Import von Office-Dokumenten in eine Python-Webanwendung

Bachelorarbeit

von

Eike Schlingensief

aus

Düsseldorf

vorgelegt am

Lehrstuhl für Rechnernetze und Kommunikationssysteme
Prof. Dr. Martin Mauve
Heinrich-Heine-Universität Düsseldorf

August 2012

Betreuer:
Philipp Hagemeister, M.Sc.

Danksagung

An dieser Stelle möchte ich *Herrn Philipp Hagemeister* für die gute Betreuung danken. Durch viele Anregungen und Ratschläge half er mir meine Bachelorarbeit stetig voranzubringen.

Bei *meiner Familie* und *meiner Lebensgefährtin* möchte ich mich herzlich für ihre moralische Unterstützung und die alltägliche Entlastung bedanken. So konnte ich mich voll und ganz auf meine Bachelorarbeit konzentrieren.

Nicht zuletzt danke ich allen, die mich bei der Textkorrektur mit großem Engagement unterstützt haben.

Inhaltsverzeichnis

Abbildungsverzeichnis vii

Tabellenverzeichnis ix

1. Einleitung 1
1.1. Motivation 1
1.2. Problemstellung 2
1.3. Struktur der Arbeit 2

2. Grundlagen 3
2.1. Python-Webanwendung 3
2.2. Dokument-Typen 3
2.2.1. Office Open XML 3
2.2.2. Open Document 6
2.3. LibreOffice und OpenOffice.org 7
2.4. Universal Network Object (UNO) und Python-UNO-Brücke 8
2.5. Markdown 8

3. Designüberlegung zum Einlesen der Office-Dateien 11
3.1. XML-Parser 11
3.1.1. SAX Parser 12
3.1.2. DOM Parser 12
3.2. Python-UNO-Brücke und LibreOffice/ OpenOffice 13
3.3. Bibliotheken und Skripte 14
3.3.1. PyOdConverter 15
3.3.2. html2text 15

3.3.3. Python-Docx 15
3.3.4. openxmllib 16
3.3.5. odfpy 16
3.3.6. Office File Converter 17
3.4. Vergleich der Umwandlungsmöglichkeiten 17

4. Implementierung **19**
4.1. DocParser 19
4.1.1. Einsatz von *LibreOffice* und Python-UNO 19
4.1.2. Verschiedene XML-Parser 24
4.2. Integration in die Python-Webanwendung 26
4.2.1. Controller 26
4.2.2. Validation einer Office-Datei 27

5. Evaluation **29**
5.1. Durchführung der Benchmarks 30
5.2. Ergebnisse und Auswertung 30

6. Sicherheit **35**

7. Resümee und Ausblick **37**
7.1. Resümee 37
7.2. Ausblick 38

A. Zusätzliche Grafiken **39**

Abbildungsverzeichnis

2.1. Der strukturelle Aufbau eines Containers eines Textdokumentes im *OOXML*-Format 4
2.2. Der strukturelle Aufbau eines Containers eines Textdokumentes im *Open Document Format* 6

3.1. Die Möglichkeiten verschiedene Dokument-Formate mit den gegebenen Bibliotheken und Skripten einzulesen und umzuwandeln 18

4.1. Der schrittweise Ablauf des DocParsers beim Einlesen und Konvertieren eines Office Dokuments 25
4.2. Die Hochlademaske eines Kommentars von *adhocracy* 28

5.1. Gegenüberstellung der beiden Test-Dokumente bei der Konvertierung mit *LibreOffice* in HTML 31
5.2. Der Zeitverbrauch bei der Konvertierung des Test-Dokuments "Wikipedia_Deutschland.docx" 32
5.3. Die Verteilung des Zeitverbrauchs bei der Konvertierung des Test-Dokuments "Wikipedia_Deutschland.doc"in HTML mit *LibreOffice* 33

A.1. Darstellung des Beispieldokuments im Programm *LibreOffice* 39
A.2. Darstellung des Beispieldokuments im Browser *Mozilla Firefox* vor der Reinigung des HTMLs durch den DocParser 40
A.3. Darstellung des Beispieldokuments im Browser *Mozilla Firefox* nach der Reinigung des HTMLs durch den DocParser 41

Tabellenverzeichnis

2.1. Ein Teil der Formatierungsmöglichkeiten mit Markdown 9

4.1. Die Optionen des DocParsers 20

5.1. Benchmark für das Einlesen und Umwandeln des Deutschen Grundgesetzes (ca. 58 Seiten) 34

5.2. Benchmark für das Einlesen und Umwandeln des Wikipediaeintrages "Deutschland"(ca. 77 Seiten) 34

Kapitel 1.

Einleitung

1.1. Motivation

Neue Technologien ermöglichen es zum ersten Mal, direkte Demokratie effizient einzusetzen. Die dieser Arbeit zugrunde liegende Kommunikationsplattform *adhocracy* [15] wurde für das theoretische Konzept *Liquid Democracy* [14] entwickelt. *Liquid Democracy* kombiniert die direkte Demokratie und die repräsentative Demokratie, was auch als "Direkter Parlamentarismus" bezeichnet wird. Den Bürgerinnen und Bürgern soll es ermöglicht werden direkt an Diskursen mitzuarbeiten und über verschiedene Vorschläge abzustimmen. Dem Bürger steht es frei, ob er nur über den Inhalt eines Diskurses abstimmen oder aktiv Ideen einbringen möchte. Dieses Konzept findet nicht nur in der Politik Verwendung, sondern auch in Vereinen, Firmen und Organisationen.

Der heutige Stand der Technik ermöglicht es dieses theoretische Konzept umzusetzen. *Adhocracy* organisiert die Diskurse und die Entscheidungsprozesse und gewährleistet, dass diese transparent und nachvollziehbar sind. In der Heinrich-Heine-Universität wird *adhocracy* als Diskussionsplattform genutzt. Zur Zeit müssen neue Diskussionen und Kommentare über ein Textfeld in der Webanwendung eingegeben werden. Alle Diskussionen und Vorschläge, die über ein Office-Dokument liefen, müssen in Handarbeit in *adhocracy* übertragen werden. Momentan werden die Office Dokumente durch einen Mitarbeiter in das universitätseigene *adhocracy* eingepflegt. Um dies zu erleichtern, soll

eine Importmöglichkeit für Office-Dokumente in *adhocracy* geschaffen werden.

1.2. Problemstellung

Im Rahmen dieser Bachelorarbeit soll die Möglichkeit eines Imports von mindestens einem Office Dokumentformat in eine Python-Webanwendung geschaffen werden. Das im Web-Framework *Pylons* entwickelte *adhocracy* soll damit erweitert und vereinfacht werden. Durch die Erweiterung sollen Inhalte, sowie nach Möglichkeit Kommentare zu Inhalten und Änderungen an Inhalten ("Änderungen verfolgen"), durch einfaches Importieren eines Dokuments hinzugefügt werden können. Es werden dazu zunächst verschiedene bestehende Lösungen für das Importieren eines Office-Dokumentes in Python untersucht und verglichen. Anschließend wird die am besten geeignete Lösung implementiert.

1.3. Struktur der Arbeit

In Kapitel 2 wird zunächst auf das verwendete Web-Framework *Pylons* eingegangen. Anschließend wird die Struktur von zwei XML-basierter Dokument-Typen erläutert. Diese und ein weiteres Dokument-Format sollen in das Web-Framework integriert werden. Danach wird das *Universal Network Object (UNO)* und die *Python-UNO-Brücke* erläutert, die für die spätere Implementierung von Bedeutung sind. Mit einer Übersicht über die Möglichkeiten, einen Text mit *Markdown* zu formatieren, wird dieses Kapitel abgeschlossen. In Kapitel 3 werden eine Reihe von Bibliotheken zur Konvertierung von Office-Dokumenten vorgestellt. Des Weiteren werden die verschiedenen Möglichkeiten des Einlesens diskutiert. Kapitel 4 erläutert daraufhin die Implementierung der beiden gewählten Möglichkeiten des Importes. Kapitel 5 beschäftigt sich mit der Evaluation der Implementierungen. Es wurden grundsätzlich verschiedene Dokument-Typen in einigen Benchmarks gegenübergestellt. Dafür werden verschiedene Test-Dokumente verwendet. Das 6. Kapitel stellt mehrere Strategien vor ein Programm auf einem Linux-Server abzusichern. Das letzte Kapitel 7 beinhaltet ein Resümee und einen Ausblick der Arbeit.

Kapitel 2.

Grundlagen

2.1. Python-Webanwendung

Pylons [6] ist eine open source Web Applikation und wurde in Python geschrieben. *Pylons* ermöglicht es, eigene WSGI-Frameworks (Web Server Gateway Interface) zu realisieren. Dabei bietet *Pylons* hauptsächlich ein Request-Handling und einen Debugger an. Andere Komponenten wie Session, Routing, Templating oder Datenbankzugriffe müssen zusätzlich integriert werden. Welche Komponenten dafür benutzt werden, steht dem Entwickler frei. Außerdem lassen sich die Komponenten relativ einfach austauschen.

2.2. Dokument-Typen

2.2.1. Office Open XML

Office Open XML (OOXML) [5] wurde von Microsoft als offener Standard für XML-basierte Dateiformate entwickelt und ist ein ISO-Standard. Dieses Format wird in *Microsoft Office* 2007/2010/2013 verwendet. *OOXML* spezifiert Text-, Tabellen- und Präsentationsdokumente. Hier wird nur auf das Format der Textdokumente eingegangen.

Die Textdokumente werden als Packages gespeichert. Die ZIP-Technologie wird als Container benutzt, um den Inhalt des Dokumentes in verschiedene Ordner und Dateien zusammen zu packen. Diese Struktur folgt den Regeln von *OPC (Open Packaging Conventions)* [18]. Dieses Package kann mit einem Standard-Datenkompressionsprogramm geöffnet werden und ist damit frei einsehbar. Die meisten Dateien in dem Container beinhalten XML. Es können sich aber auch binäre Dateien im Container befinden, wie z.B. Bilder oder Makros. Durch die Ablage der Textinformationen als XML-Dateien lassen sie sich leicht einlesen und verarbeiten. Die Dateien im "*word*" Ordner sind von großem Interesse, da diese die Texte und die Formatierungen des Textdokuments beinhalten. Der Inhalt eines Containers wird in der Abbildung 2.1 dargestellt.

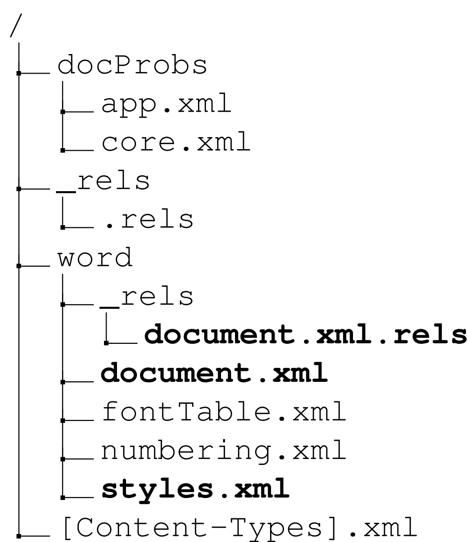

Abbildung 2.1.: Der strukturelle Aufbau eines Containers eines Textdokumentes im *OOXML*-Format

Die XML-Dateien eines Textdokumentes beruhen auf der *WordProcessingML* Architektur [10] von Microsoft. Die XML-Dateien haben verschiedene Funktionen. Die meisten beinhalten Informationen über Format und Struktur des Textdokuments. Einige XML-Datei enthalten Informationen über die im Container befindlichen Datei-Typen und die Verlinkungen zu verschiedenen Dateien. Die wichtigsten XML-Dateien für diese Arbeit sind die Datei *document.xml* und die Datei *style.xml*, da diese den gesamten geschriebenen Text und die Formatvorlagen des Textdokuments beinhalten.

