



**Hochschule für Technik
und Wirtschaft Berlin**

University of Applied Sciences

**Angewandte Informatik
Systementwicklung und Frameworks**

Konzeption und Entwicklung einer CD-Tauschbörse mit Ruby on Rails

Christian Bunk
Alexander Miller
Christian Sandvoß
Antonia Ziegler

Abgabedatum: 08.01.2012

Prof. Dr. Albrecht Fortenbacher

Inhaltsverzeichnis

1. Einleitung	1
2. Erfahrungen vor dem Projekt	2
2.1. Antonia Ziegler	2
2.2. Christian Sandvoss	2
2.3. Alexander Miller	2
2.4. Christian Bunk	3
3. Ruby-on-Rails Framework	4
3.1. Rails-Konventionen	4
3.2. Model-View-Controller (MVC)	4
3.3. Don't Repeat Yourself (DRY)	5
3.4. Object-Relation-Mapper (ORM)	5
4. Anwendung	7
4.1. Projektmanagement	7
4.1.1. Meilensteinplanung	7
4.1.2. Versionsverwaltung von Quellcode	8
4.1.3. Deployment	9
4.2. Konzeption der Datenbasis	11
4.3. Realisierung der Funktionalität	12
4.3.1. Benutzerverwaltung	12
4.3.2. Compact Discs	14
4.3.3. Tausch-Transaktionen	14
4.3.4. Sonstige Anforderungen	16
4.4. Graphical User Interface	17
4.5. Internationalisierung	18
4.6. File Upload (Paperclip)	19
4.7. MusicBrainz	20
4.8. Tests	21
5. Zusammenfassung	26
5.1. Antonia Ziegler	26
5.2. Christian Sandvoss	26
5.3. Alexander Miller	26

5.4. Christian Bunk	27
A. Anhang	29
A.1. Aufgabenstellung	29

1. Einleitung

In der Softwaretechnik wird unter dem Begriff Framework ein Ordnungsrahmen verstanden, innerhalb dessen die Entwickler eine Anwendung erstellen. Das Ziel ist die Definition einer einheitlichen Struktur mittels komponentenbasierten Entwicklungsansätzen sowie die Vereinfachung der Entwicklung durch die Verwendung von zur Verfügung stehenden Frameworks-Bestandteilen. Im Laufe der Lehrveranstaltung "Systementwicklung und Frameworks" wurden die Prinzipien der unterschiedlichen Frameworks wie zum Beispiel EJB und .NET vorgestellt und diskutiert. Als Prüfungsleistung muss ein Projekt erfolgreich realisiert werden, wobei jede Gruppe von vier bis fünf Personen die Anwendung mit einem ausgewählten Framework entwickeln müssen.

In der vorliegenden Ausarbeitung handelt es sich um eine Dokumentation zum Projekt, wobei eine CD-Tauschbörse konzipiert und implementiert wurde. Wie in der Aufgabenstellung bzw. im Pflichtenheft definiert (Anhang A.1) wurde eine Rich-Applikation (RIA) modelliert und entwickelt, wobei die meisten Funktionalitäten nicht von Hand, sondern durch die Verwendung von Framework-Komponenten realisiert wurden. Die Gruppe besteht aus vier Personen (Christian Bunk, Alexander Miller, Christian Sandvoß, Antonia Ziegler) und entwickelt das Projekt mit dem Framework "Ruby on Rails".

Ruby on Rails ist ein Framework für Webapplikationen, welches in der Programmiersprache Ruby entwickelt wurde. Model-View-Controller (MVC) Architektur ermöglicht eine Isolation der Businesslogik von der graphischen Benutzeroberfläche und gewährleistet das "don't repeat yourself" (DRY) Prinzip.

Der SourceCode wird mit Git (<https://github.com/christianb/SE-FW>) verwaltet. Hierzu wird die online Plattform GitHub als zentrales Repository verwendet, die sowohl die aktuellen Aufgaben (Issues) als auch die Meilensteine verwaltet. Um eine Arbeitsgrundlage zu erschaffen, haben alle Gruppenmitglieder sich das grundlegende Wissen über das ausgewählte Framework angeeignet. Wie die geforderten Funktionalitäten im Einzelnen implementiert sind, kann den folgenden Kapiteln entnommen werden. An dieser Stelle muss noch ergänzt werden, dass alle gestellten Anforderungen erfüllt wurden.

2. Erfahrungen vor dem Projekt

In diesem Kapitel beschreiben die Projektteilnehmer ihre Erfahrungen mit Webframeworks vor dem Projekt.

2.1. Antonia Ziegler

Mit dem Ruby on Rails habe ich noch keine grossen Erfahrungen, die Webentwicklung ist zwar für mich ein sehr interessantes Gebiet. Bisweilen habe ich nur mit HTML4 und CSS gearbeitet und erste Erfahrungen mit PHP gesammelt. Ruby on Rails wird als ein sehr modernes und zukunftsorientiertes Framework dargestellt. Ich finde es deswegen interessant und möchte herausfinden, ob es diesen Versprechungen gerecht wird.

2.2. Christian Sandvoss

Ich finde, das man Rails sehr gut verwenden kann um auch umfangreiche Webanwendung zu realisieren. Rails stellt ein Grundsystem bereit, das durch viele Pugins erweiterbar ist. Allerdings muss man dabei darauf achten ob das Plugin zur genutzten Rails Version kompatibel ist oder überhaupt noch weiter entwickelt wird. Des weiteren ist bei einigen Plugins die Dokumentation sehr kurz gehalten. Die Dokumentation von Rails ist dagegen, wie ich finde, sehr gut. Es gibt Tutorials zum Einstieg sowie eine aktive Community. Neu für mich war die Arbeit mit JavaScript bzw. JQuery. Es war für mich sehr interessant das Zusammenspiel von verschiedenen Sprachen (Ruby, HTML/CSS, JavaScript) kennenzulernen.

2.3. Alexander Miller

Vor der Lehrveranstaltung SE-FW hatte ich keine Erfahrungen mit Ruby on Rails gehabt. Es war mir klar auf welchen Ansätzen dieses Framework basiert. Um mehr darüber zu erfahren und die einzelnen Vorteile durch ein Projekt zu erkennen, habe ich in meiner Gruppe ebenfalls für Ruby on Rails abgestimmt. Meine Erwartungen an dieses Framework sind: Flexibilität, Einfachheit, Erweiterbarkeit und schnelle Implementierung.

2.4. Christian Bunk

Ich habe bisher nur sehr wenig mit web frameworks gearbeitet. Im fünften Semester Bachelor haben wir an der Plattform NetBase Education von Prof. Langebein gearbeitet. Dort wurde Struts verwendet. Aufgrund der Größe des Projekts war die Arbeit mit Struts sehr umständlich. Viele Arbeiten mussten von Hand erledigt werden. Die Arbeit mit Web Frameworks war für mich eine eher lästige Angelegenheit. In meinem Praktikum bei ART+COM habe ich dann einiges über Ruby on Rails gehört. Dort wurde mir dann klar, dass ich mich unbedingt dieses Framework in mein Portfolio aufnehmen muss. Leider hatte ich seitdem noch keine Gelegenheit etwas praktisches damit umzusetzen. Aus diesem Grund wollte ich in dem Fach Systementwicklung und Frameworks etwas mit diesem Framework machen.

3. Ruby-on-Rails Framework

Ruby On Rails (Rails) ist ein Web-Framework welches auf der Programmiersprache Ruby basiert. Ruby ist eine Objektorientierte Programmiersprache. Anweisungen werden nicht durch ein Semikolon abgeschlossen sondern durch einen Zeilenumbruch. Jede Rails Anwendung hat eine feste Ordnerstruktur und verfolgt das Prinzip "Convention Over Configuration". Was bedeutet das der Konfigurationsaufwand durch einhalten von Konventionen so gering wie möglich gehalten werden soll. Zusätzlich zu allen erstellten Controllern wird auch ein so genannter Application-Controller erstellt. Alle weiteren Controller sind Unterklassen von diesem und erben daher auch alle Methoden die darin deklariert wurden. Im nächsten Abschnitt sind die wichtigsten Konventionen näher erläutert.

3.1. Rails-Konventionen

Wie oben erwähnt verfolgt Rails das Konzept "Convention Over Configuration". Darunter fallen z.B. die Namenskonventionen welche besagen, dass die Namen der Controller möglichst im Plural benannt werden und Models im Singular. Des Weiteren wird die Schreibweise von Variablen- und Klassennamen geregelt. Durch diese Konventionen weiss Rails welcher Controller zu welchen Model gehört, sowie welche Views zu einem Controller. Dadurch entfällt die explizite Zuweisung zwischen den einzelnen Komponenten. Die Methoden im Controller werden Actions genannt. Zu jeder Action gibt es meist eine View welches Äquivalent zur Bezeichnung der Action benannt ist. Sobald einen View und eine Action den gleichen Namen haben, wird beim Aufruf dieser automatisch die entsprechende View geladen.

