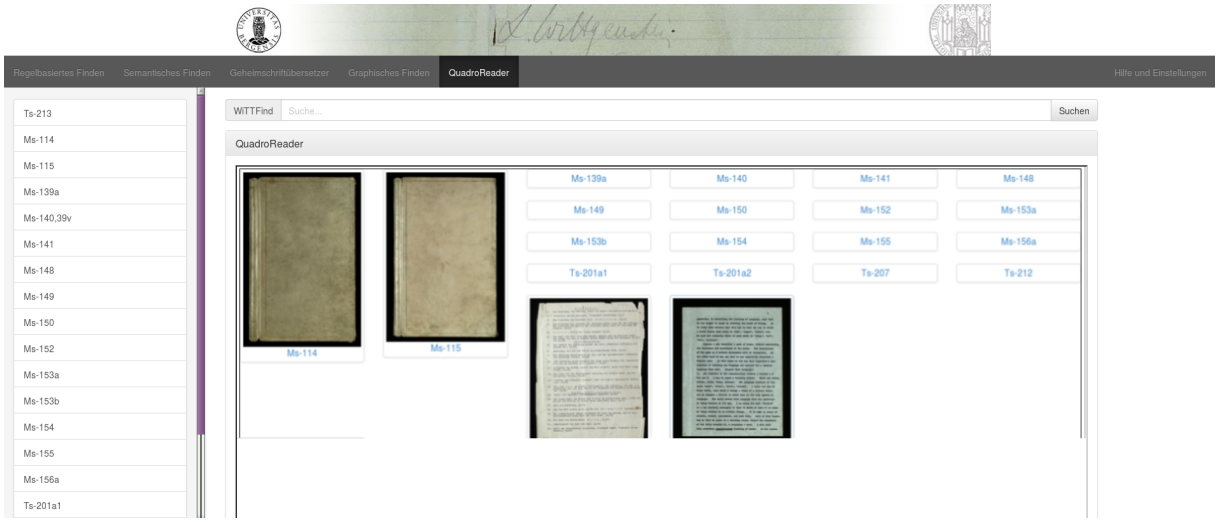

Abbildung 12.1: Integration des Faksimileviewers

Teil III

Anhang

Abbildungsverzeichnis

Tabellenverzeichnis

Listings

Literaturverzeichnis

[1] M. Lindinger, "Highlighting von Treffern des Suchmaschinentools WiTTFind im zugehörigen Faksimile," Master's thesis, CIS, LMU, 2013.

[2] Wikipedia, "Ludwig Wittgenstein," 2005. [Online]. Available: http://de.wikipedia.org/wiki/Ludwig_Wittgenstein. [Accessed: 25-Jun-2015].

[3] U. Bergen, "Wittgenstein's Nachlass. The Bergen Electronic Edition (BEE)," 2010. [Online]. Available: http://wab.uib.no/wab_BEE.page. [Accessed: 25-Jun-2015].

[4] A. Pichler, "Wittgenstein Source: Bergen Text Edition (BTE)." [Online]. Available: http://129.177.5.31/documentation/de/BTE.html. [Accessed: 03-Jun-2013].

[5] A. Pichler, "Wittgenstein Source - Wittgenstein Archives at the University of Bergen (WAB)." [Online]. Available: http://129.177.5.31/documentation/en/home.html. [Accessed: 03-Jun-2013].

[6] C. Müller, "Highlighting von Treffern einer Suchmaschine im Facsimile des Dokuments: Automatische Ermittlung von Positionen," Master's thesis, CIS, LMU, 2014.

[7] R. Capsamun, "WiTTFind: Semiautomatische Korrektur des Highlighting im Faksimile," Master's thesis, CIS, LMU, 2015.

[8] M. Hadersbeck and D. Bruder, *WAST: Wittgenstein Advanced Search Tools Dokumentation*. 2014.

[9] Wikipedia, "LAMP (Softwarepaket)," 2014. [Online]. Available: https://de.wikipedia.org/wiki/LAMP_%28Softwarepaket%29. [Accessed: 02-Jul-2015].

[10] N. Foundation, "Node.js," 2015. [Online]. Available: https://nodejs.org/. [Accessed: 05-Jul-2015].

[11] I. MongoDB, "Operators," 2015. [Online]. Available: http://docs.mongodb.org/manual/reference/operator/. [Accessed: 05-Jul-2015].

[12] F. Lindesay, "Language Reference," 2015. [Online]. Available: http://jade-lang.com/reference/. [Accessed: 06-Jul-2015].

[13] M. Geers, "Jade Syntax Documentation," 2015. [Online]. Available: http://naltatis.github.io/jade-syntax-docs/. [Accessed: 06-Jul-2015].

[14] A. Anbari, "MEAN vs LAMP (Web Development)," 2014. [Online]. Available: https://www.linkedin.com/pulse/20140603115714-58192521-mean-vs-lamp-web-development. [Accessed: 18-Jul-2015].