WordProcessingML (WordML) [10] wurde von Microsoft entwickelt und ist eine Anwendung von XML. In dieser Arbeit werden nur die wichtigsten Tags und Attribute des XMLs erläutert. Da *Markdown* nur die wesentlichen Textformatierungen unterstützt, werden nur die relevanten Elemente behandelt. *Markdown* wird im Kapitel 2.5 erläutert. Das Tag *w:body* umfasst den Inhalt des *WordML*-Dokuments. Das *w:p* Tag kennzeichnet einen Absatz und definiert über das Tag *w:pPr* die Formatierung des Textes für den gesamten Absatz. Dieses Tag kann mehrere *w:r* (Runs) Tags enthalten, die für einzelne Textabschnitte individuelle Formatierungen vorgeben können. Dieses Formatierungen befinden sich im *w:rPr* Tag, das dem *w:r* Tag untergeordnet ist. Das *w:rPr* Tag kann z.B. *w:i* Tag für "kursiv", *w:b* Tag für "fett", *w:br* Tag für einen Umbruch und *w:rStyle* für die Verlinkung zu den Formatvorlagen enthalten. Das *w:t* Tag beinhaltet nur Text und hat keine untergeordneten Tags mehr.

```
<w:p>
  <w:pPr>
    <w:pStyle w:val="berschrift1"/>
  </w:pPr>
  <w:r>
    <w:t>Eine Überschrift</w:t>
   </w:r>
</w:p>
<w:p>
  <w:r>
    <w:rPr>
      <w:rStyle w:val="Hervorhebung"/>
      <w:i/>
      <w:b/>
    </w:rPr>
    <w:t>Ein Text</w:t>
  </w:r>
</w:p>
```

Listing 2.1: Auszug aus einem WordML-Dokument

2.2.2. Open Document

OASIS Open Document Format for Office Applications wurde ursprünglich von Sun Microsystems entwickelt und von der Organisation OASIS [9] weitergeführt und ist standardisiert. Einige Office-Pakete wie z.B. LibreOffice[11], KOffice und OpenOffice[19] haben dies als Standardformat eingeführt. Hier wird wieder nur auf das Format der Textdokumente eingegangen.

Diese Textdokumente sind ebenfalls als Packages gespeichert und benutzen die ZIP-Technologie als Container. Allerdings sieht die Struktur der Dateien und Ordner anders aus als bei dem *OOXML*-Format. Die Elemente des XMLs sind an den Standard HTML angelehnt und stärker verschachtelt als bei dem *OOXML*-Format. Die Texte und die Formatvorlagen befinden sich in den XML-Dateien *content.xml* und *style.xml*. Eine Darstellung eines solchen Containers findet man in der Abbildung 2.2. Ein Absatz im Text

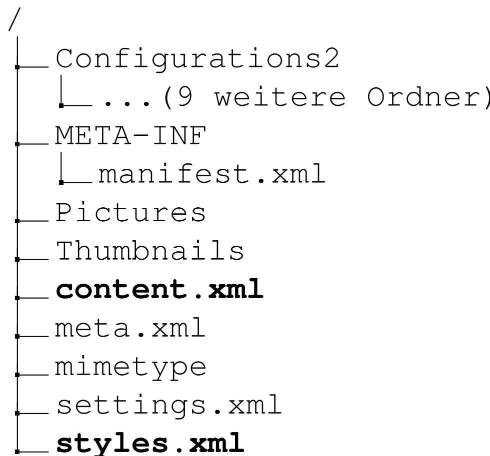

Abbildung 2.2.: Der strukturelle Aufbau eines Containers eines Textdokumentes im *Open Document Format*

wird im XML als *text:p* Tag dargestellt. Der Text kann in verschiedene text:span Tags verschachtelt sein. Die Information über die Formatierung des Textes befindet sich im Style-Tag am Anfang der XML-Datei oder in den Formatvorlagen in der *style.xml*-Datei. Diese werden über das Attribut *style:name* im Tag verlinkt, ähnlich wie die Klassen im HTML.

```
<!-- Ein Auszug aus der style.xml -->
<style:style style:family="paragraph" style:name="T9"
      style:display-name="T9">
  <style:text-properties style:font-name="DejaVuSerif"
              fo:font-weight="bold"/>
</style:style>

<!-- Ein Auszug aus der content.xml -->
<text:p text:style-name="Kein_20_Leerraum">
  <text:span text:style-name="Absatz-Standardschriftart">
    <text:span text:style-name="T6">Ein Text</text:span>
  </text:span>
  <text:span text:style-name="T9">
    Ein Text fett
  </text:span>
</text:p>
<text:p text:style-name="P2"/>
<text:h text:style-name="P3" text:outline-level="1">
  Eine Überschrift
</text:h>
```

Listing 2.2: Auszug aus dem XML eines Textdokuments im *Open Document Format*

2.3. LibreOffice und OpenOffice.org

LibreOffice [11] ist eine freie Software und ein Projekt der *Document Foundation*. *LibreOffice* läuft unter verschiedenen Betriebssystemen wie *Microsoft Windows 2000* oder höher, *Linux* (Kernel 2.6.18 oder höher) oder *MacOS X*. In dem Office Paket bietet *LibreOffice* unter anderem die Möglichkeit der Textverarbeitung. *LibreOffice* ist eine Abspaltung von *OpenOffice* und wird unter der Leitung der *Document Foundation* weiterentwickelt. Durch die ständige Weiterentwicklung können auch zukünftige Formate von Office-Dateien umgewandelt werden.

OpenOffice.org [19] ist eine freie Software der *Apache Software Foundation* und bietet momentan ähnliche Features wie *LibreOffice*. Ob das Projekt *OpenOffice* weiterentwi-

ckelt wird bzw. in welchem Umfang, ist derzeit unklar.

2.4. Universal Network Object (UNO) und Python-UNO-Brücke

Universal Network Objects (UNO) [20] ist ein Komponentenmodell, das von LibreOffice/OpenOffice genutzt wird. UNO gewährleistet Interoperabilität zwischen verschiedenen Programmiersprachen, Objekt-Modellen und Plattformen, entweder über LAN oder Internet. UNO basiert auf Schnittstellen wie COM und COBRA. Die Kommunikation zwischen den Komponenten läuft über Schnittstellen, die in verschiedenen Programmiersprachen implementiert werden können. UNO garantiert Objekt-Identität, Thread-Identität, und die Reihenfolge der Aufrufe. Die UNO-Komponenten laufen in einer sogenannten Uno Runtime Environment (URE). Jede URE wird über seine Programmiersprache und den Prozess identifiziert. Die Kommunikation zwischen verschiedenen UREs wird durch UNO überbrückt. UNO Objekte werden in der abstrakten Meta Sprache *UNO IDL* (UNO Interface Definition Language) [24] spezifiziert. UNO wird von der OpenOffice.org und Sun Microsystems vertrieben und die Implementierung läuft unter der LGPL Lizenz.

Die Sprachanbindung Python-UNO [8] erlaubt es, mit der OpenOffice.org-API zu interagieren. Mit Python-UNO können alle UNO Komponenten verwendet werden und es können auch eigene UNO Komponenten entwickelt werden, die dann im OpenOffice Prozess laufen. Das Python-Skript, das Python-UNO benutzt, kommuniziert mit dem OpenOffice Prozess über eine Interprozessbrücke. OpenOffice wartet auf einem *Socket* auf einen Aufruf.

2.5. Markdown

Markdown [21] bietet eine Möglichkeit einen Text einfach zu formatieren, ohne dass die Lesbarkeit darunter leiden muss. Des Weiteren lässt sich mit einfachen Mitteln daraus

valider XHTML Code erzeugen, der dann an den Browser ausgeliefert werden kann. Mehrere Beispiele werden in der Tabelle 2.1 dargestellt.

Markdown	HTML	Browser
# Überschrift 1 ### Überschrift 3	Überschrift 1 Überschrift 3	Überschrift 1 Überschrift 3
Kursiv	Kursiv	*Kursiv*
Fett	Fett	**Fett**
Fett kursiv	 Fett und Kursiv 	***Fett kursiv***
* Punkt 1 * Punkt 2	 - Punkt 1 - Punkt 2 	• Punkt 1 • Punkt 2
[Link zu Google] (http://www.google.de)	 Link zu Google	Link zu Google
[Beschreibung vom Bild] !(http://www.beispiel.de/bild.jpg)		

Tabelle 2.1.: Ein Teil der Formatierungsmöglichkeiten mit Markdown

Kapitel 3.

Designüberlegung zum Einlesen der Office-Dateien

In diesem Kapitel werden die geprüften Vorgehensweisen erläutert, wie ein Office Dokument eingelesen und in *Markdown* bzw. in HTML umgewandelt werden kann. Dabei soll hauptsächlich das binäre Dokumenten-Format "doc" von *Microsoft Office 97/2000/XP*, das *OOXML*-Format ".docx" und das *Open Document Format* (ODF) ".odt" umgewandelt werden.

3.1. XML-Parser

Wenn das Office-Dokument im *OOXML*-Format/*ODF* vorliegt, lässt es sich als Zip-Datei öffnen und die entsprechenden XML-Dateien parsen. Dadurch werden zwei Formate einer Office-Datei abgedeckt. Das binäre Format von doc-Format kann nicht mit Hilfe eines ZIP-Dekompressors umgewandelt werden. Hier ist eine vorherige Umwandlung in das *OOXML*-Format oder *ODF* notwendig.

Um das Dokument in HTML oder *Markdown* umzuwandeln, wird die entsprechende XML-Datei eingelesen und mit einem XML-Parser geparst. Diese Dateien befinden sich im Package an verschiedenen Stellen des Text-Dokumentes. Eine Struktur für das Packa-

ge der jeweiligen Dokument-Formate wird in den Abbildungen 2.1 und 2.2 dargestellt. Es gibt zwei gängige Modelle von XML-Parsern: SAX Parser und DOM Parser.

3.1.1. SAX Parser

Ein SAX Parser (Simple Api for XML) liest das XML als sequentiellen Datenstrom ein. Dabei werden bei Events, wie z.B. beim Öffnen oder Schliessen eines Tags, Callbackfunktionen aufgerufen, in denen die Attribute der Tags verarbeitet oder der Text eingelesen und entsprechend in *Markdown* umgewandelt werden kann. Da der SAX Parser immer nur Teile des XML einliest und in den Speicher legt, verbraucht der SAX Parser während des Einleses nicht viel Speicher. Ein Nachteil ist, dass der SAX Parser viele unnötige Funktionsaufrufe hat und dies die Rechenzeit verlängern kann. Das XML vom Office Open XML Format und vom Open Document Format beinhaltet viele Informationen, die für die Umwandlung irrelevant sind. Für diese Teile des XML werden trotzdem Callbackfunktionen aufgerufen, da immer die komplette XML-Struktur durchlaufen wird. Die Standard-Bibliothek von Python bietet einen SAX Parser an, mit dem Package *xml.sax*. Die bessere Alternative zu der Standard-Bibliothek von Python ist die *lxml* [12] Bibliothek. Diese Bibliothek nutzt die C-Bibliotheken *libxml2* und *libxslt* und ist dadurch wesentlich schneller als die Standard-Bibliothek. *lxml* wird unter der BSD Lizenz und die C-Bibliotheken unter der MIT Lizenz vertrieben. Dies bedeutet, dass sie frei verwendet werden dürfen.

3.1.2. DOM Parser

Ein DOM Parser liest das komplette XML auf einmal ein und parst es zu einer Baumstruktur. Dies hat den Vorteil, dass man einzelne Elemente suchen, ersetzen oder ändern kann. Durch das komplette Einlesen des XML wird zunächst mehr Speicher verbraucht. Beim Durchlaufen der XML-Baumstruktur wird jedoch Rechenzeit gespart, da nicht für jedes Tag, das geöffnet und geschlossen wird, eine Callbackfunktion aufgerufen wird. Beim Durchlaufen der XML-Baumstruktur lassen sich die Attribute der Elemente und der Text durch Funktionsaufrufe ermitteln und verarbeiten, um danach den Text in das

entsprechende *Markdown* umzuwandeln. Dadurch, dass der DOM Parser eine Baumstruktur aus dem XML erzeugt, können die irrelevanten Teile übersprungen werden. Die Standard-Bibliothek von Python bietet mit dem Package *xml.dom* einen DOM Parser an. Auch in diesem Fall ist die im Kapitel SAX Parser erwähnte Bibliothek *lxml* [12] die bessere Alternative.