3.2. Model-View-Controller (MVC)

Das Rails-Framework ist nach dem Model-View-Controller Konzept aufgebaut. Dabei dient das Model zur Kommunikation mit der Datenbank. Außerdem sorgt es durch Validierung dafür, dass Daten im korrekten Format in die Datenbank geschrieben werden. Die Views sind die HTML-Seiten, welche dem Nutzer angezeigt werden. Sie dienen zur Anzeige der Daten aus der Datenbank oder dem Erfassen von Nutzereingaben mit Hilfe von Formularen bzw.

Eingabefeldern. Der Controller ist der Vermittler zwischen der View und dem Model. Er empfängt Daten von der View und sendet sie an das Model weiter. Umgekehrt werden Daten vom Model über den Controller der View zur Verfügung gestellt.

3.3. Don't Repeat Yourself (DRY)

DRY beschreibt das "Don't Repeat Yourself" Konzept. Dies bedeutet das so wenig wie möglich Code dupliziert wird. Zur Einhaltung dessen gibt es in Rails einigen Vorkehrungen. Als erstes wären da die Helper-Methoden zu nennen. Dies sind Methoden welche oft dazu genutzt werden, häufig genutzten Code durch Schlüsselwörter zu ersetzen. Die Helper-Methoden werden meist in den Views verwendet um dort HTML-Code zu generieren. Ein oft verwendeter Helper ist beispielsweise "link-to", welcher den aus HTML bekannten Link-Tag generiert.

Ein weiterer Punkt ist die Definition des Seitenlayouts. Das Layout legt die Darstellung bzw. das Aussehen der HTML-Seiten fest. Dabei kann ein globales, für alle Seiten gültiges Layout definiert werden oder auch Layout-Vorlagen für spezielle Seiten definiert werden. Das Layout welches mit "application.html.erb" benannt wird, ist für alle Seiten gültig. Falls ein anderes Layout verwendet werden soll, muss diese explizit eingebunden werden. Ansonsten wird immer das in der Datei "application.html.erb" definierte genutzt. Das gleiche gilt auch für den Application-Controller. Hier definierte Methoden sind in allen anderen Controllern verfügbar und müssen dadurch nicht in jedem Controller separat implementiert werden.

Seit der Version 3.1 von Rails, wird standardmäßig jQuery als Javascript Framework verwendet. Auch hier gibt es eine globale Datei (application.js) auf die aus allen Views zugegriffen werden kann. Javascript Code wird in der Regel nicht in den Views eingefügt, sondern es wird auf die zu verändernden HTML Elemente durch dessen IDs oder Klassen zugegriffen. Dies dient den Prinzip des Unobtrusive (unaufdringlich) Javascript und erhöht die Lesbarkeit des Codes der Views. Eine weitere Neuerung der Version 3.1, ist die Asset-Pipeline. Darin werden die CSS und JavaScript Dateien gespeichert. Diese werden dann, vor Auslieferung an den Browser komprimiert, um die zu übertragene Datenmenge zu verringern und dadurch die Zugriffszeit zu erhöhen.

3.4. Object-Relation-Mapper (ORM)

Um bei Rails Objekte in einer Datenbank zu speichern, kommt der Object-Relation-Mapper ActiveRecord zum Einsatz. Dabei werden die Tabellen als Klasse gesehen, die Zeilen als Objekt und die Spalten sind die Objekt-Attribute.

Durch den ORM müssen keine SQL-Statements mehr geschrieben werden, sondern für den Zugriff auf die Datenbank werden Methoden genutzt. Beim erstellen einer Tabelle werden von Rails automatisch Methoden generiert. Eine Klasse hat unter andren die Methoden "new" und "find". Diese dienen zum erstellen oder finden eines Objekts. Das Objekt besitzt die Methoden "save", "update" und "delete". Diese werden zum Speichern, Aktualisieren und Löschen genutzt.

4. Anwendung

4.1. Projektmanagement

Eine der wichtigsten Aufgaben bei der Realisierung eines informationstechnischen Projektes ist das Projektmanagement. Es ist äußerst wichtig eine genaue Planung sowie die Konzeption durchzuführen, damit alle Anforderungen, Termine und der Kostenrahmen eingehalten werden. In den nächsten Kapiteln wird es beschrieben wie das Projektmanagement erfolgte, wobei intensiv auf die Meilensteinplanung sowie die Versionsverwaltung und Deployment eingegangen wird.

4.1.1. Meilensteinplanung

Um das Zeitmanagement zu organisieren, wurde die Meilensteinplanung durchgeführt. Es wurde vereinbart, dass bei der Entwicklung der meisten Funktionalitäten immer zwei Gruppenmitglieder beteiligt sein müssen, um bei krankheitsbedingten Ausfällen eines Gruppenmitglieds die Entwicklung fortsetzen zu können.

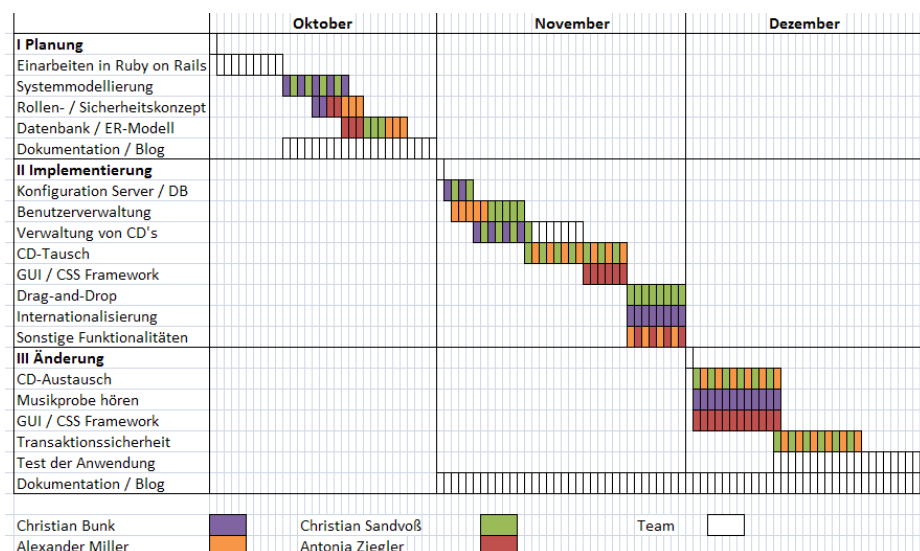


Abbildung 4.1.: Meilensteinplanung

Darüber hinaus wurde es bestrebt, die Konzeption sowie die Systemmodellierung als Gruppenarbeit durchzuführen. Für diese Zwecke wurden wöchentliche Termine (dienstags 12:00-14:00 und donnerstags 11:15 - 12:30) vereinbart. An diesen Terminen wurden in Rahmen einer Gruppenarbeit sowohl die aufgetretenen Schwierigkeiten gelöst, die wöchentliche Präsentation vorbereitet als auch der weitere Projektverlauf diskutiert.

4.1.2. Versionsverwaltung von Quellcode

Für die Quellcode-Verwaltung wurde ein GitHub-Repository verwendet. Dabei wurde an das Branching-Modell orientiert, die in (<http://nvie.com/posts/a-successful-git-branching-model/>) vorgestellt wird.

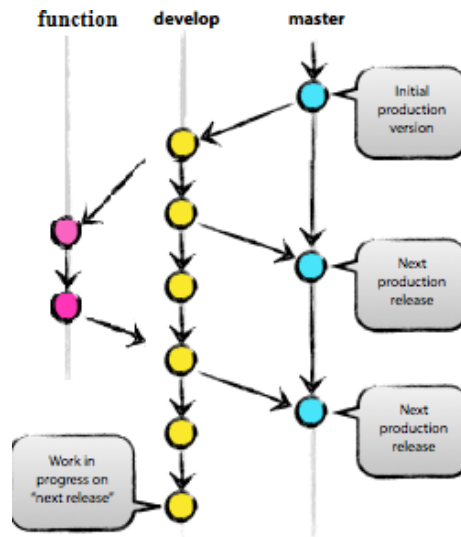


Abbildung 4.2.: Git - Function, Develop und Master Branches

Dabei wird folgende Strategie verfolgt: Als erstes werden zwei Branches angelegt *develop* und *master*. Die eigentliche Entwicklung geschieht auf dem *develop* Branch und lediglich die Zwischenversionen, die eine neue und vor allem fehlerfreie Implementierung eine zusätzlichen Funktionalität beinhalten, werden auf den *master* Branch gelegt. Durch das Erstellen von weiteren Branches für die jeweilige Funktionalität kann weiterer Komfort bei der Entwicklung erreicht werden. Jede zusätzliche Funktionalität wird dabei separat entwickelt, so dass eine Aufteilung der Aufgaben sowie die anschließende Zusammenführung der Quelltexte ohne Einschränkungen und Probleme erfolgen können. Wenn eine Funktion fertig gestellt worden ist, wird diese ins *develop* Branch kopiert. Falls mehrere Funktionalitäten im *develop* Branch getestet und die identifizierten Fehler behoben wurden, wird die Anwendung ins *master* Branch gemergt.

