Entwicklung eines Online Kochbuchs mithilfe von Java Server Pages und Hibernate

Webengineering II

Dokumentation für Webengineering im Studiengang Angewandte Informatik

an der Dualen Hochschule Baden-Württemberg Stuttgart

von

Christian Burkard

Juni 2012

Matrikelnummer, Kurs 4206853, TITAIA2010
Ausbildungsfirma Hewlett-Packard GmbH, Böblingen
Betreuerin Prof. Dr. Barbara Dörsam

Inhaltsverzeichnis

1 Aufgabenstellung

Ziel der Vorlesung Webengineering II war es den Studenten die serverseitige Programmierung anhand von Java Server Pages (JSP) beizubringen. Es kommt eine Drei-Schichten-Architektur (MVC) zum Einsatz. Die Anbindung an die MySQL-Datenbank erfolgt über Hibernate. Dieses Dokument dient der Dokumentation der wichtigsten Module des Kochbuchs.

2 Datenbankschnittstelle: Hibernate

Dieses Kapitel behandelt die Datenbankanbindung der Kochbuchanwendung.

2.1 Object-Relational Mapping (ORM)

Hibernate bietet die Möglichkeit relationale Datenbankstrukturen auf Java Objekte zu mappen. Betrachtet man das ER-Model der Datenbank (s. Abb. 2.1) und das UML Diagramm der Modellschicht (s. Abb. 2.2), fällt einem sofort die strukturelle Ähnlichkeit auf. Hibernate bietet die Möglichkeit mithilfe eines Mappings durch XML Files die Modellschicht direkt auf das relationale Schema der Datenbank abzubilden.