3.2. Python-UNO-Brücke und LibreOffice/ OpenOffice

Eine andere Möglichkeit ein Office Dokument einzulesen und umzuwandeln, besteht über die Python-UNO-Brücke mit *LibreOffice*. Dazu ist es notwendig, *LibreOffice* auf dem Server zu installieren und als Dienst im Hintergrund laufen zu lassen. Dadurch wird allerdings Arbeitsspeicher und Speicherplatz des Servers verbraucht. Über die Python-UNO-Brücke, die mit *LibreOffice* mit installiert wird, lässt sich dann *LibreOffice* mit einem Python-Skript über einen *Socket* steuern. Die hochgeladene Office-Datei wird dann mit der Python-UNO-Brücke in *LibreOffice* geladen und beispielsweise als HTML-Datei umgewandelt und abgespeichert. Bei der Umwandlung in HTML werden die Bilder in einer Office-Datei automatisch von *LibreOffice* in *Base64* konvertiert und direkt in das jeweilige *img*-Tag eingefügt. Dies wird mit sogenannten *Data-URLs* [1] realisiert. Mit *Base64* lassen sich binäre Dateien in eine Zeichenfolge kodieren. Diese Zeichenfolge besteht aus lesbaren *ASCII*-Zeichen. In *LibreOffice* können verschiedene Autoren Kommentare im Text setzen. Diese Kommentare werden im HTML mit einem *title*-Attribut in einem Tag gekennzeichnet. Diese müssen separat herausgefiltert werden.

Durch die vorherige Umwandlung in HTML wird weiterer Speicherplatz verbraucht, dessen Größe schwer abschätzbar ist. Die Größe des benötigten Speicherplatzes ist abhängig von der Länge der Texte, der Menge der Formatierungen sowie der Anzahl der Bilder in einer Office-Datei. Optional kann das HTML als Datei abgespeichert werden. Diese müsste dann nach der Konvertierung geöffnet werden, um an den Inhalt zu gelangen. Dafür müsste für jeden Hochladevorgang ein eindeutiges temporäres Verzeichnis angelegt werden, um paralleles Hochladen und Umwandeln von Office-Dateien zu

[1] Data-URLs bieten die Möglichkeit Daten direkt in das HTML einzubetten ohne auf eine externe Referenzierung zurückzugreifen. Dafür werden die Daten mit *Base64* kodiert.

ermöglichen. Weiterhin muss dieses temporäre Verzeichnis wieder gelöscht werden, um "Datenleichen" zu vermeiden, die beim Abbruch durch den Benutzer während des Hochladens entstehen können. Allerdings ist es auch möglich, die Office-Datei umzuwandeln ohne sie abzuspeichern.

Das erzeugte HTML enthält viele nutzlose Informationen und Strukturen und sollte deswegen gereinigt werden, um den HTML-Code auf das Wesentliche zu beschränken. Dies erleichtert die weitere Verarbeitung in *Markdown*. Die Informationen über die Formatierung des Textes befindet sich in einem Style-Tag am Anfang des HTML und muss separat eingelesen werden. Diese Informationen müssen zusätzlich in das HTML eingefügt werden, damit sie bei dem Reinigen des HTMLs nicht verloren gehen. *LibreOffice* mit der Python-UNO-Brücke ermöglicht es, mehrere Office-Dateien gleichzeitig zu laden und abzuspeichern. Durch *LibreOffice* können viele Formate von Office-Dateien umgewandelt werden, unter anderem auch das binäre Office-Format "doc" von Microsoft. Das löst das Problem, dass das binäre Office-Format "doc" nicht ohne Weiteres mit einer ZIP-Bibliothek geöffnet werden kann.

Weil *LibreOffice* als separater Prozess läuft, muss bei jedem Hochladen einer Office-Datei überprüft werden ob *LibreOffice* noch läuft. Bei einem möglichen Absturz muss dieser Prozess neu gestartet werden. Dies kostet weitere Rechenzeit aufgrund des Startvorgangs von *LibreOffice*. Durch die Python-UNO-Brücke ist es möglich, *LibreOffice* auf einem eigenen Server laufen zu lassen, da man über das lokale Netzwerk oder das Internet darauf zugreifen kann. Dies kann aus Sicherheitsgründen, auf die im Kapitel 6 genauer eingegangen wird, sinnvoll sein.

3.3. Bibliotheken und Skripte

In diesem Abschnitt werden mehrere Bibliotheken und Skripte vorgestellt, die das Einlesen oder Konvertieren von Office-Dateien unterstützen.

3.3.1. PyOdConverter

Der *PyOdConverter* [22] ist ein Python-Skript, das ein Office Dokument in ein von *LibreOffice* unterstütztes Format umwandelt, indem es das Dokument über die Python-UNO-Brücke in *LibreOffice* lädt und abspeichert. Dazu muss auf jeden Fall *LibreOffice* und die Python-UNO-Brücke installiert sein, außerdem muss *LibreOffice* mit den entsprechenden Optionen gestartet sein. Die Office-Datei wird in dem gewünschten Format abgespeichert. Dieses Skript ist hauptsächlich für den Aufruf mit einer Konsole gedacht. Die wesentlichen Methoden bestehen aus wenig Quellcode und können selbst implementiert und damit beliebig erweitert werden. Dieses Skript wird unter der LGPL vertrieben.

3.3.2. html2text

html2text [23] ist ein Python-Skript, das HTML-Code in *Markdown* umwandelt. Dieses Skript unterstützt kein CSS [2] für die Formatierung des Textes im HTML-Code. Das bedeutet, dass die für *Markdown* relevanten Formatierungen in HTML umgewandelt werden müssen. Das Skript benutzt den HTMLParser aus der Standard-Bibliothek von Python. Der HTMLParser ist ähnlich aufgebaut, wie ein SAX Parser und stellt bestimmte Callback-Funktionen zur Verfügung. Diese parsen den HTML-Code mit regulären Ausdrücken. Das Skript *html2text* wird unter der GPLv3 vertrieben.

3.3.3. Python-Docx

Die *Python-Docx* [5] Bibliothek ermöglicht das Erstellen, Lesen und Editieren eines docx-Dokuments. Der Schwerpunkt der Bibliothek liegt mehr im Erstellen und Editieren eines Word-Dokuments als im Auslesen. Das XML des Dokuments wird mit einem DOM Parser von der *lxml* Bibliothek geparst. Dazu wird die *word/document.xml* Datei eingelesen. Wenn der Text des Dokumentes ausgelesen wird, erhält man den Text ohne Formatierung zurück, wodurch eine weitere Umwandlung in HTML oder *Markdown*

[2] CSS (Cascading Style Sheets) ist eine Formatierungssprache, die als Erweiterung von HTML und anderen Auszeichnungssprachen gedacht ist.

nicht möglich ist. Falls ein Dokument Kommentare beinhaltet, gibt es keine Möglichkeit diese durch die Python-Docx Bibliothek zu extrahieren. Dies liegt daran, dass die *word/-comments.xml*-Datei nicht eingelesen wird, in der sich die Kommentare befinden. Des Weiteren lassen sich keine Bilder extrahieren. Diese Bibliothek ist unter der MIT-Lizenz freigegeben.

3.3.4. openxmllib

Durch die *openxmllib* [4] Bibliothek lässt sich der Text aus einem Mircosoft Word 2007/2010 Dokument (.docx) extrahieren, indem die *word/document.xml*-Datei mit der Bibliothek *lxml* geparst wird. Darüber hinaus können verschiedene Eigenschaften wie Autor, Erstellungsdatum und Titel ausgelesen werden. Auch bei dieser Bibliothek erhält man den Text ohne Formatierung, wodurch eine weitere Verarbeitung in HTML oder *Markdown* ebenfalls nicht möglich ist. Auch ein Auslesen der Kommentare oder Bilder eines Dokumentes ist mit dieser Bibliothek nicht möglich. *openxmllib* wird unter der GNU GPL v2 vertrieben.

3.3.5. odfpy

Der Schwerpunkt dieser Bibliothek [7] liegt bei dem Erstellen und Manipulieren von Dokumenten im *Open Document*-Format. Sie liefert mehrere Skripte zum Umwandeln dieser Dokumente in verschiedene andere Formate, unter anderem auch in HTML. Für die Umwandlung in HTML wird das Skript *odf2xhtml* verwendet. Dieses Skript nutzt einen SAX Parser aus der Python Standard-Bibliothek. Allerdings wird bei der Umwandlung in HTML die Formatierung in CSS Style-Tags eingetragen. Alle Texte werden in P-Tags geschrieben und mit einer ID oder Klasse auf das CSS referenziert. Dadurch wird das HTML kompakter, aber es erschwert die Umwandlung in *Markdown*, zum Beispiel mit dem Skript *html2text*. Dieses geht nicht auf das CSS im Style-Tag ein. Für diesen Fall müssten die relevanten Informationen aus dem *CSS* ausgelesen und in HTML umgewandelt werden (Beispielsweise wird die CSS-Angabe "font-weight:bold" in ein B-Tag umgewandelt). Auf die Kommentare und Bilder geht die Bibliothek nicht ein. Diese Bi-

bliothek ist unter der Apache Lizenz 2.0 freigegeben.

3.3.6. Office File Converter

Microsoft bietet eine Möglichkeit an, das binäre Dokumenten-Format (.doc) in das *Office Open XML* 2007 Format (.docx) zu konvertieren. Dies wird durch das Tool *Office File Converter* [16] realisiert. Dieses Tool gehört zu der Sammlung des *Office Migration Planning Manager* (OMPM) und kann ein oder mehrere binäre Office Dateien von Microsoft in das DOCX-Format umwandeln. Die konvertierten Office-Dateien können beispielsweise mit einem XML-Parser weiter in *Markdown* umgewandelt werden. Die Voraussetzung dieses Tools ist die Installation des *Microsoft Office Compatibility Pack*. Dieses Tool läuft nur unter einem Windows Betriebssystem und wird über eine Konsole gesteuert. *Microsoft Office* ist nicht lizenzfrei erhältlich.

Um eine Windows-Installation auf dem Server zu vermeiden, bietet das Programm *Wine* [17] eine Alternative. Über Wine lassen sich Windows-Programm unter Linux installieren und ausführen. Zur Zeit werden noch nicht alle Windows-Programme unterstützt, aber Wine wird ständig weiterentwickelt. Da Wine kein Emulator ist, sondern die Systemaufrufe direkt an Linux weitergibt, dürfte die Performance des Programms akzeptabel sein. Wine wird unter der LGPL vertrieben.

3.4. Vergleich der Umwandlungsmöglichkeiten

Alle geforderten Dokument-Formate abzudecken, gelingt nur durch eine Kombination aus Bibliotheken und Skripten. Die Abbildung 3.1 zeigt verschiedene Kombinationen von Bibliotheken und Skripten. Die meisten Bibliotheken decken nur ein Dokument-Format ab und oftmals erhält man den Text ohne Formatierung. Dadurch fallen die meisten Bibliotheken weg.