Bei der Einhaltung des vorgestellten Modells sieht die Quellcode-Verwaltung wie folgt aus:

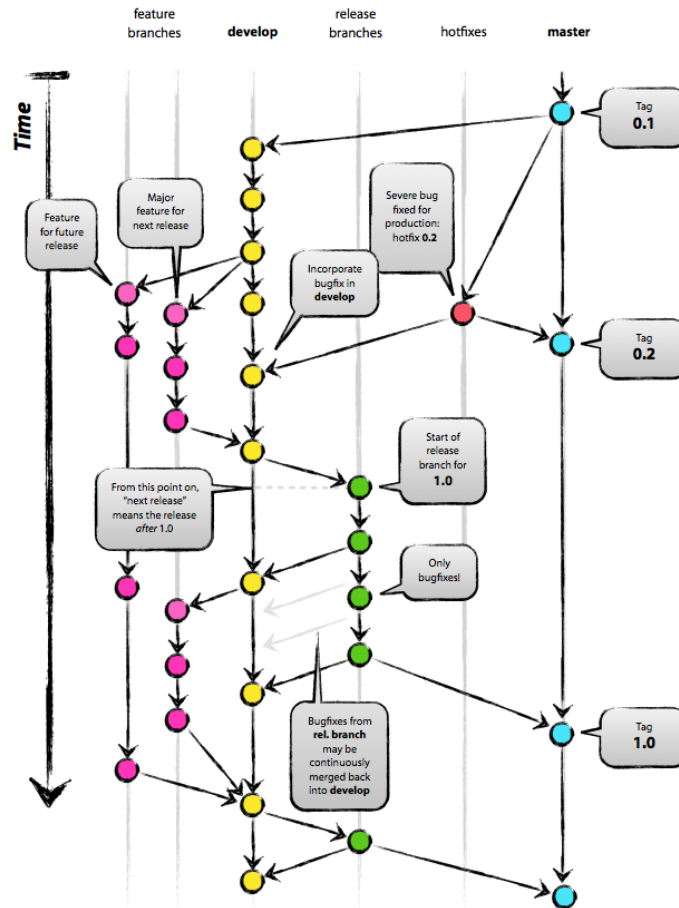


Abbildung 4.3.: Git - Feature und Develop Branches

4.1.3. Deployment

Da die Anwendung auf einem Webserver zur Verfügung gestellt werden soll, wurde dieser eingerichtet¹. Dafür wurde auf den uns zur Verfügung stehenden Server, als erstes die benötigte Software installiert. Um überhaupt Rails Anwendungen ausführen zu können, wurde Ruby in der Version 1.9.2 und Rails 3.1.1 installiert. Anschließend wurde der Apache Webserver installiert. Dieser wurde entsprechend konfiguriert sowie eine PostgreSQL Datenbank installiert. Desweiteren wurde, Capistrano ein OpenSource Deployment-Tool für Rails-Anwendungen, installiert. Dabei handelt es sich um eine Software für das automatisierte Ausführen von Aufgaben auf einem oder mehreren entfernten

¹<http://kallisto.f4.htw-berlin.de>

Servern. Das zentrale Prinzip des Tools ist die vollständige Automatisierung des Verteilungsprozesses. Somit sind die einzelnen Schritte in einem zusammengefasst, wie: Auschecken der Software aus der Versionskontrolle, Ausführen der Unit-Tests, Übertragung der Software auf die Ziel-Server, Aktualisierung der Datenbanken und Neustart des Webservers. Diese Schritte wären einzeln ausgeführt sehr zeitintensiv, fehleranfällig und zu dem auf Dauer langweilig, vor allem da in einem agilen Entwicklungsprozesses dies relativ häufig ausgeführt werden muss.

4.2. Konzeption der Datenbasis

Für die Realisierung der gestellten Aufgabe, wurde im Rahmen einer Gruppenarbeit das Entity-Relationship-Model entwickelt, um eine fundierte Datenbasis zu erstellen:

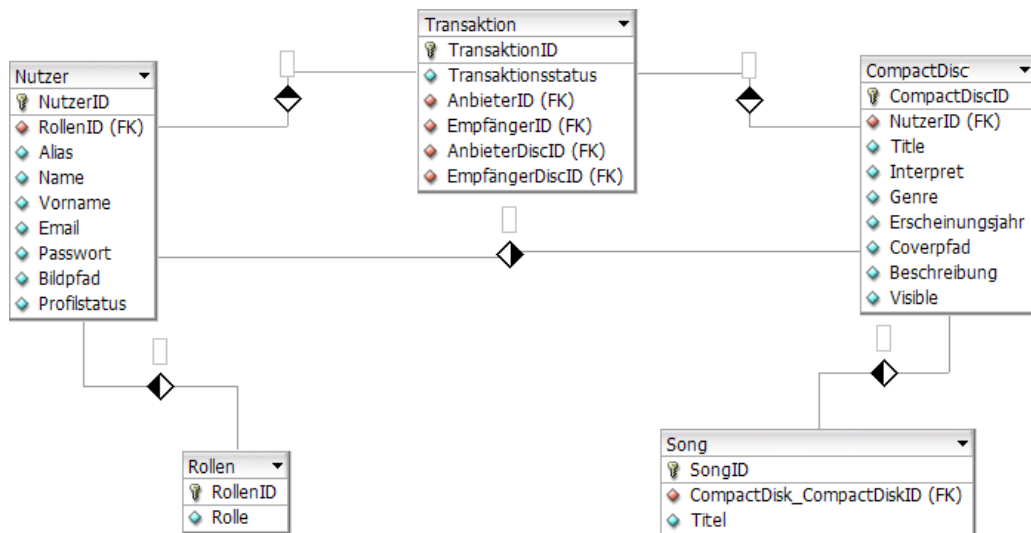


Abbildung 4.4.: Erste Version des ER-Modells für die CD-Tauschbörse

Um eine Arbeitsgrundlage zu erstellen, wurde zuerst das Datenbankschema in Rails erstellt. Rails bietet dazu einen ActiveRecord an der als Layer zwischen dem SourceCode und der Datenbank fungiert. Dazu wird für jede Tabelle ein Model erstellt. Zwar bietet Rails die Möglichkeit mit einem einzigen Kommando sowohl Model, Controller und Views zu generieren. Davon wurde aber kein Gebrauch gemacht, da es entschieden wurde die Anwendung Schritt für Schritt zu entwickeln, um größtmögliche Kontrolle zu erreichen. Es wurden fünf Models mit dem Kommando *rails generate model <Bezeichnung>* erstellt. Das einzige was Rails noch macht ist für jedes Model eine TestKlasse anzulegen. Nachdem alle erforderlichen Modelle angelegt wurden, konnte das Schema in die Datenbank migriert werden mit dem Ziel das Modelierte Schema zu testen. Rails bietet dazu eine geeignete Testumgebung an. Hierzu wurden die Unit Tests entwickelt. Auf diese Art und Weise wird es sichergestellt, dass das Datenbankschema einwandfrei funktioniert ohne auch dafür eine einzige Zeile Code geschrieben (z.B. für Controller oder View) zu haben. Basierend auf einer funktionierenden Datenbank werden die einzelnen Views und Controller integriert. Die Änderungen im Datenbankschema können nachträglich ohne Probleme integriert werden.

Nach der Bekanntgabe der Änderungen im Pflichtenheft, wurde sofort ersichtlich, dass das erstellte Datenbankschema modifiziert werden soll. Für die

Realisierung eines Austauschs mit mehreren CD's, wurde die n:m Beziehung bei den Transaktionen hinzugefügt. Dabei wurden zwei zusätzliche Tabellen mit aufgenommen. Darüber hinaus sind weitere Attribute für die Realisierung anderer Funktionalitäten hinzugekommen. Folgende Abbildung stellt das endgültige Schema der Datenbasis dar:

