



Programmentwurf 1

Aufgabenteil E_002

Dokumentation für Webengineering im Studiengang Angewandte Informatik
an der Dualen Hochschule Baden-Württemberg Stuttgart

von

Christian Burkard & Florian Kanngiesser

Juni 2012

Matrikelnummer, Kurs
Ausbildungsfirma
Betreuerin

4206853 & 7353550, TITAlA2010
,
Prof. Dr. Dirk Reichardt

Inhaltsverzeichnis

1	Einführung	1
1.1	Input File Format	1
2	Verfahren	2
2.1	Dempster-Shafer Theorem	2
3	Java Server Pages	3
3.1	Seitenfluss	3
3.2	Die Edit.jsp Seite	4
3.3	Das verwenden eines Controllers	6
3.4	MVC-Pattern des Kochbuchs	7
4	Bilderupload	9
5	Appendix	i
	Abbildungsverzeichnis	ii
	Quellcodeverzeichnis	iii

1 Einführung

Anders als im Readme beschrieben haben wir uns für einen einzigen File als Input entschieden. Der Gründe dafür waren das im ersten Beispiel nur eine einzige Datei vorlag und wir deshalb unseren ursprünglichen Entwurf darauf ausgelegt hatten. Außerdem spielt es auch keine Rolle ob man ein oder zwei Dateien verwendet.

1.1 Input File Format

Das Input-File ist in 2 Sektionen unterteilt, welche mittels Semikolon voneinander getrennt sind. Die einzelnen Attribute werden mittels Komma separiert. Aus diesem Format sind wir nun in der Lage Primitive und Relationen zu generieren. Zuerst werden die Primitive ausgelesen und in eine geeignete Datenstruktur gespeichert, anschließend die Relationen.

```
1 re;13;0.1
2 ra,9,0.95
3 ra,10,0.9
4 ra,8,0.4
5 ;
6 inside,2,1
7 inside,12,5
8 inside,7,6
```

2 Verfahren

In diesem Kapitel wird das verwendete Verfahren noch einmal theoretisch beleuchtet.

2.1 Dempster-Shafer Theorem

Während des Auslesens des Input Files wird ein HashTable der einzelnen Primitiven gebildet. Sollte über ein Primitiv mehrere Aussagen vorliegen so wird nun zuerst die Gesamtevidenz über dieses Primitiv gebildet. Die Ausgangslage zur Evidenzbestimmung ist die Zuverlässigkeit der Aussage über ein Primitiv (Basismass). Nun wird für jede "inside"Relation die Evidenz mittels der Dempster-Shafer Regel zusammengefasst, also für Objekte mit mehreren Aussagen die Gesamtevidenz und für einzelne die gegebene Zuverlässigkeit.

Sollte es dabei zu einem Konflikt kommen wird zusätzlich noch ein Korrekturfaktor bestimmt. Anschließend wird der Believe über die verschiedenen Straßenschilder gebildet, Grundlage dafür sind die in der Aufgabenstellung beschriebenen Straßenschilder und die im Input File gegebenen Relationen.

Anschließend wird noch bestimmt ob gewisse Schilder übereinander ("ontop")liegen können. Dazu werden die ontop Relationen geprüft und die Objekte die bereits als Schild identifiziert wurden zusammen genommen. Mittels Dempster-Shafer Regel wird nun die Evidenz für die Relation bestimmt und falls vorhanden die Relation ausgegeben, also welches Schild über welchem Schild liegt. Abschließend wird die Plausibilität, ob wir ein Schild finden können zusammen mit dem Believe das wir ein Schild gefunden haben ausgegeben.

3 Java Server Pages

Die serverseitige Programmierung wird durch JSPs realisiert. Dadurch ergeben sich zahlreiche Vorteile, wie das plattformunabhängige Hosting von Websites, Kosten- und Zeitersparnis beim Programmieren und Pflegen des Webauftritts, da Java und dadurch JSP eine weitverbreitete und dadurch bestens dokumentierte und unterstützte Sprache ist.