Die Kombination aus *LibreOffice* und Python-UNO ermöglicht die Konvertierung aller geforderten Dokument-Formate in ein einheitliches und valides HTML-Dokument. Ein

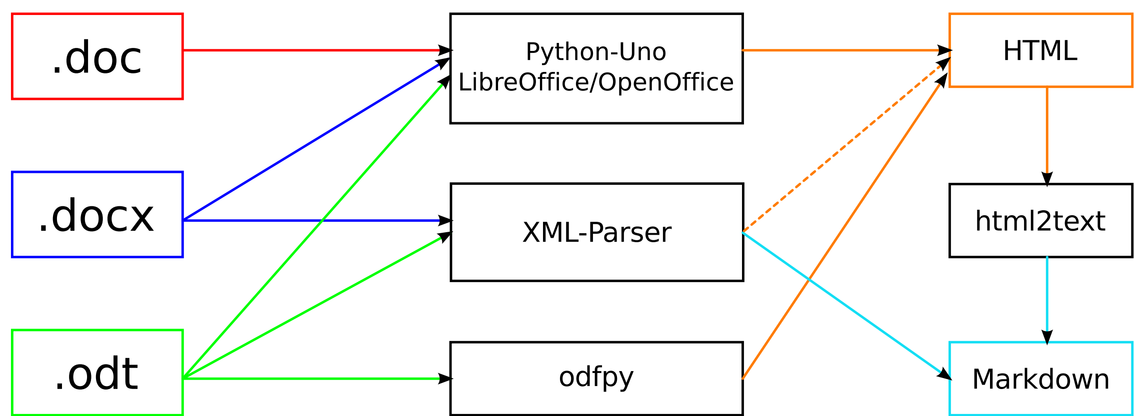

Abbildung 3.1.: Die Möglichkeiten verschiedene Dokument-Formate mit den gegebenen Bibliotheken und Skripten einzulesen und umzuwandeln

Nachteil ist, dass auf dem Server *LibreOffice* installiert und als Dienst gestartet werden muss. Zudem muss der Server die entsprechenden Ressourcen bereitstellen, um eine Verarbeitung der Office-Dokumente in einer angemessenen Zeit gewährleisten zu können. Weiterhin muss bei jedem Hochladen eines Dokuments kontrolliert werden, ob der Dienst *LibreOffice* noch aktiv ist. Im Falle eines Absturzes muss *LibreOffice* neu gestartet werden. Zusätzlich erfordert das Programm eine Absicherung und regelmäßige Aktualisierungen um die Sicherheit des Servers zu gewährleisten. Das führt zu einem erhöhten Wartungsaufwand und kann bei dem Aktualisieren des Programms zu Störungen im Betrieb der Webanwendung führen. Das erzeugte HTML-Dokument kann dann mit dem Skript *html2text* in *Markdown* umgewandelt werden.

Die XML-Parser können nur zwei der drei Dokument-Formate umwandeln. Es wird bei dem Umwandeln der Office-Dokumente kein zusätzlicher Arbeitsspeicher verbraucht, was ein Vorteil gegenüber einem separaten Programm ist. Allerdings muss für jedes Dokument-Format ein eigener XML-Parser geschrieben werden, der das Office-Dokument in *Markdown* oder HTML konvertiert. Die Struktur des XML kann sehr komplex sein und muss vorher gesichtet werden. Für die Kommentare und Bilder muss eine zusätzliche Lösung gefunden und implementiert werden. Diese Lösung zu implementieren, bedeutet einen hohen Aufwand für den Programmierer. Dafür kann das Ausgabeformat selbst gewählt und individuell angepasst werden.

Kapitel 4.

Implementierung

In diesem Kapitel wird auf die konkrete Implementierung des DocParsers und der XML-Parser eingegangen. Anschließend wird die Integrierung des DocParsers in das Web-Framework *Pylons* erläutert.

4.1. DocParser

Die DocParser Klasse wurde in Python geschrieben und steuert das Einlesen und Umwandeln eines Office-Dokuments. Der DocParser bietet zwei Möglichkeiten, ein Office-Dokument einzulesen und umzuwandeln. Zum einen kann das Office-Dokument mit dem Textverarbeitungsprogramm *LibreOffice* umgewandelt werden, zum anderen mit einem der selbst geschriebenen XML-Parser. Dies kann über die Optionen gesteuert werden. Im DocParser sind vier Klassen enthalten, die in den anschließenden Unterkapiteln erläutert werden. Die Abbildung A.1 zeigt ein Beispieldokument in *LibreOffice*.

4.1.1. Einsatz von *LibreOffice* und Python-UNO

Die Abbildung 4.1 zeigt das schrittweise Einlesen und Umwandeln eines Office Dokuments.

Optionen	Beschreibung	Standardwert
-f [Datei]	Pfad zu der Datei oder die Datei als bytestrings	—
-s	Verwendet einen SAX Parser	nein
-d	Verwendet einen Dom Parser	nein
-o	Verwendet LibreOffice	ja
-i	Bilder werden beachtet	nein
-m	Falls LibreOffice verwendet wird, wird der Text weiter in Markdown umgewandelt	nein

Tabelle 4.1.: Die Optionen des DocParsers

Schritt 1

Im ersten Schritt wird durch den Benutzer ein Kommentar oder ein Vorschlag in *adhocracy* erstellt. Dazu wählt er über eine Hochlademaske eine Office-Datei aus und lädt diese hoch. Ein Beispiel für eine Hochlademaske zeigt die Abbildung 4.2. *Pylons* validiert diese Office-Datei und erstellt eine Instanz vom DocParser. Anschließend wird dem DocParser die Office-Datei mit den entsprechenden Optionen übergeben. Daraufhin werden alle Bibliotheken geladen und die Klassenvariablen vom DocParser initialisiert. Außerdem werden die Optionen mit der Python-Bibliothek *optparse* geparst. Die verfügbaren Optionen zeigt die Tabelle 4.1.

Schritt 2

Zunächst wird mit Hilfe der Python-UNO Bibliothek eine Verbindung zu *LibreOffice* über ein Socket aufgenommen. Dazu wird als erstes die *UNO*-Komponente *Kontext* geladen. Über diese Komponente wird der *ServiceManager* verfügbar, mit dem die Dokumente geladen werden können. Der *ServiceManager* liefert die Klasse *UnoUrlResolver*, über den sich die lokale Kontext-Komponente mit der Kontext-Komponente von *LibreOffice* verbindet. Die Klassen können mit der Methode *createInstanceWithContext* erstellt werden. Für diese Verbindung wird das Protokoll *urp* (UNO remote protocol) verwendet. Nun kann die Kontext-Komponente von *LibreOffice* benutzt werden.

Schritt 2.1 und 2.2

Falls die Kontext-Komponente von *LibreOffice* nicht antwortet, weil beispielsweise das Programm nicht gestartet ist, wird versucht *LibreOffice* zu starten. Wenn dies scheitert, wird ein Fehler geworfen. Für den Start von *LibreOffice* wird pauschal eine Wartezeit von einer Sekunde angesetzt. Wie lange *LibreOffice* tatsächlich braucht, hängt vom System ab und gegebenenfalls muss die Wartezeit angepasst werden. Anschließend wird Schritt 2 wiederholt. Wenn der Verbindungsaufbau bei dem zweiten Versuch scheitern sollte, wird ebenfalls ein Fehler geworfen.

Schritt 3

Um die Office-Datei in *LibreOffice* zu laden, wird die Klasse *Desktop* benötigt, die ebenfalls mit der Methode *createInstanceWithContext* erstellt wird. Über diese Klasse wird die Office-Datei in *LibreOffice* mit Hilfe der Klasse *SequenceInputStream* geladen. Dazu wird der Bytecode der Datei als *ByteSequence* der Klasse *SequenceInputStream* übergeben. Dafür wird von *UNO* die Klasse *ByteSquence* verwendet. Zurück erhält man das *XTextDocument* Interface. Über dieses Interface wird der komplette Inhalt des Textdokuments verfügbar.

Schritt 4

Anschließend wird *LibreOffice* angewiesen, die Office-Datei in XHTML umzuwandeln. Um den Filter des Rückgabeformates zu bestimmen, werden mit der Klasse *PropertyValue* die Eigenschaften zusammengestellt und *LibreOffice* zugesendet. Für die verschiedenen Textdokument-Formate gibt es eine Reihe von Filtern. Es können auch weitere Extensions in *LibreOffice* installiert werden, wie zum Beispiel das MediaWiki-Format. Für *Markdown* gibt es noch keinen Filter, daher ist es nötig, eigene Umwandler zu benutzen. Anschließend wandelt *LibreOffice* das geladene Textdokument entsprechend des Rückgabeformates um. In diesem Fall in XHTML. Die Abbildung A.2 zeigt ein Beispieldokument, das von *LibreOffice* in HTML konvertiert wurde.

Alle im Textdokument befindlichen Bilder werden von *LibreOffice* automatisch in *Base64* codiert und in das HTML eingefügt. Bilder werden im HTML über das *img*-Tag eingebunden. Üblicherweise wird das Bild in einer externen Datei abgespeichert und

dann über das *src*-Attribut im *img*-Tag referenziert. Es ist aber auch möglich, das Bild im *src* als *Base64*-Code einzubinden. Dies hat den Vorteil, dass das Bild nicht separat abgespeichert werden muss. Dafür wird der HTML-Code jedoch wesentlich länger. Die Länge des *Base64*-Codes hängt von der Größe des Bildes ab. Der Browser interpretiert den *Base64*-Code und stellt das Bild dar. Dies wird von allen gängigen Browsern und *Internet Explorer* ab Version 8 unterstützt.

Die Bilder können vor der Konvertierung in HTML aus dem Textdokument gelöscht werden. Dazu wird der Inhalt des Dokuments durchlaufen und jedes Bild entfernt. Der Inhalt wird über das *XTextDocument* Interface verfügbar. Statt die Bilder zu löschen, können sie auch im *png*-Format abgespeichert werden.

Es ist auch möglich das HTML temporär mit *LibreOffice* zu speichern. Dafür muss ein temporärer Ordner erstellt werden. Um einen temporären Ordner zu erzeugen, wird ein Verzeichnis mit Lese- und Schreibrechten benötigt. Dafür eignet sich am besten der betriebssystemeigene Temp-Ordner. Ein temporärer Ordner sollte mehrere Voraussetzungen erfüllen. Dieser Ordner muss einen eindeutigen Namen haben um verschiedenen Benutzern das parallele Hochladen und Verarbeiten von Dokumenten zu ermöglichen. Außerdem benötigt das Python-Skript die Lese- und Schreibrechte für diese Ordner. Schließlich muss dieser Ordner am Ende des Prozesses wieder gelöscht werden. Mit dem Python-Modul *tempfile* lassen sich eindeutige temporäre Ordner erzeugen. Diese befinden sich dann im spezifischen Temp-Ordner des Betriebssystems.

Schritt 5

Das zurück erhaltene HTML ist hauptsächlich dafür bestimmt, das Textdokument in einem Browser anzuzeigen. Da sich viele unnötige Tags und CSS-Angaben im HTML befinden, ist es sinnvoll, das HTML einmal zu durchlaufen und alle unnötigen Tags und CSS-Angaben zu entfernen. Für das Durchlaufen des HTMLs wird ein SAX Parser von der Bibliothek *lxml* verwendet. Damit die Informationen aus dem CSS nicht verloren gehen, wird das CSS mit der Bibliothek *cssutils* [3] geparst. Dabei werden nur die Eigenschaften "bold", für "Text fett markieren" und "italic", für "Text kursiv stellen", beachtet. Andere Eigenschaften, wie beispielsweise die Schriftart oder Schriftgröße, würden die einheitliche Darstellung der Webanwendung verändern und werden somit nicht beachtet.

Zusätzlich spielen diese Eigenschaften bei der Konvertierung in *Markdown* keine Rolle. Die beiden relevanten Eigenschaften im CSS werden in das äquivalente HTML-Tag umgewandelt. Syntaktische Fehler im HTML-Code werden durch das Reinigen nicht behoben. Die Abbildung A.3 zeigt das Beispieldokument nach dem Reinigen des HTMLs und ohne die herausgefilterten Kommentare.

Nachdem das HTML durchlaufen wurde, können sich leere Tags im HTML befinden, die zumeist auch noch verschachtelt sind. Diese Tags vergrößern unnötig den HTML-Code und verursachen oft Fehler bei der späteren Konvertierung in Markdown. Diese leeren Tags entstehen beispielsweise, wenn im Textdokument ein Leerzeichen "fett" markiert wurde. Um diese Tags zu löschen, wird ein regulärer Ausdruck verwendet. Da die leeren Tags verschachtelt sein können, wird das HTML in einer Schleife so lange durchlaufen, bis alle leeren Tags mit dem regulären Ausdruck gefunden und anschließend gelöscht wurden. Zum Schluss werden noch die Leerzeichen zwischen Text und Tag vor oder hinter das Tag verschoben, um Fehler bei der Konvertierung in Markdown zu verhindern.

```
<!-- Vor dem Reinigen -->
<html><head>
  <style type="text/css">
  #wrap { font-weight:bold; font-style:italic; }
  </style>
  </head>
  <body>
    <div><p id="wrap"> Text fett und kursiv</p></div>
    <span> </span>
  </body>
</html>

<!-- Nachdem dem Reinigen -->
<div> <p><b><i>Text fett und kursiv</b></i></p></div>
```

Listing 4.1: Vor und nach dem Reinigen des HTMLs

Die Kommentare eines Textdokuments werden im HTML-Code gekennzeichnet. Dafür wird ein *title*-Attribut eines *span*-Tag von werden *LibreOffice* zweckentfremdet und das Wort "annotation" in das Attribut eingefügt. Der Kommentar im HTML befindet sich ungefähr auf der Höhe wo er im Text eingefügt wurde. Allerdings geht die Referen-

zierung auf einzelne Worte oder Textpassagen verloren. Die Kommentare werden beim Durchlaufen des HTMLs gesondert behandelt und aus dem HTML herausgefiltert. Wie auf der Abbildung A.2 zu sehen ist, fügt *LibreOffice* vor dem eigentlichen Kommentar einen Text ein. Diese Informationen werden nicht benötigt und mittels eines regulären Ausdrucks entfernt.