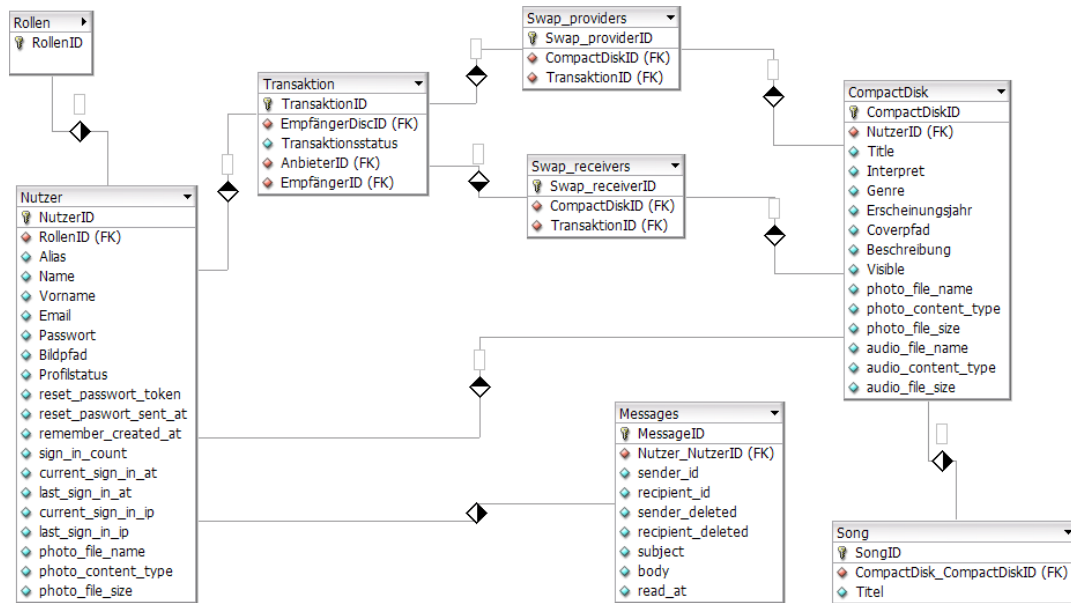


Abbildung 4.5.: ER-Model für die CD-Tauschbörse

4.3. Realisierung der Funktionalität

Die Grundlage der Anwendung, wurde mit dem Erstellen von Models, geschaffen. Für die Realisierung der geforderten Funktionalität wurden Gedanken über die benötigten Controller und Views gemacht werden. Controller werden in Rails mittels des Kommandos `rails generate controller <Bezeichnung>` erstellt. Zu jedem Controller können mehrere Views gehören. In der Datei `routes.rb` wird definiert welche Methode im Controller auszuführen ist, wenn eine bestimmte View im Browser aufgerufen wird. Des Weiteren wurden, für jede Anforderung im Pflichtenheft, User Stories geschrieben, um die Aufgaben besser zu verteilen.

4.3.1. Benutzerverwaltung

Als erstes wurde untersucht, ob eine eigene Implementierung aufwendig ist, oder nicht. Hierzu wurde mittels eines Tutoriell die Benutzerverwaltung rea-

lisiert. Es muss gesagt werden, dass diese Implementierung nur ein Grundgerüst der Autorisierung-Funktionalität darstellt und mehrere Sicherheitsrisiken ausweist. Aus diesem Grund wurde entschieden ein sicherer und gut dokumentierter Plug-In zu verwenden.

Für diese Zwecke wurden mehrere Plug-Ins evaluiert, sodass wir auf die Entscheidung gekommen sind Devise zu verwenden. Diese Erweiterung bietet eine flexible Lösung zur Benutzerauthentifizierung für Rails und besteht aus 12 Modulen wie zum Beispiel:

1. Confirmable
2. Authenticatable
3. Recoverable
4. Registerable
5. Rememberable (Cookies)
6. Validatable
7. Encryptable
8. ...

Alle diese Module sind nach dem MVC-Prinzip implementiert und können nach Bedarf kombiniert werden. Dadurch wird erreicht, dass jede Applikation nur die geforderten Funktionalitäten hat. Die Installation dieses Plug-Ins ist entsprechend einfach und kann nur in wenigen Schritten realisiert werden:

1. Gemfile: `gem 'devise'`
2. `bundle install` -> fehlende Gems installieren
3. `rails generate devise:install` -> Plug-In installieren
4. Migration(en) anpassen oder mit `rails generate devise user` ein neues Model erstellen
5. Migrationen ausführen `rake db:migrate`

Da das Plug-In *Devise* nur die Benutzerauthentifizierung gewährleistet wurde nach einem Rollenverwaltungstool recherchiert, um ein Rollenmanagement zu realisieren, dadurch den Nutzern unterschiedliche Rechte zu geben und dadurch die Anwendung zu schützen. Hierzu wurde *CanCan* analysiert und getestet. *CanCan* ist ein Plug-In, welches auf Devise aufgebaut wird. Nach der Installation wird eine neue Klasse *ability* angelegt, die nach Bedürfnissen konfiguriert werden soll. Somit bietet diese Erweiterung die Definition von mehreren Gruppen (Gast, Benutzer, Admin) sowie eine einfache und vor allem schnelle Rechtevergabe.

4.3.2. Compact Discs

Für die Darstellung von mehreren CD's auf einer Seite wurde das Plug-In *WillPaginate* für Rails untersucht und implementiert. Denn wenn die CD's einem Benutzer angezeigt werden, ist es nicht erwünscht immer alle CD's auf einer Seite zeigen, da die Anzahl sogar mehrere hundert CD's betragen kann. Einerseits wäre diese Darstellung zu unübersichtlich und zum Anderen ist auch die Zeit die der Server für die Antwort der Anfrage benötigen würde wäre viel zu lang. Daher ist es sinnvoll, die CD's auf mehrere Seiten aufzuteilen. Genau das macht *WillPaginate*. Es muss lediglich definiert werden wie viele CD's in der View gezeigt werden müssen. Sind mehr CD's vorhanden als auf der Seite dargestellt, sind diese über weitere Seiten (1,2,...) erreichbar.

4.3.3. Tausch-Transaktionen

Für die Realisierung der Hauptfunktionalität wurde überlegt wie der Austausch von CD's erfolgen kann. Da diese Funktion mit einem Plug-In bzw. durch die Verwendung eines internen Messaging-Systems realisiert werden sollte, wurde ermittelt welche davon in Frage kommen. Es wurde entschieden ein Plug-In für die interne Kommunikation bzw. den Nachrichtenaustausch auszuwählen und für unsere Zwecke zu verwenden. Hierzu wurden mehrere Plug-Ins recherchiert und mit einander verglichen:

- has-messages
- simple-messaging
- simple-private-message

Diese Plug-Ins wurden hinsichtlich der Funktionalität, Dokumentation, Verbreitung sowie der Weiterentwicklung bzw. Support untersucht. Es wurde herausgestellt, dass simple-private-message die erforderlichen Funktionalitäten zur Verfügung stellt und zu den wenigsten Plug-Ins gehört, die fertiggestellt sind und auch mit der Rails Version 3.0.1 kompatibel sind. Darüber hinaus wurde die Funktionalität anhand eines Tutorialis überprüft. Die Installation dieses Plug-Ins ist sehr einfach und benötigt lediglich drei Schritte:

1. Gem File anpassen + *bundle install*
2. *rails generate simple-private-messages:model User Message*
3. Controller "Transaktionen" erstellen

4. Anwendung

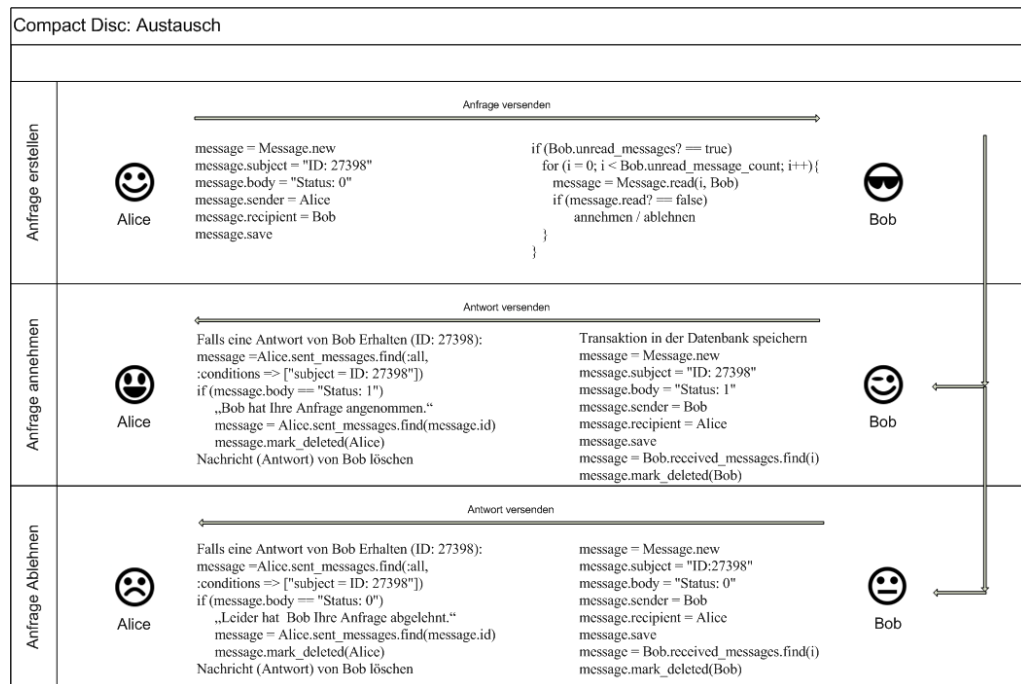


Abbildung 4.6.: Verlauf CD-Tauschbörse

Um die Implementierung der Funktionalität zu vereinfachen wurde die Kommunikation visuell dargestellt. Somit konnten alle Anforderungen hinsichtlich der Sicherheit erfüllt werden.

Bei der Implementierung dieser Funktionalität wurde sowohl auf die Korrektheit, als auch auf die Sicherheit des Transaktionsaustauschs geachtet. Falls die Anfrage abgelehnt wird, werden die zugehörigen Nachrichten gelöscht, ohne einen Datenbank Eintrag zu machen. Wenn der Austausch akzeptiert wird bzw. zustande kommt, werden die CDs ausgetauscht und die gesamte Transaktion in der Datenbank gespeichert. Somit besteht die Möglichkeit es zu jeder Zeit zu prüfen, wer mit wem welche CD's getauscht hat. Am 1. Dezember erfolgten die Änderungen des Pflichtenhefts, so dass es erforderlich war die Vorgehensweise etwas zu Verändern. Darüber hinaus wurden weitere Tabellen in die Datenbank aufgenommen.