3.1 Seitenfluss

In Abbildung 3.1 ist der Programmfluss der Anwendung schematisch dargestellt. *Start* markiert die JSP, die beim Starten der Anwendung geladen wird. Die StartJSP der Anwendung wird in der *web.xml*, die sich im Verzeichnis *"/WebContent/WEB-INF"* befindet, mit der Einstellung in Quellcode 3.1 Zeile 2 definiert.

```
1 <welcome-file-list>
2   <welcome-file>View.jsp</welcome-file>
3 </welcome-file-list>
```

Quellcodeverzeichnis 3.1: Auswahl der zuerst initialisierten JSP

Von der *View* Seite aus kann man neue Rezepte anlegen (*Edit View*) oder Einträge aus der Übersicht dem Kochbuch hinzufügen, die *Detailansicht* eines Rezeptes aufrufen, von dem aus das Rezept ebenfalls dem Kochbuch hinzugefügt werden kann, oder zur *Kochbuchansicht* wechseln. Von dort aus kann das Kochbuch gelöscht, "gedruckt" oder zur *Übersichtsseite (View)* navigiert werden. Ruft man die *Edit View* auf wird ein neues Rezept in der Session erstellt und man kann entweder zur *Übersichtsseite (View)* zurückkehren oder seine Eingaben überprüfen. Von der *Confirm Ansicht* aus, kann man ein Bild zum Rezept hinzufügen und auf den Server laden oder zur *Equipment* oder *Ingredients* Ansicht wechseln. In diesen können die jeweiligen Optionen dem

Rezept hinzugefügt werden und abschließend zum Rezept (*Confirm View*) zurückkehren. Von hier aus kann das Rezept Hinzufügen abgeschlossen werden, indem das Rezept in der Datenbank gespeichert wird. Durch das Sichern (*Save*) wird das Rezept in der Datenbank gespeichert und man kehrt zur *Edit View* zurück, von der aus man (s.o.) neue Rezepte erstellen oder zur Übersichtsseite zurückkehren kann.