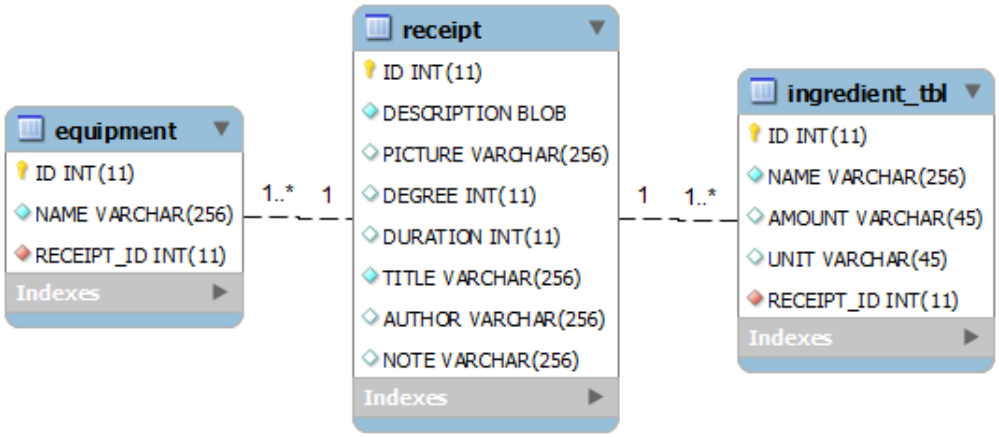

Abbildung 2.1: ER-Model der Kochbuch Datenbank

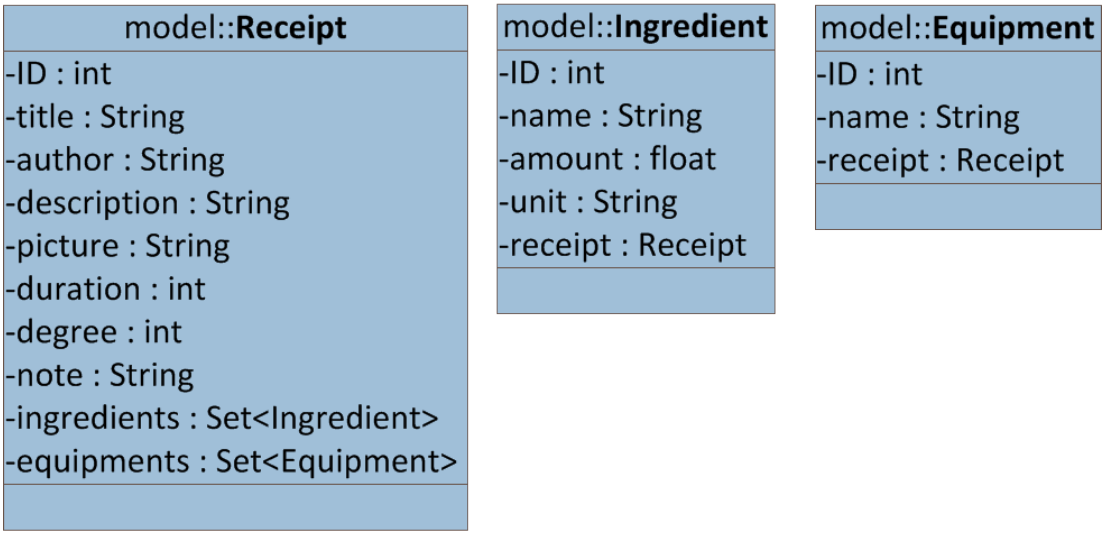

Abbildung 2.2: UML der Klassen der Modellschicht

2.2 Klassen - Tabellen Mapping

Wie in Abbildung 2.1 und 2.2 zu sehen ist werden drei Klassen auf drei Tabellen abgebildet. Jeweils eine Klasse wird auf eine Tabelle abgebildet. ($Namentlich: Receipt \rightarrow receipt, Ingredient \rightarrow ingredient\_tbl\ und\ Equipment \rightarrow equipment$) Das Mapping findet statt, indem für jede Klasse, die auf eine Tabelle abgebildet werden soll, eine XML Konfigurationsdatei im selben package wie die Klasse angelegt wird, die dieses Mapping definiert. Heißt die Klasse *"Foo.java"* ist der Name der Konfigurationsdatei *"Foo.hbm.xml"*. Sie sagt der Klasse auf welche Tabelle sie abgebildet wird. In diesem Mapping können dann Attribute und Relationen für diese Tabelle/Klasse definiert werden.

2.3 Mapping einfacher Attribute

Ein einfaches Attribut wird über das proberty Tag auf eine Tabellenspalte gemappt. Ein solches Beispielmapping eines Attributs sieht man in Quellcode 2.1. Das Attribut *"name"* benennt das Attribut der Klasse. *"column"* nennt die Spalte der Tabelle, auf die das Attribut, das durch *"name"* definiert wird, gemappt wird. *"type"* ist der Datentyp in dem das Attribut aus der Klasse gelesen und in der Datenbank gespeichert wird. Der Datentyp ist weder der der Javaklasse noch der Datentyp der Datenbanktabellenspalte. Es ist ein Hibernate mapping Typ. Im Beispiel Quellcode 2.1

ist außerdem, ein optionales Attribut *"not-null"* zu sehen. Die vorherigen Attribute waren alle Pflichteingaben, auf die weitere optionale Attribute folgen können. *not-null="true"* erlaubt es nur Werte ungleich null zu der Spalte hinzuzufügen.

```
1     <property name="title" column="TITLE" type="string" not-null="true"
          />
```

Quellcodeverzeichnis 2.1: Mapping eines Attributes

2.4 Mapping von Relationen

In Hibernate gibt es zwei Möglichkeiten des Mappings für 1:n Beziehungen. Man kann das Mapping aus zwei Perspektiven sehen: one-to-many oder eine many-to-one Relation. In unserem Kochbuch sind die Rezepte 1:n mit den Zutaten verbunden. Es ist eine one-to-many Beziehung. Genauso könnte man sagen sind die Zutaten many-to-one mit den Rezepten verbunden. So kann man die Beziehung wie in Beispiel Quellcode 2.2 als one-to-many Beziehung von Rezepten zu Zutaten beschreiben. Die one-to-many Beziehung wird in der Klasse mithilfe eines HashSets dargestellt. In Hibernate werden Sets mit dem *set* Tag dargestellt. Dieses HashSet wird hier mit dem *"name"* Attribut angesprochen. *"cascade"* und *"lazy"* sind optionale Attribute und beschreiben das Verhalten bei Updates/Löschen (hier: Zutaten werden auch aktualisiert, wenn Rezepte aktualisiert wurden) oder versetzen Transaktionen mit einer Priorität (hier: nur Indizes des Sets werden geladen. Das Objekt selber wird erst bei Bedarf aus der Datenbank geladen. Für große Anwendungen bietet diese Option Performance-Vorteile). Der Tag *key* mit seinem Attribut *"column"* beschreibt die Spalte in der Tabelle, in der die Relation als Fremdschlüssel gespeichert wird. Das *one-to-many* Tag gibt mit seinem *"class"* Attribut die Klasse in der sie referenziert wird an. Dadurch ist auch die Hibernate Konfigurationsdatei und darin die Tabelle in der Datenbank zu dieser Klasse bekannt.

```
1     <set name="ingredients" cascade="all" lazy="true">
2     <key column="RECEIPT_ID" />
3     <one-to-many class="model.Ingredient" />
4   </set>
```