Wie im Pflichtenheft gewünscht ist, wurde der Austausch von CDs durch ein internes Messaging-System realisiert. Bei der Implementierung wurde darauf geachtet, dass die Transaktionen sicher durchgeführt werden, sodass in der vordefinierten Reihenfolge die Datenbankeinträge gemacht werden und keine Information verloren gehen können. Nun wurde der Fall nicht berücksichtigt, falls ein Benutzer die CD löscht, die immer noch im Anfragemodus befindet. Nach einer Diskussion wurden zwei Möglichkeiten überlegt:

1. Wenn eine CD gelöscht wird, wird es geprüft ob es die Anfragen mit dieser

CD gibt, oder nicht. Wenn das nicht der Fall ist, kann die CD gelöscht werden. Wenn allerdings eine Anfrage besteht, wird diese automatisch abgebrochen und die Beteiligten erhalten eine Nachricht.

2. Die einfachere Methode ist es das Löschen zu verbieten solange eine Anfrage besteht. D.h. bevor eine CD gelöscht werden kann, müssen alle zugehörigen Abfragen gelöscht werden.

4.3.4. Sonstige Anforderungen

Darüber hinaus wurde die Anforderung erfüllt, den Austausch mit Drag and Drop zu entwickeln. Die Drag and Drop Funktionalität wurde durch jQuery realisiert. Dafür wurde die View, auf der der CD-Tausch vollzogen wird, in vier Bereiche (DIV-Elemente) unterteilt. Alle vier Bereiche bzw. alle darin enthaltenen Elemente (CDs) bieten die Drag und Drop Funktion. Dafür werden die von jQuery bereitgestellten Methoden `draggable` und `droppable` genutzt. Diese Methoden werden den einzelnen Div-Elementen zugewiesen. Zur Identifizierung und Nutzung der Div-Elemente innerhalb der jQuery Javascript Datei, dient die eindeutige ID, welche jedes Element besitzt.

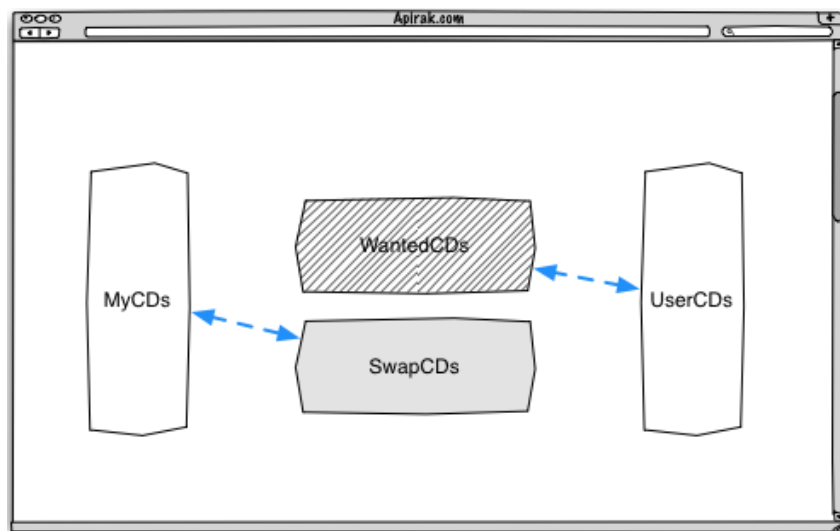


Abbildung 4.7.: Drag nad Drop Funktionalität

Die Grafik zeigt den Schematischen Aufbau der Seite. Die äußeren Bereiche zeigen alle CDs, des aktuell angemeldeten Benutzers (MyCDs) und des Tauschpartners (UserCDs), an. Diese CDs können dann in einen der mittleren Bereiche gezogen werden (WantedCDs oder SwapCDs). Um sicherzustellen, dass die Bereiche WantedCDs und SwapCDs nur Elemente aufnehmen

welche aus einem bestimmten Div-Container kommen, wurden Beschränkungen in jQuery definiert. Dies geschieht durch die Option accept, welches die droppable Elemente bieten.

```
$('#swapCDs').droppable(  
  accept: "#myCDs > img"  
);
```

Wie im Beispiel zu sehen, akzeptiert das Element swapCDs nur img-Elemente aus dem MyCDs Bereich. Dadurch kann verhindert werden, dass ein Nutzer CDs aus der Sammlung des Tauschpartners in beide Bereiche (WantedCDs und SwapCDs) zieht und somit eine Anfrage erstellen kann, ohne eine CD aus seiner eigenen Sammlung zum Tausch angeboten zu haben. Die IDs der CDs, die auf dem Swap oder Wanted Bereich gezogen werden, werden in einem Array gespeichert (Für jeden Bereich ein separates Array). Falls ein Element wieder aus einem der mittleren Bereiche entfernt wird, wird es auch aus dem entsprechenden Array gelöscht. Sobald eine Anfrage Abgeschickt werden soll, werden die Werte des Arrays ausgelesen und als Parameter an den Link zur Action im Controller angefügt.

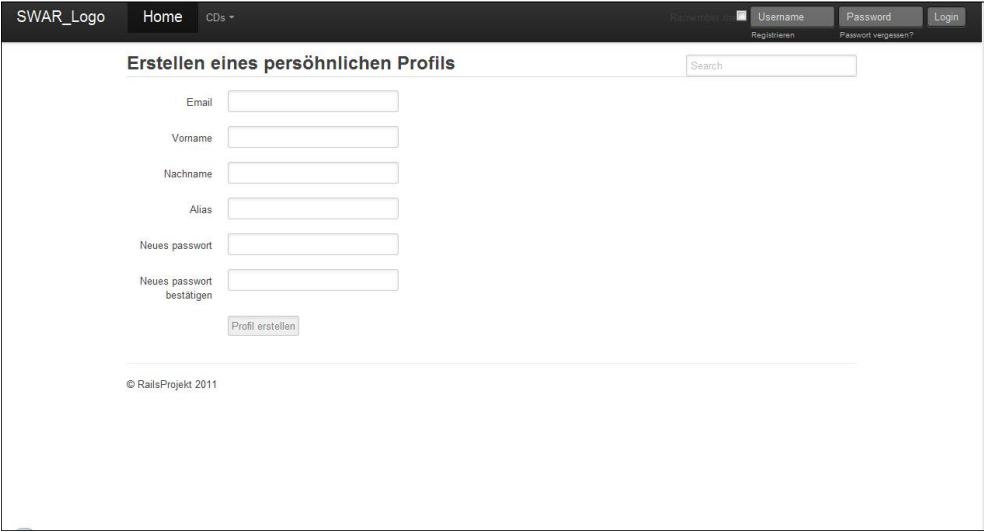
4.4. Graphical User Interface

In der ersten Woche wurden erste Entwürfe zum Design der CD-Tausch-Plattform erstellt. Diese wurden während der Teammeetings in Form von einfachen Skizzen in analoger Form festgehalten und im Anschluss mittels Adobe Photoshop als Mockups ausgearbeitet. Basierend auf diesen Entwürfen wurde ein HTML5 Layout mittels CSS-Stylesheet erstellt.



Abbildung 4.8.: Design 1

Im Laufe der Entwicklung wurde das Design noch einmal überarbeitet, da es dem ersten Entwurf an Struktur fehlte. Im Team wurde entschieden dieses noch einmal zu überarbeiten und dafür ein Template zu nutzen, um eine einheitliche Darstellung zu erzielen. Zur engeren Auswahl standen Blueprint CSS, 960gs und Bootstrap. Die Entscheidung fiel auf Bootstrap von Twitter. Bootstrap ist ein spezielles Toolkit, das im Wesentlichen ein HTML- und CSS-Template bereitstellt, in das Twitter seine Erfahrungen einfließen ließ. Webentwickler, die Bootstrap in ihr Frontend einbinden, sollen so auf praxiserprobte Frontend-Entwurfsmuster zurückgreifen können und unter anderem ohne größeren Aufwand ein Grid-System einbinden, das bei Twitter erprobte Styling für typographische Elemente übernehmen, Formulare und Buttons verwenden oder ihre Seiten mit Modal-Boxen, Tooltips und Popovers ergänzen. Dieses Template enthält viele Lösungsansätze zur Realisierung von Javascript Effekten. So auch die von Tooltips und Popover. Für die Realisierung von den Tooltips haben wir die Funktion `.popover` von Bootstrap verwendet. Dort wird in einer Variable, der anzuzeigenden Text an. Dieser wird in einem Popover angezeigt, wenn mit dem Mauszeiger über die verlinkte Region gefahren wird.