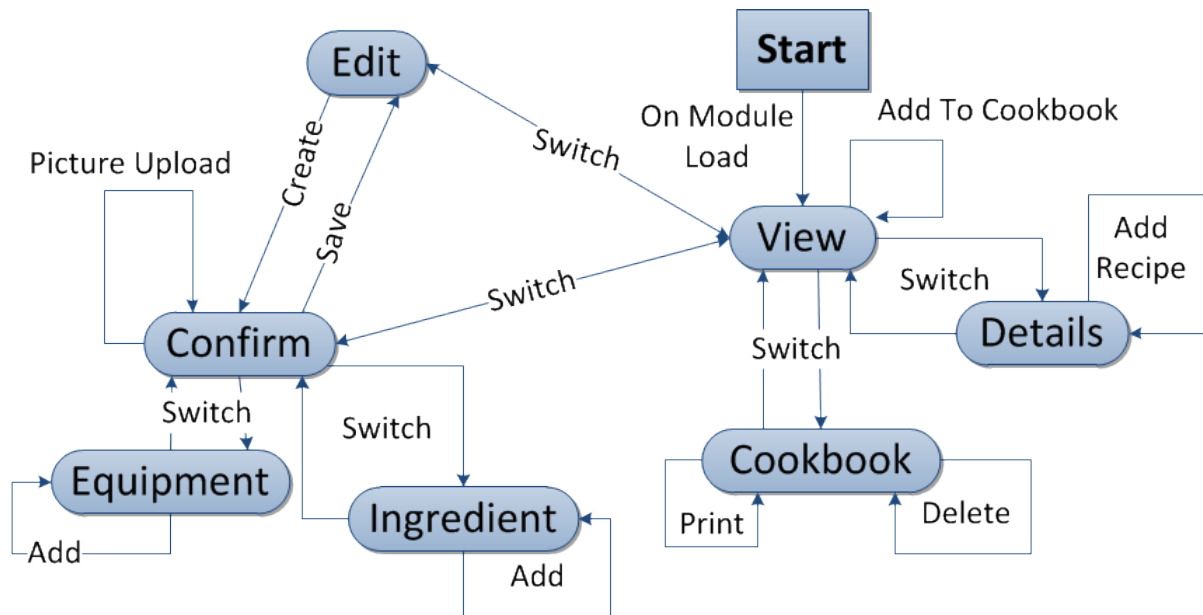


Abbildung 3.1: Programmfluss der Kochbuchanwendung

3.2 Die Edit.jsp Seite

Die Edit.jsp Seite (s. Quellcode 3.2) besteht aus einem typischen HTML-Gerüst. Teile dieses Gerüsts werden von den in JSP verfügbaren Direktiven ergänzt. Zu Beginn der Datei werden die Metainformationen zu den Java Bestandteilen aufgelistet. Ab Zeile 9 beginnt der body des HTML-Gerüsts. Durch das `jsp:useBean` in Zeile 10 wird die Klasse *Receipt* (eine *JBean*) unter dem Variablennamen *receipt* bekannt gemacht und falls unbekannt initialisiert. Mit `scope=„session“` wird der Gültigkeitsbereich der Bean auf die User-Session begrenzt. Defaultwert wäre *page*. Die `jsp:setProperty` Anweisung setzt alle Attribute der per `jsp:useBean` bekanntgemachten Bean, sofern sie ungleich null sind. Darauf folgt ein Formular von Zeile 12-22. Dieses überträgt seine Daten per *POST* und lässt die übertragenen Daten von einem Controller verwalten. Welche Datei beim Aufruf dieses Controllers ausgeführt wird, wird ebenfalls in der *web.xml*

Datei definiert. Das Formular besitzt mehrere Felder. Diese Felder im Textformat sind später über ihren in *name* definierten Index bekannt. Über das Feld des Types *submit* mit dem Index *enterReceipt* werden die Formulardaten mit der *POST*-Methode an den Controller gesendet. Die *Controller.java* Datei, die als Controller aufgerufen wird, prüft zunächst die Parameter des *POST*-Requests. Befindet sich unter diesen Parametern der Index *enterReceipt*, mit dem der Submit des Formulars gesendet wurde, wird die Destination für die Seite, die geladen wird, auf *Confirm.jsp* geändert. Später im Programmfluss des Controllers wird, falls die Destination *Confirm.jsp* ist, überprüft, ob es schon ein Attribut in der Session namens *receipt* gibt. Falls dies nicht der Fall ist, wird ein neues Attribut für das Rezeptobjekt erstellt und mit den Werten aus den Feldern (Titel des Rezepts, Name des Autors, Dauer, Bemerkung, etc.) initialisiert. Danach befindet sich in Zeile 23 ein Link auf die *View.jsp* Seite, auf der eine Übersicht der bekannten Rezepte angezeigt wird.

```
1 <%@ page language="java" contentType="text/html; charset=UTF-8"
2   pageEncoding="UTF-8"%>
3 <!DOCTYPE html PUBLIC "-//W3C//DTD HTML 4.01 Transitional//EN" "http://
4   www.w3.org/TR/html4/loose.dtd">
5 <html>
6 <head>
7 <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
8 <title>Rezepteingabe</title>
9 </head>
10 <body>
11   <jsp:useBean id="receipt" class="model.Receipt" scope="session" />
12   <jsp:setProperty property="*" name="receipt"/>
13   <form method="POST" action="Controller">
14     <table>
15       <tr><td>Rezeptname:</td><td><input type="text" name="title"></td></tr>
16       <tr><td>Author:</td><td><input type="text" name="author"></td></tr>
17       <tr><td>Beschreibung:</td><td><input type="text" name="description"
18         ></td></tr>
19       <tr><td>Dauer:</td><td><input type="text" name="duration"></td></tr>
20       <tr><td>Schwierigkeit:</td><td><input type="text" name="degree"></td></tr>
21       <tr><td>Bemerkung:</td><td><input type="text" name="note"></td></tr>
22     </table>
23     <input type="submit" name="enterReceipt" value="Weiter">
24   </form>
```

```

23 <a href="View.jsp">Zurueck zur Uebersicht</a>
24 </body>
25 </html>

```

Quellcodeverzeichnis 3.2: Die Edit.jsp der Kochbuchanwendung

3.3 Das verwenden eines Controllers

Zum verwenden eines Controllers muss dieser der Anwendung bekannt gemacht werden. Dies geschieht, in dem er in der *web.xml* im Verzeichnis *"/WebContent/WEB-INF"* als Servlet eingetragen wird. Ein Beispiel hierfür ist im Quellcode 3.3 zu sehen.

```

1 <servlet>
2   <servlet-name>ControllerServlet</servlet-name>
3   <servlet-class>control.Controller</servlet-class>
4 </servlet>
5 <servlet-mapping>
6   <servlet-name>ControllerServlet</servlet-name>
7   <url-pattern>/Controller/*</url-pattern>
8 </servlet-mapping>