Quellcodeverzeichnis 2.2: One-To-Many Beziehung von Rezepten zu Zutaten

Die andere Richtung beschreibt das Beispiel Quellcode 2.3. Das *"name"* Attribut des *many-to-one* Tags gibt den Namen des Attributs in der Klasse an, die den Fremd-

schlüssel als Objektreferenz speichert. Das *"class"* Attribut gibt die Klasse auf die die Referenz zeigt an und entspricht dem Datentyp des Java Attributs. *"column"* gibt die Spalte in der Datenbanktabelle an, in der die Referenz über einen Fremdschlüssel abgebildet wird. Nun kommt abschließend noch ein optionales Attribut *"cascade"* und verhält sich wie in Beispiel Quellcode 2.2 beschrieben: Bei Updates oder Löschen des referenzierten Tabelle (Rezept) kaskadiert die Aktion. Die Zutaten werden ebenfalls aktualisiert/gelöscht.

```
1     <many-to-one name="receipt" class="model.Receipt" column="RECEIPT_ID"
          cascade="all" />
```

Quellcodeverzeichnis 2.3: Many-To-One Beziehung von Zutaten zu Rezepten

Die Unterschiede der Relationen

1:1 Relationen verbinden eine Zeile einer Tabelle mit einer Zeile einer anderen Tabelle. Sobald eine Zeile einer Tabelle mit mehreren Zeilen einer anderen Tabelle verbindet ist es eine 1:n Relation. Bei n:m Relationen werden mehrere Zeilen der einen Tabelle mit mehreren Zeilen einer anderen Tabelle verbunden. Da diese Information jedoch nicht mehr in einer Spalte sinnvoll gespeichert werden kann, benötigt man eine extra Tabelle um diese Verbindungsinformationen zu speichern.

In Hibernate ist dies ebenfalls so. Durch die one-to-one, one-to-many, many-to-one und many-to-many Tags können diese Relationen auch über Hibernate abgebildet werden.

2.5 Bidirektionales Mapping

Indem eine Relation in beiden Konfigurationsdateien definiert wird, macht man die beiden Tabellen bidirektional bekannt. Das bedeutet, dass ein Update eines Objektes, dass auf eine dieser Tabellen abgebildet ist, auch die anderen Objekte, die zu dieser Relation gehören, ebenfalls aktualisiert. So genügt es auch beim Einfügen in die Tabelle eines der Objekte der Relation hinzuzufügen. Würde man die Tabellen nicht bidirektional bekannt machen, wäre dies eine mögliche Fehlerquelle im späteren Programmieren der Anwendung. Durch bidirektionales Verknüpfen wird diese Fehlerquelle konzeptionell vermieden.

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.