The screenshot displays a web interface for creating a profile. At the top, there is a dark navigation bar with the 'SWAR_Logo' and 'Home' links on the left, and user-related links ('Username', 'Password', 'Login', 'Registrieren', 'Passwort vergessen?') on the right. Below the navigation bar, the main content area is titled 'Erstellen eines persönlichen Profils'. It features a search bar on the right and a series of input fields for 'Email', 'Vorname', 'Nachname', 'Alias', 'Neues passwort', and 'Neues passwort bestätigen'. A 'Profil erstellen' button is located at the bottom of the form. The footer of the page indicates '© RailsProjekt 2011'.

Abbildung 4.9.: Design 2

4.5. Internationalisierung

Da es vor der Bekanntgabe der Änderungen nicht bekannt war, welche Funktionalitäten hinzukommen, wurden Gedanken über eine Internationalisierung der Anwendung gemacht. Unser Team ist davon überzeugt, dass jede gute Webanwendung mehrere Sprachen unterstützen soll, um eine möglichst große Nutzerzahl zu erreichen.

Das Einbinden von verschiedenen Sprachen erfolgt in Rails über eine *localize* Datei. Diese hat den Namen der Sprache, also z.B.: *de.yml*. In den Views werden nun Variablen für die Bezeichnungen eingetragen. In der *localize* Datei steht dann zu jeder Variablen die entsprechende Übersetzung. Fürs erste haben wir uns auf die deutsche Sprache beschränkt, da es hierzu keine Anforderungen gab. Am Ende der Entwicklung ist es sehr einfach andere Sprachen zu implementieren. Die *localize* Datei muss dazu nur kopiert werden. Die Neue Datei bekommt den Namen der neuen Sprache z.B.: *en.yml*. Nun müssen alle Bezeichnungen übersetzt werden, sodass dadurch die Lokalisierung bzw. Internationalisierung realisiert werden kann.

Damit Rails auch weiß, für welche Komponenten eine Lokalisierung zur Verfügung steht, muss diese noch in den jeweiligen Controllern ein gepflegt werden. Dazu müssen auch für alle Routen entsprechende Einträge vorgenommen werden. Dabei zählt auch welche Sprachen unterstützt werden sollen. Das sind aber nur einmalige Anpassungen, die wenig Zeitaufwand ausweisen. Herauszufinden welche Sprache der Benutzer hat kann über die HTTP Header geschehen. Alternativ kann der Benutzer selber eine Sprache auswählen. Eine Sprache kann auch als Standard gesetzt werden wenn keine Informationen vorliegen. Dabei wurde herausgestellt, dass eine Anwendung zu internationalisieren erfordert nur einmalig ein wenig Aufwand. Danach kann man aber leicht die Anwendung um jede gewünschte Sprache erweitern.

4.6. File Upload (Paperclip)

In der Webanwendung sollen zwei Arten von Dateien hochgeladen werden können: Bilder und Audio. Um Dateien zum Server zu laden stehen für Rails zwei Plugins in Rails zur Verfügung *Attachment_fu* und *Paperclip*². Wir haben uns für *Paperclip* entschieden das etwas flexibler als *Attachment_fu* ist. Möchte man Bilder mit *Paperclip* zum Server hochladen wird zusätzlich noch das Tool *ImageMagick* benötigt, welches in der Lage ist verschiedene Versionen eines Bildes zu erstellen sowie die Meta-Informationen auszulesen. Im Model wird definiert welche Art von Datei hochgeladen werden kann. So sollen in für den User als auch für die CD's Bilder hochgeladen werden können. Dazu müssen in der Datenbank zusätzliche Spalten angelegt werden. *Paperclip* speichert die hochgeladenen Daten nicht in der Datenbank sondern auf dem Dateisystem. In der Datenbank wird lediglich die URL zu der Datei gespeichert. Das macht den ganzen Prozess wesentlich performanter, da die Datenbank nicht so vollgestopft wird. Das hochladen von Audio funktioniert fast genauso, nur das man hier auf andere Mime-Types prüft. Es hat sich gezeigt das verschiedene Browser und auch verschiedene Plattformen unterschiedlich mit den Mime-Types

²<https://github.com/thoughtbot/paperclip>

umgehen. Mit Paperclip kann sehr schnell ein Datei Upload eingerichtet werden. Im folgenden ist zu sehen wie ein Attachment dem Model hinzugefügt werden kann inklusive Validierung. Im folgenden ist für das Model User die Anpassung zu sehen um ein Cover hochzuladen. Die entsprechenden Attribute müssen in der Datenbank in der entsprechenden Tabelle über eine Migration eingefügt werden.

```
1 has_attached_file :photo, :styles => { :normal => "150x150>", :small =>
  "70x70>" },
2 :url => "/system/users/:id/:style/:basename.:extension",
3 :path =>
  ":rails_root/public/system/users/:id/:style/:basename.:extension",
4 :default_url => "/assets/user_default_:style.png"
5
6 validates_attachment_size :photo, :less_than => 5.megabytes
7 validates_attachment_content_type :photo, :content_type => ['image/jpeg',
  'image/png']
8
9 t.string      "photo_file_name"
10 t.string     "photo_content_type"
11 t.integer    "photo_file_size"
```

4.7. MusicBrainz

MusicBrainz³ ist ein Webservice ueber den Daten zu Alben und Kuenstlern abgerufen werden können. Die API erlaubt Abfragen. Dabei kann man entweder allgemein suchen dann bekommt man eine Ergebnisse liste die nach einen Score sortiert ist. Man kann aber auch speziell nach einer eindeutigen ID der MBID suchen. Unsere Vorstellung war es MusicBrainz zur Erstellung einer CD zu verwenden. Der Benutzer muss dazu mindestens das Album und den Kuenstler angeben. Diese Daten sind fest und werden von MusicBrainz nicht verändern. Anhand dieser Daten wird versucht den Künstler und das Album zu finden. War die Suche erfolgreich wird nach dem Erscheinungsjahr und Tracks geschaut. Diese werden in die Form eingetragen und können vom Benutzer noch bearbeitet, ergänzt oder gelöscht werden bevor das Formular abgeschickt wird und die CD erstellt wird. Gleichzeitig wird eine Amazon Standard Identifikation Number (asin) gespeichert welche die CD eindeutig bei Amazon identifiziert. Darüber kann eine URL erstellt werden, über welche ein Cover heruntergeladen werden kann. Diese hat die Form <http://images.amazon.com/images/P/<ASIN>.jpg>, wobei ASIN irgendeine eindeutige Nummer darstellt. Ist ein Cover vorhanden wird das über die URL heruntergeladen, so das der Benutzer selbst kein Cover hochladen muss.

³<http://musicbrainz.org/>

```
1 # Songs eines Albums
2 def searchMBrainz(art, alb)
3   query = Webservice::Query.new
4   filter = Webservice::ReleaseFilter.new(:title => alb, :artist => art)
5   release = query.get_releases(filter)
6   if (!release.empty?)
7     # get mbid of first
8     mbid = release.entities[0].id.to_s;
9
10    entities = release.entities
11    asin = ""
12    year = nil
13    tracks = nil
14
15    entities.each { |e|
16      logger.debug "next entity"
17      mbid = e.id.to_s
18      r = query.get_release_by_id(mbid, :artist=>true, :tracks=>true,
19                                :release_events => true)
20      if (!r.asin.nil?)
21        asin = r.asin.to_s
22        if year.nil?
23          year = r.release_events[0].date.year
24        end
25        if tracks.nil?
26          tracks = r.tracks
27        end
28        break
29      end
30    }
31
32    cover_url = nil
33    unless asin.nil?
34      cover_url = generate_cover_url(asin)
35    else
36      cover_url = ""
37    end
38
39    results_from_rbrainz = {
40      :tracks => tracks.to_a,
41      :cover_url => cover_url,
42      :year => year
43    }
44
45    return results_from_rbrainz
46  end
47  return nil
48 end
```

4.8. Tests

Wir haben das Standard Testframework von Rails verwendet um umfangreiche Unit Tests zu schreiben. Dies basiert auf der Idee des Test-Driven-Development. Also zuerst Tests schreiben, schauen wie sie fehlschlagen und dann die Funktionalität implementieren und prüfen ob der Test erfolgreich ist. Die Unit Tests sichern unsere Grundfunktionalität. Des Weiteren haben wir mit Cucumber Tests geschrieben. Mit Cucumber (Behavior-Driven-Development)

ist es möglich Tests in Prosaform zu schreiben. Das macht die Tests besser lesbarer auch für nicht Informatiker. Die Tests werden häufig als User Stories geschrieben in der Form: "Als <Rolle>, möchte ich <Ziel/Wunsch>, um <Nutzen>". Umfangreiche Tests wie sie in einem realen Projekt gemacht werden sollten konnten wir aufgrund der beschränkten Ressourcen jedoch nicht durchführen. Rails eignet jedoch wunderbar für ein Test-Driven-Development da es ein Standard Testframework integriert. Hier ist exemplarisch ein Unit Tests für das Model User zu sehen. So wird z.B. auf korrekte Form der eMail Adresse getestet und ob die Attribute Name, Vorname, eMail und Passwort gesetzt sind.

```
1 require 'test_helper'
2
3 class UserTest < ActiveSupport::TestCase
4   # set up test data for table user
5   fixtures :roles, :users
6
7   def setup
8     @user = users(:peter)
9   end
10
11   def teardown
12     @user = nil
13   end
14
15   test "user attributes must not be empty" do
16     user = User.new
17     assert user.invalid?
18     assert user.errors[:firstname].any?
19     assert user.errors[:lastname].any?
20     assert user.errors[:email].any?
21     assert user.errors[:password].any?, "password must mut not be empty"
22   end
23
24   test "email" do
25     @user.email = nil
26     assert @user.invalid?, "email must exist"
27   end
28
29   test "user email must be unique" do
30     user = User.new(:email => users(:peter).email,
31                     :firstname => "vorname",
32                     :lastname => "nachname",
33                     :password => "abcdef12sdf2",
34                     :state => "active")
35     assert !user.save
36     #assert_equal I18n.translate('activerecord.errors.messages.taken'),
37     #      user.errors[:email].join(';')
38   end
39
40   #test "image uri" do
41   #  @user.image_uri = "image.sh"
42   #  assert @user.invalid?, "image uri should be invalid"
43
44   #  @user.image_uri = "image.png"
45   #  assert @user.valid?, "image uri should be valid"
46
47   #  @user.image_uri = nil
48   #  assert @user.valid?, "image uri is allowed to be nil"
```