```

Quellcodeverzeichnis 3.3: Beispielkonfiguration für einen Controller

Die Aufgabe des Controllers ist es den Seitenfluss der Anwendung zu steuern. Er ist Bestandteil des MVC Patterns und dient dazu die Auswertung der Benutzereingaben und die Logik von der Präsentationsschicht zu trennen. Es ist der Präsentationsschicht egal, was passiert, wenn ein Element geklickt wird. Sie kümmert sich lediglich um die Anzeige der Inhalte der Seite. Der Controller greift dann die ausgelösten Aktionen auf und kümmert sich um die mit den Aktionen verbundene Logik. Dies erhöht die Wartbarkeit von Anwendungen. Vereinfacht die Skalierbarkeit und vermeidet Fehler, da der Programmcode übersichtlicher zu lesen ist. In diesem Fall kümmert sich der Controller hauptsächlich um den Seitenfluss und das dazugehörige Sessionmanagement. Wird ein ausgefülltes Formular gesendet, kümmert sich der Controller um das aufbereiten der nächsten Seite und das verarbeiten der gesendeten Daten.

3.4 MVC-Pattern des Kochbuchs

Am einfachsten ist die Anwendung des MVC-Patterns des Kochbuchs zu erkennen, wenn man die Struktur des Projekts (Abb. 3.2) betrachtet. Die Struktur lässt durch die Benennung der Pakete direkt die der Model und des Controllers zugehörigen Dateien erkennen. Nimmt man noch die JSP Dateien im *WebContent* Verzeichnis als Präsentationsschicht hinzu hat man die strukturelle Aufteilung in das MVC-Pattern komplett. Der Controller ist, wie oben beschrieben, für den Seitenfluss und das Sessionmanagement zuständig. Das Model, welches die Daten beinhaltet, enthält in unserem Beispiel die *jBeans*, die in der Datenbank gespeichert werden. Sie enthalten die Daten, die für die Kochbuchanwendung nötig sind. Die Präsentationsschicht durch die JSP Dateien zeichnet sich dadurch aus, dass die JSP Dateien nur die für die Darstellung der Inhalte benötigte Funktionalität beinhalten. So besteht die Anwendung aus drei Hauptfunktionalitäten (Model, View und Controller), wie sie das MVC-Pattern vorschreibt. Weitere Elemente wie Validierung sind nicht genau definiert und in diesem Fall im *util*-Paket zusammengefasst.