Schritt 6

Das gefilterte HTML wird mit dem Skript *html2text* in *Markdown* umgewandelt. Alle heraus gefilterten Kommentare müssen separat in *Markdown* umgewandelt werden.

4.1.2. Verschiedene XML-Parser

Durch die XML-Parser können die zwei Dokument-Formate *Office Open XML* Format (.docx) und *Open Document* Format (.odt) umgewandelt werden. Dafür werden beide Formate mit der Bibliothek zipfile geöffnet und die entsprechende XML-Datei eingelesen. Die entsprechenden XML-Dateien sind beim *Office Open XML* Format die *document.xml* im Verzeichnis *word* und beim *Open Document* Format die *content.xml* im Hauptverzeichnis. Alle XML-Parser unterstützen folgende Formatierungen: Fett, kursiv, Listen, Links und Überschriften. Das XML kann mit zwei verschiedenen XML-Parsern in *Markdown* umgewandelt werden. Es wurden jeweils ein SAX Parser und ein DOM Parser programmiert.

SAX Parser

Für die Implementierung des SAX Parsers wurde die *lxml* Bibliothek verwendet. Der SAX Parser durchläuft das XML und ruft bei den Events "ein Tag wird geöffnet", "ein Tag wird geschlossen" und "Text wird eingelesen" Callbackfunktionen auf. In der Funktion, die beim Tag öffnen aufgerufen wird, werden die Attribute des Tag eingelesen und analysiert. Damit soll ermittelt werden, ob es sich hier zum Beispiel um eine Überschrift handelt oder der Text fett markiert werden soll. Sobald ein Teil des Textes eingelesen

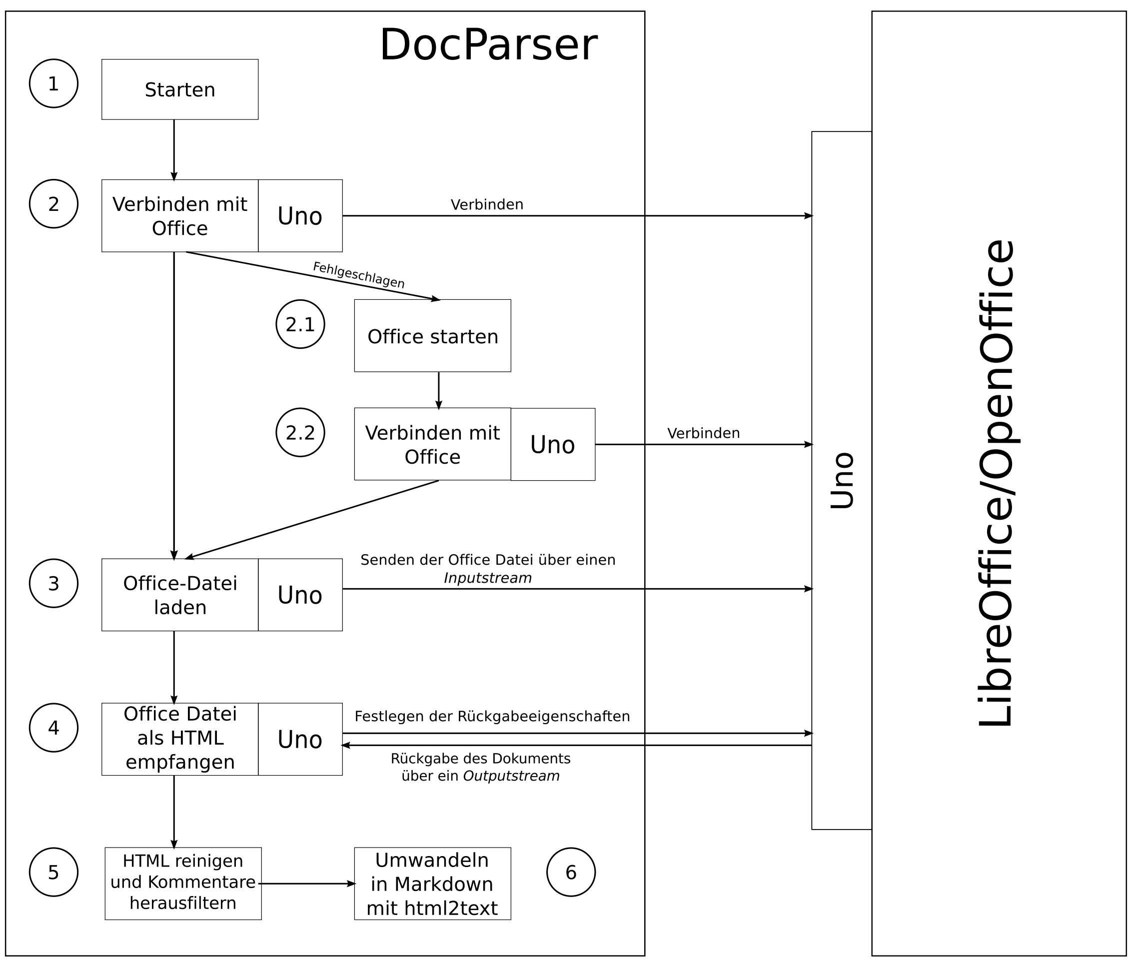

Abbildung 4.1.: Der schrittweise Ablauf des DocParsers beim Einlesen und Konvertieren eines Office Dokuments

wird, muss dieser entsprechend der analysierten Attribute und der Art des Tags, in Markdown umgewandelt werden.

Um den Text des Dokumenten-Formats ".docx" zu parsen, müssen vorher die Informationen über die Formatierung des Textes eingelesen werden. Diese befinden sich im Hauptverzeichnis des Containers in der *style.xml*-Datei. Dazu wird auch ein SAX Parser verwendet. Für die Hyperlinks muss die *document.xml.rels*-Datei eingelesen werden. Dazu wird ein DOM Parser verwendet. Bei diesem ist die Länge des Quellcodes kürzer und verbraucht demnach bei dem Einlesen weniger Zeit.

DOM Parser

Das Parsen des XML mit dem DOM Parser wurde ebenfalls mit der *lxml* Bibliothek realisiert. Der DOM Parser erstellt eine Baumstruktur aus dem geparsten XML. Diese Baumstruktur wird dann in einer Schleife durchlaufen. Auch in diesem Fall werden die Attribute der Tags analysiert und der Text dementsprechend in *Markdown* konvertiert.

Bei dem Dokumenten-Format ".docx" muss zunächst die *document.xml.rels*-Datei im *word/rels*-Ordner eingelesen werden. Dort befinden sich die Hyperlinks, die über eine ID im Text verknüpft sind. Danach wird die *document.xml* geparst und in *Markdown* umgewandelt.

Für das Dokumenten-Format ".odt" werden als erstes alle Informationen über die Formatierung am Anfang des XMLs geparst. Diese Information befinden sich in der *content.xml*-Datei. Anschließend werden die Texte in der gleichen Datei geparst und entsprechend der vorher erhaltenen Informationen in *Markdown* umgewandelt.

4.2. Integration in die Python-Webanwendung

4.2.1. Controller

Sobald das Formular mit der Office-Datei abgeschickt wird, werden zunächst alle erforderlichen Formularfelder validiert. Die Datei wird über die Klasse *FieldStorage* verfügbar. Diese liefert direkt ein *File* Objekt, über den man den Inhalt der Datei als bytestrings erhält. Der Inhalt der Datei wird direkt an den DocParser zur Konvertierung übergeben. Die komplette Konvertierung findet im DocParser statt. Anschließend wird der Text und alle weiteren erforderlichen Informationen in der Datenbank gespeichert. Da *adhocracy* auch bisher *Markdown*-Eingaben erlaubt, ist keine Änderung des rendering-Codes nötig. Die Abbildung 4.2 zeigt die realisierte Funktionalität von *adhocracy*. Momentan muss immer eine Benutzereingabe in das Textfeld erfolgen, auch wenn der eigentliche Text über eine Office-Datei hinzugefügt wird. An dieser Stelle ist die Integration des DocParsers in *adhocracy* noch nicht komplett.

4.2.2. Validation einer Office-Datei

Benutzereingaben sollten immer überprüft werden, um die Integrität der Webanwendung nicht zu gefährden. In diesem Fall sollte mindestens die Größe und der Typ der Office-Datei bei dem Hochladen überprüft werden. Für die Validation der hochgeladenen Office-Datei wird die Komponente *formencode* benutzt. Diese Komponente bietet schon einige Validatoren an, wie z.B. *MaxLength* zur Überprüfung der Länge eines Strings. Um den Dateityp und die Dateigröße zu überprüfen, findet sich kein passender Validator unter den von *formencode* zu Verfügung gestellten Validatoren. Die Komponente *formencode* bietet aber an, einen eigenen Validator zu entwickeln. Dies ermöglicht eine optimale Einbindung in Python-Webanwendung *Pylons*.

Dafür wird eine eigene Validator-Klasse geschrieben und diese von der Klasse *FancyValidator* von *formencode* abgeleitet. Die Methode "validate_python" wird überschrieben. Diese überprüft den Dateityp mittels der Bibliothek *mimetype* und die Größe der Office-Datei. Falls eine der beiden Bedingungen nicht erfüllt ist, wird ein Fehler geworfen mit der Komponente *formencode*. Die Office-Datei wird daraufhin nicht hochgeladen und in der Webanwendung ein Fehler, entsprechend der Fehlerursache, angezeigt.