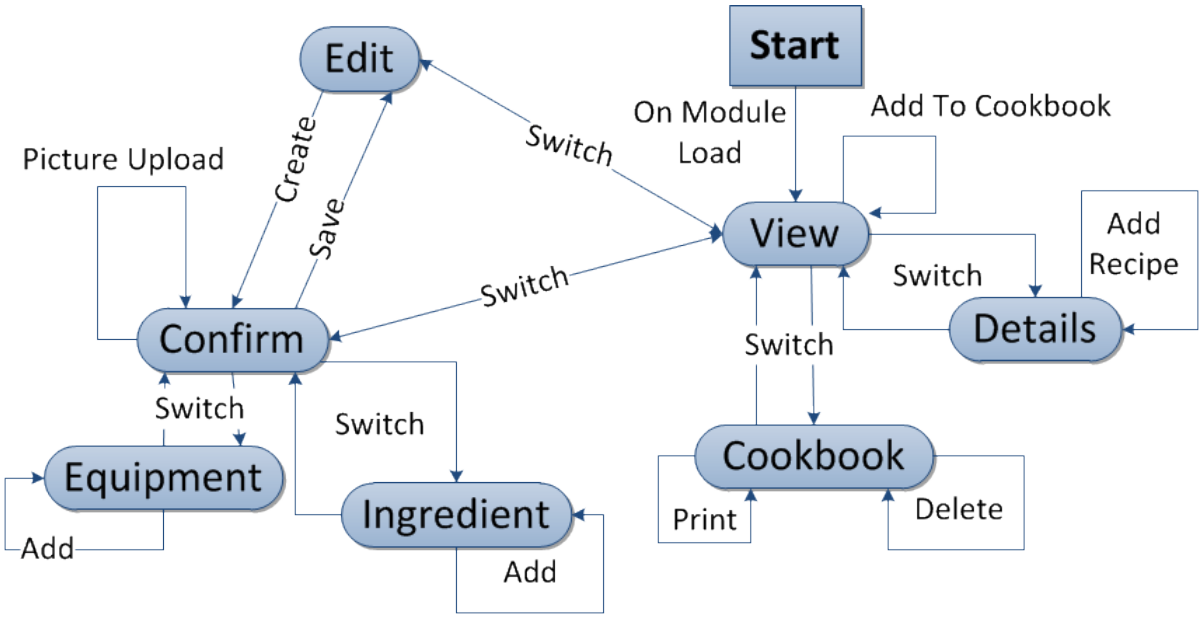

Abbildung 3.1: Programmfluss der Kochbuchanwendung

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 dieses 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://
       www.w3.org/TR/html4/loose.dtd">
4  <html>
5  <head>
6  <meta http-equiv="Content-Type" content="text/html; charset=UTF-8">
7  <title>Rezepteingabe</title>
8  </head>
9  <body>
10   <jsp:useBean id="receipt" class="model.Receipt" scope="session" />
11   <jsp:setProperty property="*" name="receipt"/>
12   <form method="POST" action="Controller">
13     <table>
14       <tr><td>Rezeptname:</td><td><input type="text" name="title"></td></
             tr>
15       <tr><td>Author:</td><td><input type="text" name="author"></td></tr>
16       <tr><td>Beschreibung:</td><td><input type="text" name="description"
             ></td></tr>
17       <tr><td>Dauer:</td><td><input type="text" name="duration"></td></tr
             >
18       <tr><td>Schwierigkeit:</td><td><input type="text" name="degree"></
             td></tr>
19       <tr><td>Bemerkung:</td><td><input type="text" name="note"></td></tr
             >
20     </table>
21     <input type="submit" name="enterReceipt" value="Weiter">
22   </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.

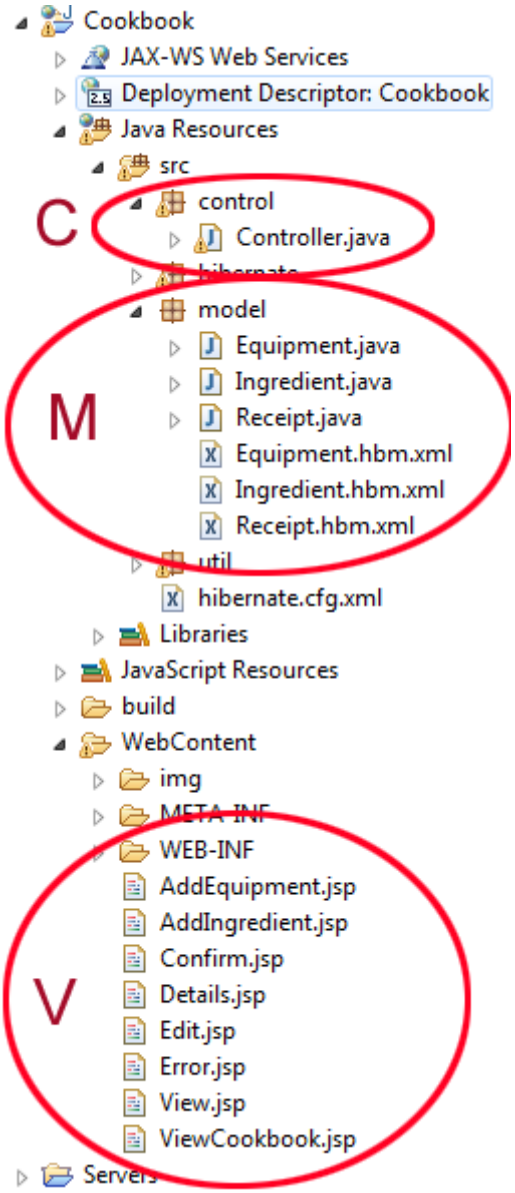

Abbildung 3.2: Strukturansicht des Kochbuch Projekts

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.

```
<form enctype="multipart/form-data" action="Controller" method=POST>
  <table>
    <tr>
      <td><b>Datei auswaehlen:</b></td>
      <td><input name="fileInput" type="file"></td>
    </tr>
    <tr>
      <td colspan="2"><input type="submit" value="Datei uploaden">
      </td>
    </tr>
  </table>
</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 Betriebsystems aufruft. Durch klick auf das Eingabefeld des Typs *ßubmit"* 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.

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

```
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 Funkion *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 au das darauffolgende "-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. Ein 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

Quellcodeverzeichnis

Akronyme

HTML Hypertext Markup Language.

JSP Java Server Pages.

MVC Model-View-Controller.

XML Extensible Markup Language.

Glossar

Hibernate

Ein Open-Source Framework, mit Object-Relational-Mapping für Java Anwendungen.

MySQL

Eine sehr weit verbreitete, relationale Open-Source-Datenbank.