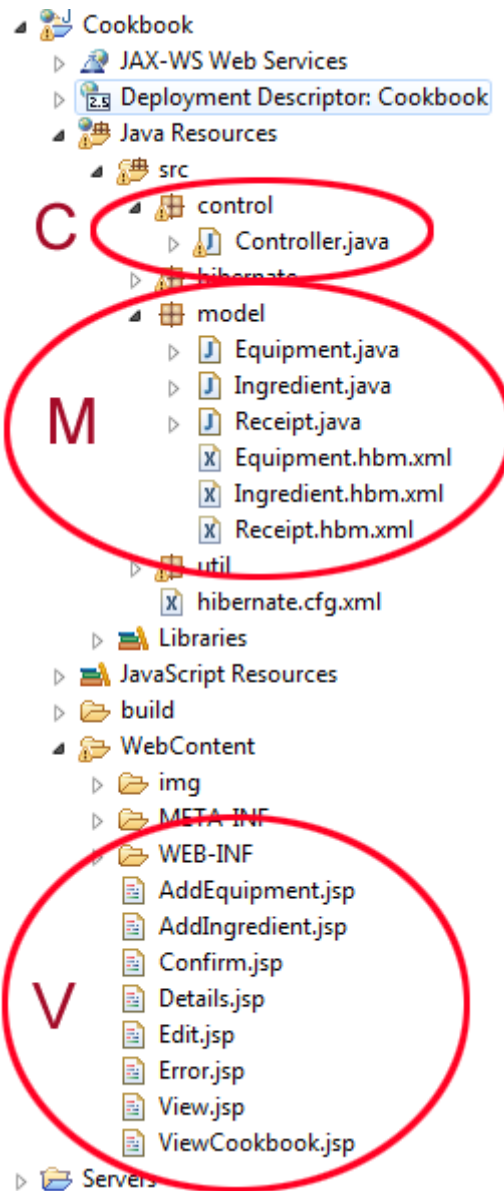


Abbildung 3.2: Strukturansicht des Kochbuch Projekts

4 Bilderupload

Das Hochladen von Bildern benötigt zunächst eine Eingabemöglichkeit für die Nutzer. Dies wird durch ein Formular realisiert und wird in Quellcode 4.1 dargestellt.

```
1 <form enctype="multipart/form-data" action="Controller" method=POST>
2   <table>
3     <tr>
4       <td><b>Datei auswaehlen:</b></td>
5       <td><input name="fileInput" type="file"></td>
6     </tr>
7     <tr>
8       <td colspan="2"><input type="submit" value="Datei uploaden">
9     </td>
10    </tr>
11  </table>
12 </form>
```

Quellcodeverzeichnis 4.1: Formular zum Hochladen von Bildern

Wichtig ist das *enctype* Attribut mit dem Wert *"mulitpart/form-data"* als Verschlüsselungsparameter der beim Verwenden des Eingabetyps *"file"* benötigt wird. Dadurch ist auch die Übertragungsmethode *POST* vorgeschrieben. Auch dieses Formular wird vom Controller verarbeitet. Darauf folgt ein gewöhnliches Formular Eingabefeld mit dem Typ *"file"* welches erlaubt direkt den Pfad zu einer Datei einzugeben oder durch klick auf einen "DurchsuchenButton ein Auswahlfenster des Betriebssystems aufruft. Durch klick auf das Eingabefeld des Typs *submit*" wird die Formaction ausgeführt. Im Controller selbst muss nun die Aktion auf die passende Verschlüsselungsmethode überprüft werden. Dies geschieht durch die Abfrage des in Quellcode 4.2 sichtbaren Codeausschnitts.

```
1  if ((contentType != null) && (contentType.indexOf("multipart/form-
2    data") >= 0))
3    {
4      ...
```

```
4 }
```

Quellcodeverzeichnis 4.2: Abfrage auf Verschlüsselungsmethode

Daraufhin muss der Datenstrom des Fileuploads eingelesen werden. Dies geschieht in unserem Fall über einen Zwischenpuffer. Der Code im Quellcode 4.3 beschreibt dies näher.

```
1   DataInputStream in = new DataInputStream(req.getInputStream());
2   int formDataLength = req.getContentLength();
3   byte dataBytes[] = new byte[formDataLength];
4   int byteRead = 0;
5   int totalBytesRead = 0;
6   while (totalBytesRead < formDataLength) {
7       byteRead = in.read(dataBytes, totalBytesRead, formDataLength);
8       totalBytesRead += byteRead;
9   }
```

Quellcodeverzeichnis 4.3: Zwischenspeichern des Datenstroms in einen Puffer

In Zeile 1 wird der Datenstrom initialisiert. In Zeile 2 die Länge des Datenstroms in *formDataLength* gespeichert. Zeile 3 initialisiert den Zwischenspeicher als Array des Types *byte* mit der Länge des Datenstroms. Zeile 4 und 5 initialisieren die Zähler für die gerade gelesenen Bytes und die insgesamt gelesenen Bytes. Die Schleife über Zeile 6-9 schaut, ob noch Daten aus dem Datenstrom fehlen. Solange das wahr ist wird aus dem Datenstrom gelesen und an *dataBytes[totalBytesRead]* und die folgenden Felder des Arrays geschrieben, bis maximal *formDataLength* Bytes gelesen wurden. Der Leseversuch kann aber auch schon früher abbrechen oder gar keine Bytes lesen. Daher muss das ganze iterativ in einer Schleife ausgeführt werden. Die Funktion *read* gibt die Anzahl gelesener Bytes zurück. Dieser Wert wird zu den insgesamt gelesenen Bytes addiert. So kann die Schleife ausgeführt werden, bis alle Bytes gelesen wurden (also bricht ab, sobald *totalBytesRead == formDataLength*).