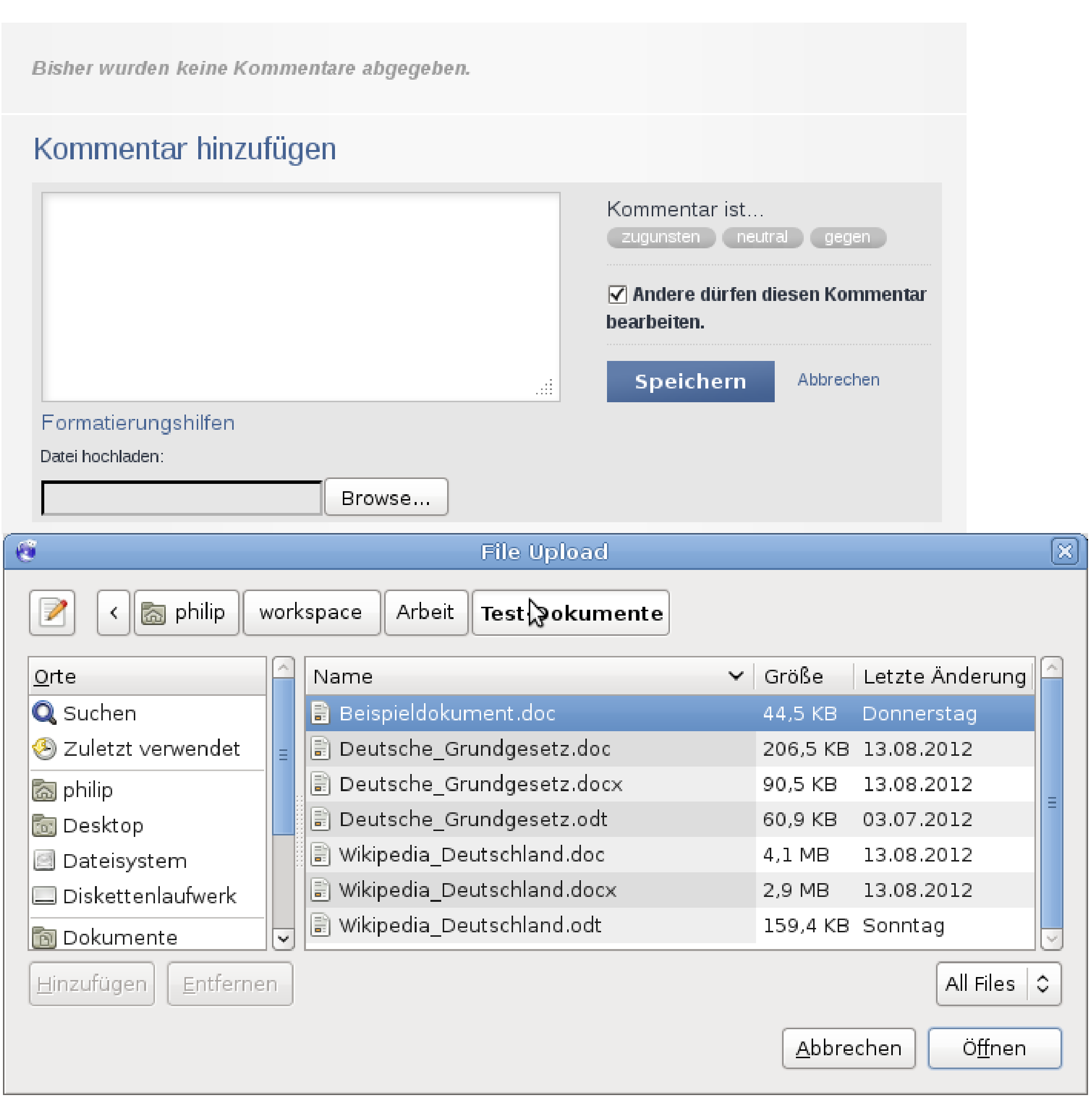

Abbildung 4.2.: Die Hochlademaske eines Kommentars von *adhocracy*

Kapitel 5.

Evaluation

Dieses Kapitel befasst sich mit der Durchführung verschiedener Benchmarks und anschließend mit der Auswertung und Interpretation der erhaltenen Messwerte. Die Benchmarks wurden auf einer virtuellen Maschine (*VMware* 4.0.4) mit dem Betriebsystem *Debian* 6.0.5 (Kernel 2.6.32-5-amd64) und vier Intel Core i7 CPU Q720 1,60 GHz und 2 GB Arbeitsspeicher durchgeführt. Die *VMware* läuft auf dem Betriebssystem Windows 7 (64 Bit) mit 4 GB Arbeitsspeicher. In den Tabellen wird jeweils der Median von 100 Einzelmessungen dargestellt. Es wurde der Median gewählt, da dieser robuster auf Schwankungen reagiert. Als Test-Dokumente wurden das Deutsche Grundgesetz mit ca. 58 Seiten und der Wikipedia-Eintrag für Deutschland mit ca. 77 Seiten verwendet. Das Deutsche Grundgesetz enthält nur wenige Formatierungen und keine Bilder, wohingegen der Wikipedia-Eintrag für Deutschland eine Vielzahl an Bildern und Formatierungen beinhaltet. Diese Dokumente stehen in den Tests repräsentativ für alle Dokumente bezüglich der Umwandlungsmöglichkeiten der verwendeten Programme. Beide Test-Dokumente wurden mit den drei Dokument-Formaten .doc, .docx und .odt getestet. Als Textverarbeitungsprogramm wurde *LibreOffice* mit der Version 3.4.6 verwendet. Für die XML-Parser wird die Bibliothek *lxml* der Version 2.3.4 verwendet. Alle Messungen und Profile finden Sie auf der beigefügten CD-ROM unter *Arbeit/Benchmarks/*.

5.1. Durchführung der Benchmarks

Das Python Skript *benchmark.py* führt die verschiedenen Tests durch. Um die Messwerte zu erhalten wurde das *time* Kommando unter Linux verwendet. In den Messwerten ist die Zeit und der maximale Speicherverbrauch inbegriffen. Für die Profile wurde die Python Bibliothek *cProfile* verwendet.

Als erstes wird *LibreOffice* gestartet. Anschließend wird der DocParser als Prozess in einer Schleife hundert mal mit einem der Test-Dokumente aufgerufen. In jedem Durchlauf wandelt der DocParser mit dem Programm *LibreOffice* einmal das Test-Dokument in HTML um. Sobald der DocParser Prozess beendet ist, werden die vom Kommando *time* erzeugten Messwerte in einer csv-Datei gespeichert. Nachdem alle Einzelmessungen durchgeführt wurden, wird der Prozess von *LibreOffice* beendet und die erzeugten Messwerte ebenfalls in eine csv-Datei gespeichert. Da *LibreOffice* am Anfang einmal gestartet wird, wird nur die Gesamtzeit und der maximale Speicherverbrauch für alle 100 Messungen ermittelt.

Für den Benchmark des Skriptes *html2text* wird vorerst das Test-Dokument mit dem DocParser in HTML umgewandelt und in eine Datei gespeichert. Anschließend wird das Skript *html2text* gestartet. Ab diesem Zeitpunkt beginnt die Messung. Das Skript wandelt dann das HTML aus der Datei in *Markdown* um.

Als letztes Szenario wird die Zeit und der Speicherverbrauch der vier verschiedenen XML-Parser gemessen. Dazu wird der DocParser mit den Test-Dokumenten der beiden Dokument-Formate Docx und Odt gestartet.

5.2. Ergebnisse und Auswertung

Bei dem Versuch, das Text-Dokument "Wikipedia_Deutschland.docx" in HTML zu konvertieren, stürzte *LibreOffice* ab. Ein Versuch unter Windows 7 mit der *LibreOffice* 3.6.0.4 scheiterte ebenfalls. Nachdem alle Bilder aus dem Dokument entfernt wurden, liess sich das Text-Dokument in HTML konvertieren. Das legt nahe, dass *LibreOffice* Schwierig-

keiten hat die Bilder im Dokumenten-Format *Office Open XML* zu verarbeiten. Dabei traten diese Fehler nur auf, wenn es sich bei den Bildern um Vektorgrafiken (.svg) handelte. Bei Rastergrafiken (.jpg, .png, .gif) gab es keine Probleme.

Die Konvertierung des Test-Dokuments "Wikipedia_Deutschland.odt" war ebenfalls nicht problemlos. Ab der fünfzigsten Messung sind die Messwerte sehr unbeständig, da oftmals die binäre *URP*-Brücke zusammengebrochen ist. *LibreOffice* verbrauchte zu diesem Zeitpunkt sehr viel Arbeitsspeicher und kam somit an die Leistungsgrenze des Betriebssystems.

Bei der Gegenüberstellung der beiden Test-Dokumente ist an der Abbildung 5.1 deutlich erkennbar, dass das Test-Dokument "Wikipedia_Deutschland" bei der Konvertierung wesentlich mehr Zeit in Anspruch nimmt. Da dieses Dokument viele Bilder beinhaltet und der Text stark formatiert ist, benötigt *LibreOffice* mehr Zeit das Dokument zu laden und zu konvertieren. Dies bedeutet, dass die Dauer der Konvertierung stark von der Anzahl der Bilder, Kommentare und der Menge der Formatierungen im Text abhängt. Bei dem Vergleich der drei hier vorgestellten Methoden wird deutlich, dass

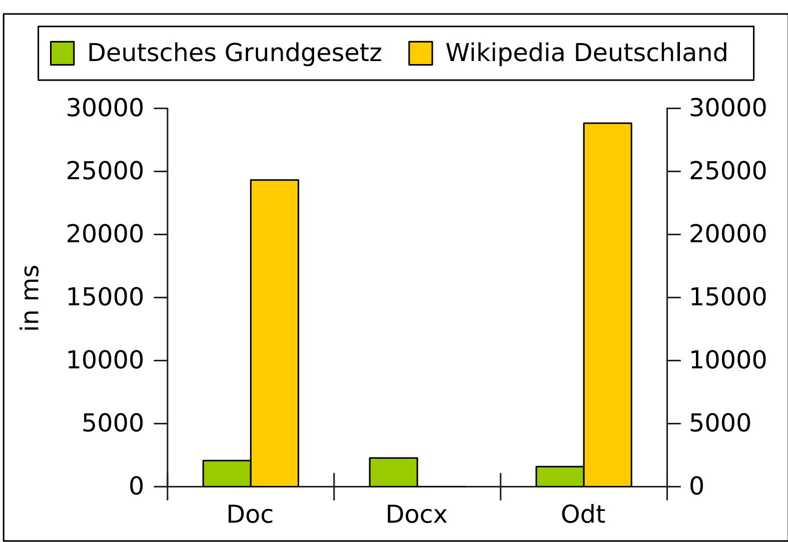

Abbildung 5.1.: Gegenüberstellung der beiden Test-Dokumente bei der Konvertierung mit *LibreOffice* in HTML

die XML-Parser wesentlich schneller sind als *LibreOffice*. Dabei ist zu beachten, dass die XML-Parser in der momentanen Implementierung keine Bilder und Kommentare

unterstützen. Dadurch sind die XML-Parser etwas im Vorteil. Um einen besseren Vergleich zwischen den XML-Parser und *LibreOffice* zu finden, wäre es notwendig beide Methoden auf den gleichen Stand zu bringen. Weiterhin fällt auf, dass der SAX Parser und der DOM Parser gleich viel Zeit und Speicher verbrauchen. Dies bedeutet, dass das sequentielle Durchlaufen des HTMLs genauso lange dauert, wie das Erzeugen einer Baumstruktur vom HTML und das anschließende Durchlaufen. Bei Betrachtung der Profile der beiden XML-Parser, bei der Konvertierung des Test-Dokumentes "Wikipedia_Deutschland.docx", fällt auf, dass der SAX Parser circa doppelt soviel Funktionsaufrufe hat, wie der DOM Parser. Die Callbackfunktion für das Öffnen eines Tags wird 77906 mal aufgerufen. Diese hohe Zahl der Funktionsaufrufe ist auch damit begründet, dass der SAX Parser immer das komplette HTML durchlaufen muss. Die Abbildung 5.2 zeigt das dazu gehörige Balkendiagramm. Die Abbildung 5.3 zeigt in einem Kreisdia-

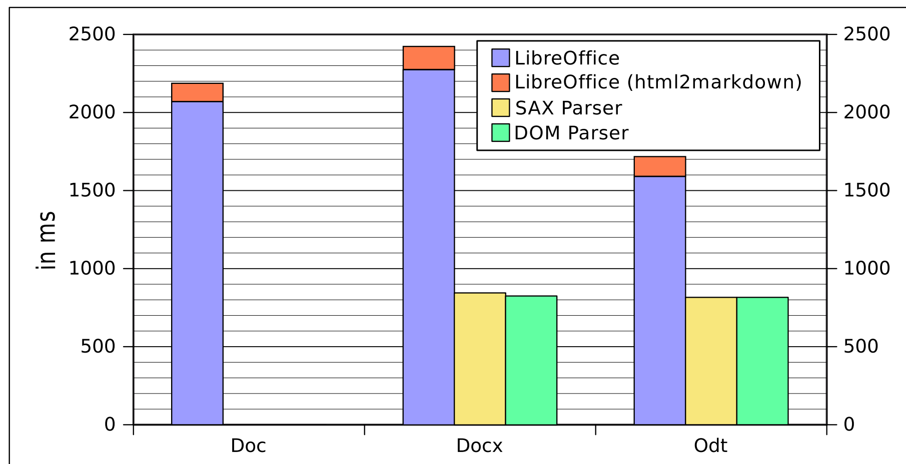

Abbildung 5.2.: Der Zeitverbrauch bei der Konvertierung des Test-Dokuments "Wikipedia_Deutschland.docx"

gramm die zeitliche Verteilung des Konvertierungsprozesses mit *LibreOffice* des Test-Dokumentes "Wikipedia_Deutschland.doc". Die meiste Zeit benötigt *LibreOffice* für das Laden und Konvertieren. Dieser Teil des Prozesses könnte möglicherweise verbessert werden, indem man eine eigene *UNO*-Komponente entwickelt, die das Dokument konvertiert. Ansonsten kann dieser Teil nicht entscheidend beeinflusst werden. Die Zeit für das Reinigen des HTMLs könnte möglicherweise noch durch Optimierung des Quellcodes verbessert werden. Es könnte beispielsweise ein DOM Parser anstatt eines SAX Parser verwendet werden, da dieser nur den Textteil durchlaufen kann. Des Weitern fal-

len die vielen Funktionsaufrufe des SAX Parsers weg. Bei Betrachtung der Messwerte