4. Anwendung

```
48
49 # @user.image_uri = ""
50 # assert @user.invalid?, "image uri is must not be empty"
51 #end
52
53 test "email pattern" do
54   @user.email = "name@domain.com"
55   assert @user.valid?, "email pattern should be valid"
56
57   @user.email = "@me.com"
58   assert @user.invalid?, "Should fail due starts with an @ sign"
59
60   @user.email = "testme.com"
61   assert @user.invalid?, "Should fail due to missing @ sign"
62
63   @user.email = "test@mecom"
64   assert @user.invalid?, "Should fail due to missing . sign in domain
    part"
65
66   @user.email = "test@me."
67   assert @user.invalid?, "Should fail due no top-level domain is given"
68
69   @user.email = "test@"
70   assert @user.invalid?, "Should fail due no domain is given"
71 end
72
73 test "role id must exist to create new user" do
74   @user.email = "abc@htw.de"
75   @user.role_id = 0
76   assert @user.save, "Should save user due role_id exist"
77
78   @user.email = "abcd@htw.de"
79   @user.role_id = 42
80   assert !@user.save, "should not save user due role_id does not exist"
81 end
82
83 =begin
84   test "search user" do
85     array = UserController.search("P")
86     assert array.count == 1, "should contain exactly one user with
    firstname: Peter"
87     assert array[0].firstname == "Peter", "Firstname should be Peter"
88
89     array = UserController.search("Sch")
90     assert array.count == 2, "should contain exactly two users with
    lastname: Schmidt"
91     assert array[0].firstname == "Peter", "Firstname should be Peter"
92     assert array[1].firstname == "Hannah", "Firstname should be Hannah"
93
94     array = UserController.search("@hotmail.de")
95     assert array.count == 1, "should contain exactly one user with
    firstname: Christian"
96
97     array = UserController.search("schmiddy")
98     assert array.count == 2, "should contain exactly two users"
99   end
100 =end
101
102 end
```

Hier ist zu sehen wie Tests mit Cucumber geschrieben werden. Zuerst werden sogenannte Features definiert. Diese sind in der <Gegeben> <Ereignis> <Ergebnis> geschrieben. Die feature sind dabei an User Stories angelegt. Diese Art

4. Anwendung

der Tests sind leicht nachzuvollziehen. So können z.B. auch die Auftraggeber eines Projekts welche evtl keine Informatiker sind einbezogen werden.

```
1 Feature: Authentication
2   As a user
3   I want to login at the plattform
4   So I can share my CDs with other people
5
6   @ok
7   Scenario: Click on Register
8     Given I am on the Welcome page
9       When I follow "Register"
10        Then I should see a "Firstname" field
11        And I should see a "Lastname" field
12        And I should see an "Email" field
13        And I should see a "Password" field
14        And I should see a "Password Confirmation" field
15        And I should see an "Alias" field
16
17   @ok
18   Scenario: Register new Account
19     Given I am on the Register page
20     When I fill in "Firstname" with "Christian" in the registration form
21     And I fill in "Lastname" with "Bunk" in the registration form
22     And I fill in "Password" with "qwertz" in the registration form
23     And I fill in "Password Confirmation" with "qwertz" in the
24       registration form
25     And I fill in "Email" with "christianb@gmail.com" in the registration
26       form
27     And I press "Erstellen"
28     Then I should see "Alle CDs"
29
30   @ok
31   Scenario: Login User (successful)
32     Given I am on the Welcome page
33     And I should see an "Email" field
34     And I should see a "Password" field
35     When I fill in "Email" with "christianb@web.de" in the login form
36     And I fill in "Password" with "christianb" in the login form
37     And I press "Login"
38     Then I should see "Alle CDs"
39
40   @wip
41   Scenario: Login User (unsuccessful)
42     Given I am on the Welcome page
43     And I should see an "Email" field
44     And I should see a "Password" field
45     When I fill in "Email" with wrong value "christ@web.de" in the
46       login form
47     And I fill in "Password" with wrong value "christian" in the login
48       form
49     And I press "Login"
50     Then the login should not be successful
51
52   @wip
53   Scenario: Logout User
54     Given I am logged in
55     When I follow "Logout"
56     Then I should see "Logout Erfolgreich"
```

Natürlich müssen die Features und Szenarien auch automatisiert abgearbeitet werden. dazu findet ein Pattern Matching statt. Trifft eine Regel zu wird der

4. Anwendung

definierte Kurs dieser Regel ausgeführt: Hier sind die Regeln und das Pattern Matching zu sehen.

```
1 When /^I fill in "[^"]*" with "[^"]*" in the edit form$/ do |field,
  value|
2   #within("#user_edit") do
3     #fill_in(field_to(field), :with => value)
4   #end
5   fill_in(field_to(field)), :with => value
6 end
7
8 When /^I fill in my Password$/ do
9   within("#user_edit") do
10     fill_in(field_to("Password confirmation"), :with => "christianb")
11   end
12
13 end
14
15 When /^I delete my Account$/ do
16   delete path_to("Account loeschen")
17 end
```

5. Zusammenfassung

In diesem Kapitel fassen die Projektteilnehmer ihre Erfahrungen nach dem Projekt zusammen.

5.1. Antonia Ziegler

Insgesamt würde ich Ruby on Rails als ein sehr modernes und zukunftsorientiertes Framework mit einer grossen Community bezeichnen. Die Entwicklungsumgebung lässt sich insgesamt gut auf verschiedenen Betriebssystemen aufsetzen. Wenn man sich an das gegebene MVC Prinzip hält, ist ein Ruby on Rails Projekt gut erweiterbar und wartbar. Was mir am besten gefällt ist die Möglichkeit, oft ziemlich einfach, Plugins einzubinden. Auf Grund der grossen Community gibt es viele Open Source Plugins, welche Funktionen bieten, wie z.B. die Pagenavigation. Diese wiederum sind gut erweiterbar oder anpassbar.

5.2. Christian Sandvoss

Ich finde, das man Rails sehr gut verwenden kann um auch umfangreiche Webanwendung zu realisieren. Rails stellt ein Grundsystem bereit, das durch viele Plugins erweiterbar ist. Allerdings muss man dabei darauf achten ob das Plugin zur genutzten Rails Version kompatibel ist oder überhaupt noch weiter entwickelt wird. Des weiteren ist bei einigen Plugins die Dokumentation sehr kurz gehalten. Die Dokumentation von Rails ist dagegen, wie ich finde, sehr gut. Es gibt Tutorials zum Einstieg sowie eine aktive Community. Neu für mich war die Arbeit mit JavaScript bzw. JQuery. Es war für mich sehr interessant das Zusammenspiel von verschiedenen Sprachen (Ruby, HTML/CSS, JavaScript) kennenzulernen.

5.3. Alexander Miller

Durch die Realisierung des Projektes konnte ich persönlich feststellen welche Vor- und Nachteile Rails ausweist. Die Verwendung von Rails, können sogar komplexe RIA Web-Anwendungen realisiert werden. Die wesentlichen Prinzipien dieses Frameworks sind Konvention over Configuration und MVC Prinzip. Im Allgemeinen ist Rails sehr gut dokumentiert und hat eine grosse und vor

allem aktive Community. Der Umfang an Möglichkeiten ist enorm gross, so dass viele Funktionalitäten problemlos implementiert werden können. Darüber hinaus besteht die Möglichkeit viele Module als Plug-Ins zu integrieren. Der Prozess der Integration von PlugIns ist trivial wenn dieser ausreichend Dokumentiert ist. Leider sind manche PlugIns schlecht dokumentiert, so dass die Auseinandersetzung manchmal viel Zeitaufwand erfordert. Insgesamt bin ich mit dem Framework sehr zufrieden, so dass es nichts dagegen spricht für die Realisierung einer weiteren Web-Applikation Rails zu verwenden.