Um die Datei speichern zu können muss ihr Dateiname aus den Metadaten extrahiert werden. Dies wird in Quellcode 4.4 beschrieben.

```
1   String file = new String(dataBytes, "ISO-8859-1");
2   int index = file.indexOf("filename=");
3   int indexFileNameStart = file.indexOf("\"", index) + 1;
4   int indexFileNameEnd = file.indexOf("\"", indexFileNameStart);
5   String saveFileName = file.substring(indexFileNameStart,
        indexFileNameEnd);
```

Quellcodeverzeichnis 4.4: Lesen des Dateinamen aus den Metadaten

Der Pfad inklusive Dateiname wird in Zeile 1 initialisiert. Dann sucht man nach dem Name *"filename="* des Name-Werte Paares für den Dateinamen. Der Dateiname startet nach dem Vorkommen des nächsten *"*-Zeichens und endet am darauffolgenden *"*-Zeichen. Aus den beiden Indizes lässt sich durch die *substring()*-Methode der String, der den Dateinamen enthält extrahieren.

Da die Einträge des Formulars über *boundarys* getrennt sind, muss zunächst der String dieser Begrenzung extrahiert werden. Dies wird in 4.5 beschrieben.

```
1         index = file.indexOf("boundary");
2         index = contentType.indexOf("=", index + 1);
3         String boundary = contentType.substring(index + 1, contentType.
           length());
```

Quellcodeverzeichnis 4.5: Extrahieren des boundary-Strings

Man sucht im Datenstrom nach dem Vorkommen von *"boundary"*. Der auf dem nächsten *"* folgende String bis zum Ende des Datentyps ist der Begrenzungsstring. Man muss außerdem den Start des Index auf den Anfang der Binärdaten setzen. Dies wird in Quellcode 4.6 beschrieben.

```
1         int indexStart = file.indexOf("Content-Type", indexFileNameEnd);
2         indexStart = file.indexOf("image", indexStart + 1);
3         indexStart = file.indexOf("\n", indexStart + 1);
4         indexStart = file.indexOf("\n", indexStart + 1);
5         indexStart++;
```

Quellcodeverzeichnis 4.6: Navigiere zum Start der Binärdaten

Daraufhin müssen die über *boundary* getrennten Eingabeelemente separiert werden. Eine Umsetzung dessen ist in Quellcode ?? zu sehen.

```
1         int boundaryLocation = file.indexOf(boundary, indexStart);
2         int endPos = file.substring(0, boundaryLocation).length() - 4;
3         filePath = "img/" + saveFileName;
4         saveFileName = "D:/Users/Watnuss/Programming/cookbook/kochbuch/
           WebContent/img/" + saveFileName;
```

Quellcodeverzeichnis 4.7: Trennung der Eingabeelemente nach dem Begrenzungsstring

Es wird der Index des Begrenzungsstring ermittelt und die Länge der zu lesenden Binärdatei. Darauf folgend wird aus dem Dateipfad zum Speicherplatz und dem Dateinamen der Pfad der Datei auf dem Server gebildet.

Als letzter Schritt wird die Zwischengespeicherte Datei an ihren Bestimmungsort geschrieben. Dies ist in Quellcode 4.8 zu beobachten.

```
1      FileOutputStream fileOut = new FileOutputStream(saveFileName);  
2      fileOut.write(dataBytes, indexStart, (endPos - indexStart));  
3      fileOut.flush();  
4      fileOut.close();
```

Quellcodeverzeichnis 4.8: Speichern der Datei auf dem Server

Es wird ein Datenstrom zum Bestimmungsort der Datei geöffnet und die Daten aus dem Puffer vom Start der Binärdatei bis zu ihrem Ende geschrieben. Danach wird der Datenstrom geschlossen und die Datei ist auf dem Server verfügbar.

5 Appendix

Abbildungsverzeichnis

3.1	Programmfluss der Kochbuchanwendung	4
3.2	Strukturansicht des Kochbuch Projekts	8

Quellcodeverzeichnis

3.1	Auswahl der zuerst initialisierten JSP	3
3.2	Die Edit.jsp der Kochbuchanwendung	5
3.3	Beispielkonfiguration für einen Controller	6
4.1	Formular zum Hochladen von Bildern	9
4.2	Abfrage auf Verschlüsselungsmethode	9
4.3	Zwischenspeichern des Datenstroms in einen Puffer	10
4.4	Lesen des Dateinamen aus den Metadaten	10
4.5	Extrahieren des boundary-Strings	11
4.6	Navigiere zum Start der Binärdaten	11
4.7	Trennung der Eingabeelemente nach dem Begrenzungsstring	11
4.8	Speichern der Datei auf dem Server	12