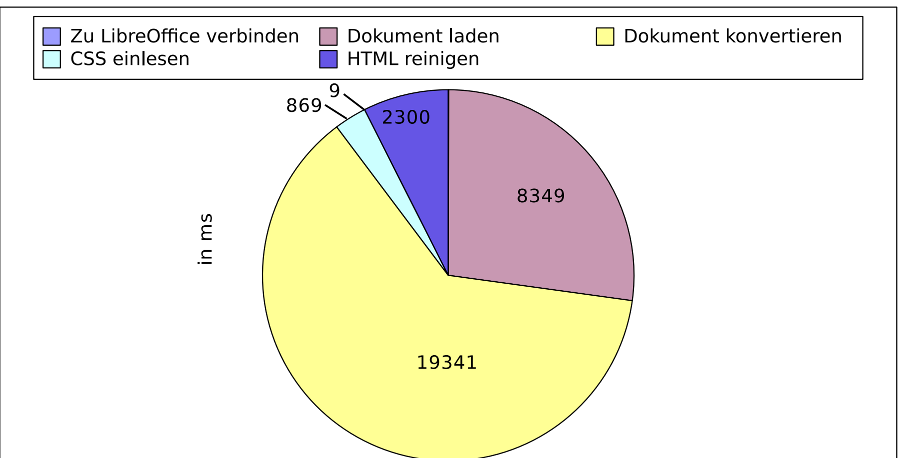

Abbildung 5.3.: Die Verteilung des Zeitverbrauchs bei der Konvertierung des Test-Dokuments "Wikipedia_Deutschland.doc" in HTML mit *LibreOffice*

aus den Tabellen 5.1 und 5.2 fällt auf, dass *LibreOffice* für das Dokumenten-Format ".doc" fast gleich viel oder weniger Zeit benötigt, als für die XML-basierten Formate. Es ist davon auszugehen, dass das Einlesen einer binären Datei etwas schneller ist, als bei den Dokument-Formaten, die auf XML-basieren. Bei den XML-basierten Dokument-Formaten müssen eine ganze Reihe von Dateien eingelesen werden.

Bei dem Vergleich des Speicherverbrauchs aus Tabelle 5.1 fällt auf, dass das Dokument-Format ".docx" dreimal so viel Speicher benötigt. Dies deutet darauf hin, dass *LibreOffice* noch erhebliche Probleme hat dieses Dokumenten-Format effizient zu verarbeiten. Wenn jetzt die verfügbaren Messwerte aus der Tabelle 5.2 in die Beobachtung mit einbezogen werden, ist zu erkennen, dass der Speicherverbrauch von *LibreOffice* signifikant ansteigt.

<table>
<tr><th>Dokument Format</th><th>Methode</th><th>Format</th><th>real Zeit in ms</th><th colspan="2">Maximaler Speicherverbrauch in KB</th></tr>
<tr><td rowspan="3">Doc</td><td rowspan="2">LibreOffice</td><td rowspan="2">HTML</td><td rowspan="2">1890</td><td>DocParser</td><td>LibreOffice</td></tr>
<tr><td>156208</td><td>625824</td></tr>
<tr><td>html2text</td><td>Markdown</td><td>120</td><td colspan="2">37664</td></tr>
<tr><td rowspan="5">Docx</td><td rowspan="2">LibreOffice</td><td rowspan="2">HTML</td><td rowspan="2">2190</td><td>DocParser</td><td>LibreOffice</td></tr>
<tr><td>157624</td><td>2351408</td></tr>
<tr><td>html2text</td><td rowspan="3">Markdown</td><td>160</td><td colspan="2">37840</td></tr>
<tr><td>SAX Parser</td><td>860</td><td colspan="2">146320</td></tr>
<tr><td>DOM Parser</td><td>840</td><td colspan="2">146320</td></tr>
<tr><td rowspan="5">Odt</td><td rowspan="2">LibreOffice</td><td rowspan="2">HTML</td><td rowspan="2">1590</td><td>DocParser</td><td>LibreOffice</td></tr>
<tr><td>157664</td><td>592784</td></tr>
<tr><td>html2text</td><td rowspan="3">Markdown</td><td>130</td><td colspan="2">37760</td></tr>
<tr><td>SAX Parser</td><td>830</td><td colspan="2">146320</td></tr>
<tr><td>DOM Parser</td><td>835</td><td colspan="2">146320</td></tr>
</table>

Tabelle 5.1.: Benchmark für das Einlesen und Umwandeln des Deutschen Grundgesetzes (ca. 58 Seiten)

<table>
<tr><th>Dokument Format</th><th>Methode</th><th>Format</th><th>real Zeit in ms</th><th colspan="2">Maximaler Speicherverbrauch in KB</th></tr>
<tr><td rowspan="3">Doc</td><td rowspan="2">LibreOffice</td><td rowspan="2">HTML</td><td rowspan="2">24320</td><td>DocParser</td><td>LibreOffice</td></tr>
<tr><td>494160</td><td>2400976</td></tr>
<tr><td>html2text</td><td>Markdown</td><td>845</td><td colspan="2">265952</td></tr>
<tr><td rowspan="5">Docx</td><td rowspan="2">LibreOffice</td><td rowspan="2">HTML</td><td rowspan="2">—</td><td>DocParser</td><td>LibreOffice</td></tr>
<tr><td>—</td><td>—</td></tr>
<tr><td>html2text</td><td rowspan="3">Markdown</td><td>—</td><td colspan="2">—</td></tr>
<tr><td>SAX Parser</td><td>830</td><td colspan="2">146320</td></tr>
<tr><td>DOM Parser</td><td>835</td><td colspan="2">146320</td></tr>
<tr><td rowspan="5">Odt</td><td rowspan="2">LibreOffice</td><td rowspan="2">HTML</td><td rowspan="2">28825</td><td>DocParser</td><td>LibreOffice</td></tr>
<tr><td>671096</td><td>7085952</td></tr>
<tr><td>html2text</td><td rowspan="3">Markdown</td><td>480</td><td colspan="2">244272</td></tr>
<tr><td>SAX Parser</td><td>840</td><td colspan="2">146320</td></tr>
<tr><td>DOM Parser</td><td>830</td><td colspan="2">146320</td></tr>
</table>

Tabelle 5.2.: Benchmark für das Einlesen und Umwandeln des Wikipediaeintrages "Deutschland" (ca. 77 Seiten)

Kapitel 6.

Sicherheit

In diesem Kapitel wird auf die Sicherheit beim Import von Office-Dokumenten der Webanwendung eingegangen. Dies ist ein neue Methode, Text in das System einzugeben. Vorher wurde der Text ausschließlich durch den Benutzer in ein Textfeld eingegeben. Durch die Formatierung in *Markdown* werden Angriffe auf die Webanwendung, wie beispielsweise *Cross-Site-Scripting* verhindert. Da die Office-Dokumente vorher in *Markdown* formatiert werden, bevor sie abgespeichert werden, muss sich nicht gesondert um die Integrität der Webanwendung gekümmert werden. Hier ist die Sicherheit von *LibreOffice* während der Konvertierung des Office-Dokumentes in HTML wichtig.

CVE (Common Vulnerabilities and Exposures) [25] ist ein Verzeichnis für öffentlich bekannte Sicherheitslücken in Computersystemen. Laut *CVE* [1] weist *LibreOffice* insgesamt sechs Sicherheitslücken auf. Davon ermöglichen vier dem Angreifer Schadcode auf den Server einzuschleusen und auszuführen. Dies bedeutet, dass *LibreOffice* immer auf dem aktuellsten Stand gehalten werden muss und auf dem Server abgesichert werden sollte. Ein Angreifer, der beliebigen Code im Kontext des *LibreOffice*-Prozesses ausführen kann, sollte keine oder nur sehr begrenzte Zugriffsrechte haben, um den Server und die Daten der Benutzer zu schützen.

Es gibt mehrere Möglichkeiten ein Programm auf einem Linux Server abzusichern. Um

[1] Genauere Informationen über die Sicherheitslücken werden unter http://www.cvedetails.com/product/21008/Libreoffice-Libreoffice.html?vendor_id=11439 dargestellt.

die Rechte des Programms einzuschränken wäre es sinnvoll, das Programm unter einem eigenen Benutzer laufen zu lassen. Des Weiteren kann man das Programm mit *chroot* in eine Art "Gefängnis" oder *Sandbox* sperren. Dabei wird das Root-Verzeichnis eines Prozesses auf ein anderes Verzeichnis gesetzt. Das Programm erhält nur auf das umgeleitete Verzeichnis Zugriffsrechte. Dies schränkt die Bewegungen eines Angreifers ein. Allerdings ist es möglich als "root"-Benutzer aus solch einer *chroot*-Umgebung wieder "auszubrechen".

Linux Container geben eine weitere Möglichkeit ein Programm abzusichern. Diese bieten virtuelle Umgebungen für Linux Distributionen oder einzelne Programme unter Linux. Die Ressourcen des Systems lassen sich beliebig weit einschränken. Für solche virtellen Umgebungen können zum Beispiel *OpenVZ* oder *Linux Container (lxc)* verwendet werden.

Die Python Webanwedung hat die Möglichkeit, über Python-UNO mit *LibreOffice* über das lokale Netzwerk oder das Internet zu kommunizieren. Es ist denkbar, *LibreOffice* auf einem eigenem Server laufen zu lassen. Sollte ein Angreifer es schaffen, diesen Server zu übernehmen oder "lahmzulegen", so bleibt die Webanwendung davon unberührt. Dadurch sind die Daten der Benutzer besser geschützt.

Kapitel 7.

Resümee und Ausblick

7.1. Resümee

Im Rahmen dieser Arbeit wurde die Möglichkeit geschaffen, ein Office-Dokument einzulesen und in *Markdown* zu konvertieren. Es werden die drei Dokument-Formate *Office Open XML* (.docx), *Open Document* (.odt) und das binäre Format *.doc* unterstützt. Für die Konvertierung wurden mehrere Bibliotheken und Skripte vorgestellt und zwei Lösungsansätze implementiert. Einer davon ist die Kombination aus dem Textverarbeitungsprogramm *LibreOffice*, der Bibliothek Python-UNO und dem Skript *html2text* und der andere sind die vier XML-Parser für zwei unterstützte Dokument-Formate. Für beide Lösungen wurde das Skript "DocParser" implementiert.

Bei den Tests traten einige Fehler bei dem Ansatz mit *LibreOffice* auf. Ein Test-Dokument liess sich nicht konvertieren. Häufig scheiterte es bei der Verarbeitung von Bildern. Dazu kommt, dass die Performance teilweise schlecht war und die Testumgebung bei der Konvertierung zum Teil überlastet wurde. Allerdings unterstützt nur *LibreOffice* alle Dokument-Formate, solange es sich um reine Textdokumente handelt.

Die Performance der XML-Parser ist wesentlich besser, besonders im Speicherverbrauch. Die XML-Parser erzeugen aber kein einheitliches Markdown und unterstützen momentan keine Bilder oder Kommentare. Dies müsste noch implementiert werden. Die Um-

setzung ist zur Zeit noch rudimentär und dient nur als Beispiel und Vergleich.

Der DocParser wurde an zwei Stellen in die Webanwendung *adhocracy* integriert. Es ist dadurch möglich den Text eines Kommentares oder Vorschlages über eine Datei in den Formaten ".docx", ".odt" oder ".doc" hinzuzufügen. Die Kommentare zu einem Office-Dokument werden mit dem DocParser herausgefiltert, aber noch nicht separat in *adhocracy* abgespeichert.