5.4. Christian Bunk

Ich muss sagen das sich meine Einstellung zu Webframeworks sehr geändert hat. Ich hatte früher nie Lust auf Webentwicklung, da mir Frameworks wie Spring und Struts viel zu kompliziert, zu sperrig, ja sich einfach nicht schön an die Entwicklung von Web Applicationen angefühlt haben. Aber mit Ruby on Rails hat sich das sehr geändert. Schon die Programmiersprache Ruby macht das schreiben von Programmen sehr angenehm. Es ist eine elegante und moderne Sprache. Darauf ein Webframework aufzubauen erscheint nur logisch. Mit Ruby on Rails hat man das Gefühl das alles gut durchdacht ist. Durch das Prinzip Convention over Configuration ist es möglich mit wenig aufwand viel zu erreichen. Alles ist da wo es sein soll und wo es hingehört. Hält man sich an bestimmte Regeln des Frameworks macht die Entwicklun einfach Spass. Das bedeutet aber nicht das die Entwicklung leicht ist. Im Gegenteil Rails ist ein Framework mit sehr grossem Funktionsumfang. Selbst in den 3 Monaten unserer Entwicklung haben wir in vielen Bereich nur an der Oberfläche gekratzt. Rails bietet eine wunderbare und umfangreiche Dokumentation. Aber um hier hinter alle Konzepte zu steigen braucht es einfach Zeit. Rails bietet mit Erweiterungen durch Module oder Plugins die Möglichkeit Funktionen in seine Webanwendung zu integrieren. Das ist gut solange man nicht mehr Funktionalität braucht als die jeweiligen Plugins bieten. Schwierig wird es wenn man doch eigene oder zusätzliche Funktionen benötigt. Dann muss man versuchen um das Plugin herum zu entwickeln, da meist genaue Dokumentation für die sehr speziellen Fälle fehlen. Mit Rails habe ich auch das Routing, also das mappen von URLs auf bestimmte Methoden in einem Controller, besser verstanden. Das Konzept von Active Record hat mich sehr Überzeugt, da hier Schnittstellen und Datenbanken optimal verschmelzen und nicht mehr an jeder Stelle im Code die Datenbank kryptisch ausgelesen oder konfiguriert werden muss. Es wird einfach ein Model definiert, über welches man die Attribute definiert. Diese werden dann automatisch in die Datenbank eingetragen. Rails eignet sich sehr gut für verteilte Entwicklung. Es werden weder Lizenzen benötigt, noch wird dem Programmierer eine IDE aufgezwungen. Alles kann in einem einfachen Texteditor entwickelt werden. Mehr als einen Browser und ein Terminal

braucht man dann nicht. Entgegen manchen Vorstellungen das Rails einfach sei muss ich widersprechen. Man braucht viel Zeit um die Konzepte sich zu eigen zu machen. Auch braucht man (wie bei jedem anderen Framework) Zeit um Funktionalitäten zu programmieren. Ich denke eh nicht das man Frameworks nach leichter und schwerer Kategorisieren kann. Ich denke das ich mit Rails wertvolle Erfahrungen gesammelt habe die mir im meinem späteren Berufsleben nützlich sein werden. Ich denke auch das ich mit Rails noch öfter in Zukunft in Berührung kommen werde. Rails muss man einfach erlebt haben.

A. Anhang

A.1. Aufgabenstellung

Pflichtenheft

CD-Tauschbörse

Systementwicklung und Frameworks
Wintersemester 2011/12

geändert: 2011-12-01

geändert: 2011-12-15

1 Zielbestimmung

Die Anwendung realisiert eine Tauschbörse für Audio-CDs. Registrierte Mitglieder können Kontaktprofile erstellen, CDs anbieten und mit anderen Mitgliedern tauschen. Gäste und Mitglieder können Profile und CDs ansehen und CDs bewerten. Gäste können sich auch registrieren, um selbst Mitglieder zu werden.

~~Jeder Benutzer kann eine Tauschanfrage auf eine fremde CD stellen. Der Besitzer kann ablehnen oder ein Tauschangebot machen, welches der erste Benutzer annehmen oder ablehnen kann. Tauschanfragen und Bestätigung/Ablehnung werden über einen internen Nachrichtendienst realisiert. Bei erfolgtem Tausch werden die CDs der Sammlung des jeweiligen Benutzers hinzugefügt.~~

Jeder Benutzer kann ein Angebot abgeben, welches mindestens eine Wunsch-CD enthält. Das Angebot kann abgelehnt oder angenommen werden. Alternativ kann auf das Angebot mit einem modifizierten Angebot (Gegenangebot) geantwortet werden.

Die Webanwendung realisiert ein Rollen- und Sicherheitskonzept. Der Tausch von CDs ist als Transaktion zu realisieren. Die Anwendung soll eine Rich Application sein, welche auch verschiedene Medien (Audio) integriert.

2 Produktfunktionen

2.1 Benutzer

/PF01/	Am System registrieren
Akteur	Gast ¹
/PF02/	Am System anmelden
Akteur	Gast
/PF03/	Eigene Profildaten ändern
Akteur	Benutzer ²
Zusatz	nur PD02, PD03, PD04, PD05, PD06, PD7
/PF04/	Eigenes Benutzerkonto löschen
Akteur	Benutzer

1 Gast = Ein am System aktuell nicht angemeldeter Benutzer.

2 Benutzer = Ein am System aktuell angemeldeter Benutzer.

/PF05/	Beliebiges Benutzerprofil einsehen
Akteur	Benutzer
Zusatz	PD02, PD03, PD04, PD06, PD07
/PF06/	Beliebiges Benutzerprofil einsehen
Akteur	Administrator
Zusatz	nur PD01, PD02, PD03, PD04, PD06, PD07
/PF07/	Beliebiges Benutzerkonto löschen
Akteur	Administrator
/PF08/	Benutzer suchen
Akteur	Benutzer, Administrator
Zusatz	Suche über Teiltex te (Substring)
Zusatz	nur PD02, PD03, PD04, PD06
/PF09/	Passwort vergessen?
Akteur	Gast
Zusatz	neues Passwort wird an Mailadresse verschickt

2.2 CDs

/PF10/	beliebigen Showcase ansehen
Akteur	Gast, Benutzer, Administrator
/PF11/	zu jeder CD weitere Infos ansehen
Akteur	Gast, Benutzer, Administrator
/PF12/	zu jeder CD Musikprobe hören (falls vorhanden)
Akteur	Benutzer
/PF13/	CD in eigenen Showcase hochladen
Akteur	Benutzer
/PF14/	CD aus eigenem Showcase entfernen
Akteur	Benutzer

/PF15/ nach CDs suchen
Akteur Benutzer
Zusatz Suche über Teiltex te (Substring)
Zusatz nur PD10, PD11, PD13, PD14

2.3 Tauschangebote

~~/PF16/ Anfrage erzeugen~~

~~Akteur Benutzer~~

~~/PF17/ Anfrage ablehnen~~

~~Akteur Benutzer~~

/PF18/ Angebot zu Anfrage erzeugen

Akteur Benutzer

/PF19/ Angebot annehmen

Akteur Benutzer

/PF19/ Angebot ablehnen

Akteur Benutzer

~~/PF20/ Angebot modifizieren und zurückschicken (Gegenangebot)~~

~~Akteur Benutzer~~

~~/PF20/ erhaltene Anfragen anschauen~~

~~Akteur Benutzer~~

~~/PF21/ meine Anfragen anschauen~~

~~Akteur Benutzer~~

/PF20/ erhaltene Angebote anschauen

Akteur Benutzer

/PF21/	meineabgegebene Angebote anschauen
Akteur	Benutzer

3 Produktdaten

3.1 Benutzerdaten

/PD02/ E-Mail Adresse

/PD03/ Vorname

/PD04/ Nachname

/PD05/ Passwort

Zusatz verschlüsselt dargestellt

/PD06/ Pseudonym (optional)

/PD07/ Benutzerbild (optional)

/PD08/ Zeitpunkt der Registrierung

3.2 CD-Daten

/PD10/ Titel

/PD11/ Interpret

/PD12/ Cover

/PD13/ Genre (Freitext, optional)

/PD14/ Erscheinungsjahr (optional)

/PD15/ Beschreibung (optional)

~~Anfrage-Daten~~

~~/PD16/ Wunsch-CD~~

~~anfragender Nutzer~~

~~angefragter Nutzer~~

3.3 Angebots-Daten

/PD2016/ Wunsch-CDs (mindestens eine)

/PD2117/ Tausch-CDs (beliebig)

/PD22/ anfragender Nutzer

/PD23/ angefragter Nutzer

/PD24/ Text

4 sonstige Anforderungen

4.1 Usability

/PS01/ Angebotserstellung über Drag & Drop

/PS02/ Darstellung von CD im Angebot durch Cover

~~**/PS03/** Beschreibung von CD im Angebot durch Tooltip~~

4.2 System

/PS04/ Angebotsworkflow über internes Nachrichtensystem

/PS05/ Transaktionssicherheit

/PS06/ Rollenkonzept mit Authentifizierung