7.2. Ausblick

Der Import eines Dokuments kann an manchen Stellen noch benutzerfreundlicher gestaltet werden. Da das Hochladen auf Grund der Größe des Office-Dokuments und der geringen Uploadgeschwindigkeit meist länger dauert, sollte man den Prozess komfortabler gestalten. Beispielsweise könnte dies über *JavaScript* und *AJAX* realisiert werden. Dabei kann ein Ladebalken angezeigt werden, um dem Benutzer zu signalisieren, dass der Prozess noch nicht beendet ist. Bei dieser Methode sollte beachtet werden, dass dieser Vorgang auch ohne *JavaScript* und *AJAX* funktionieren sollte. *JavaScript* wird auf dem Client ausgeführt und kann vom Benutzer abgeschaltet werden. Im Falle einer Abschaltung fällt sonst die komplette Funktion aus.

Anhang A.

Zusätzliche Grafiken

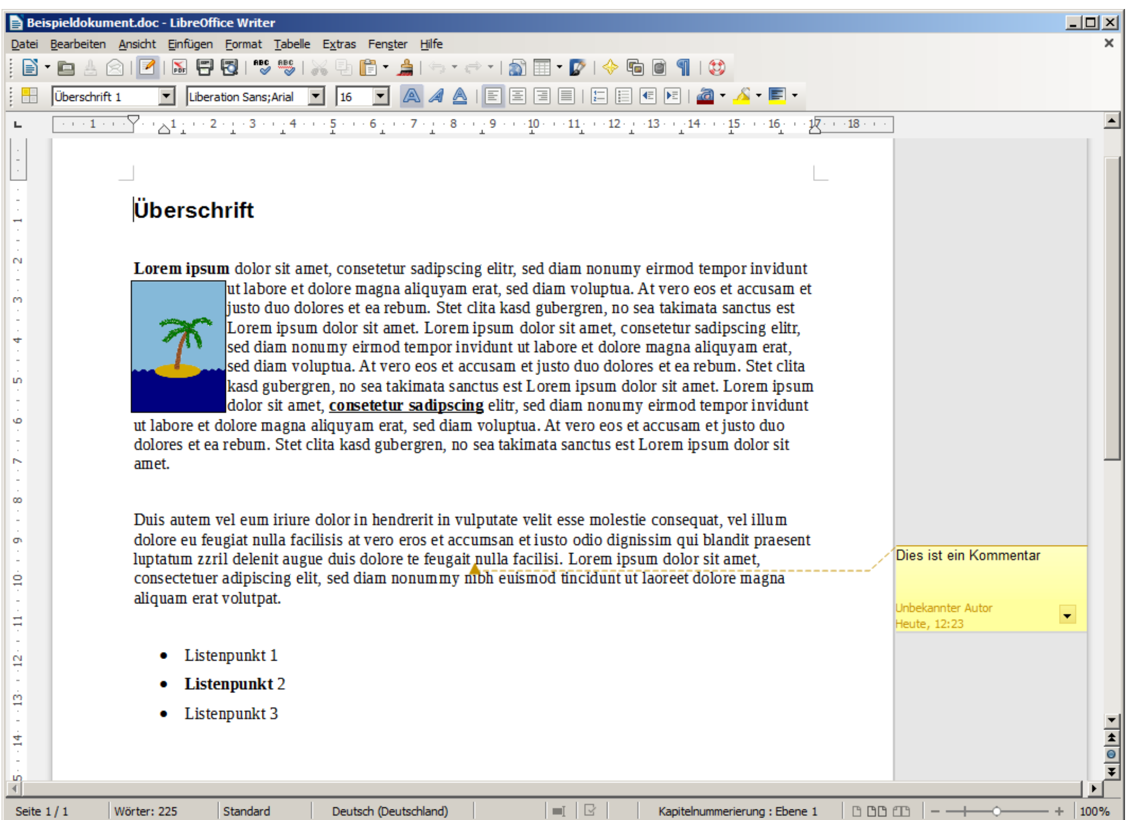

Abbildung A.1.: Darstellung des Beispieldokuments im Programm *LibreOffice*

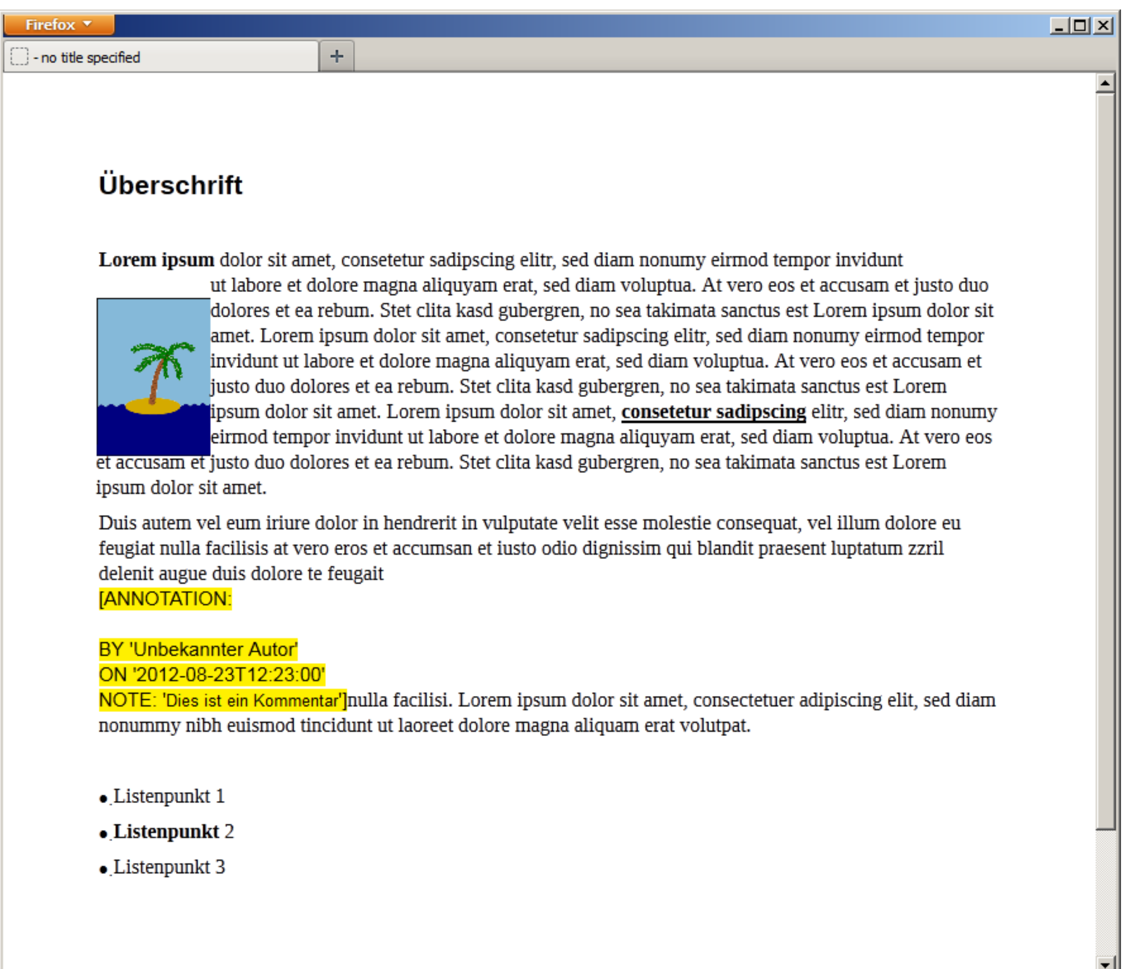

Abbildung A.2.: Darstellung des Beispieldokuments im Browser *Mozilla Firefox* vor der Reinigung des HTMLs durch den DocParser

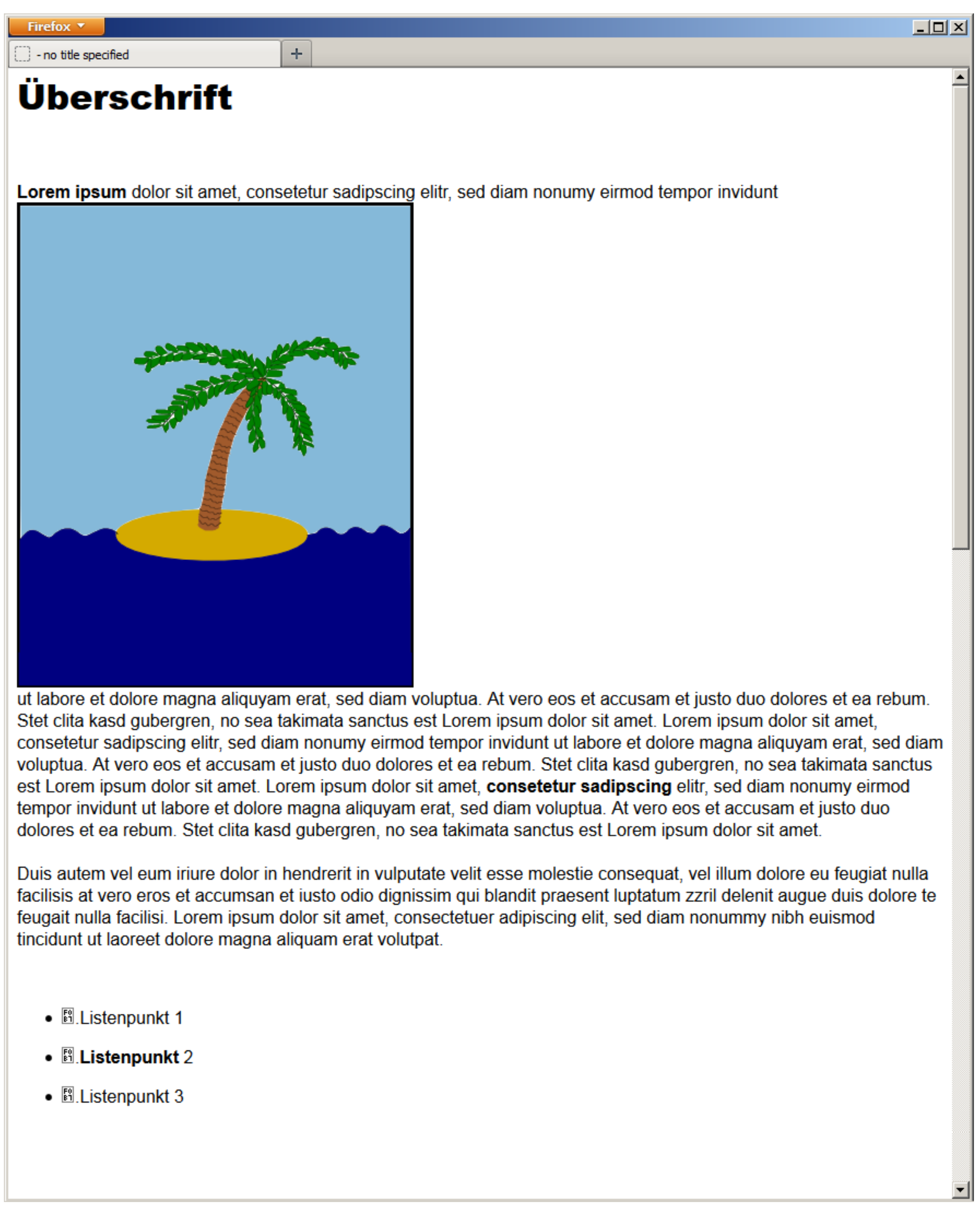

Abbildung A.3.: Darstellung des Beispieldokuments im Browser *Mozilla Firefox* nach der Reinigung des HTMLs durch den DocParser

Ehrenwörtliche Erklärung

Hiermit versichere ich, die vorliegende Bachelorarbeit selbstständig verfasst und keine anderen als die angegebenen Quellen und Hilfsmittel benutzt zu haben. Alle Stellen, die aus den Quellen entnommen wurden, sind als solche kenntlich gemacht worden. Diese Arbeit hat in gleicher oder ähnlicher Form noch keiner Prüfungsbehörde vorgelegen.

Düsseldorf, 31. August 2012 Eike Schlingensief