

Institut für Informatik

**Bachelorarbeit**

**Entwicklung einer Webapplikation zur  
kollaborativen Erstellung und  
Auswertung von Testdatensets für die  
Stimmungsanalyse von Tweets**

Miriam Beutel

13. April 2015

Erstgutachter: Prof. Dr. Oliver Vornberger

Zweitgutachterin: Prof. Dr. Elke Pulvermüller



## Danksagungen

Ich möchte mich herzlich bei den Personen bedanken, welche mich während der Bachelorarbeit unterstützten und damit erst diese Arbeit möglich gemacht haben:

- Herrn Prof. Dr. Oliver Vornberger für die Tätigkeit als Erstgutachter
- Frau Prof. Dr. Elke Pulvermüller für die Tätigkeit als Zweitgutachterin
- Nils Haldenwang für die exzellente Betreuung und die Bereitstellung der Thematik
- Dennis Altenhoff, Bianca Freitag, Hannes Janott, Lukas Kalbertodt, Eric Lanfer, Michel Meyer, Catharina Neus und Christian Ventker für das Lesen und Korrigieren der Arbeit
- Hendrik Langebrake, Dennis Altenhoff, Kevin Trebing, Svantje Jung, Niels Meyering, Julian Kniephoff, Alexander Tessmer, Manuel Schwarz, Till Grenzdörffer, Christian Heiden, Lukas Kalbertodt und Johan von Behren für das Testen der Applikation, das Aufdecken der letzten Fehler und das Feedback zum Testprozess
- Meinem Vater Dietmar Beutel für den nötigen kritischen Blickwinkel und die vielen hilfreichen Vorschläge - Und besonders dafür, mich auf den Weg des Studiums der Informatik gebracht zu haben.
- Meinen Eltern Ulrike Beutel und Dietmar Beutel für das Ermöglichen dieses Studiums
- Meinem Freund Falk-Gerrit Neus für die Unterstützung in dieser anstrengenden Zeit



## Zusammenfassung

Es gibt viele Interessenten an einer Analyse der Stimmung in Sozialen Netzwerken, in welchen zur heutigen Zeit viele Menschen öffentlich ihre Meinung äußern. Dies ist mit einer Stimmungsanalyse von Tweets automatisiert möglich, indem diese selbstständig in Stimmungskategorien, sogenannte Labels, eingeordnet werden. Je genauer die Bewertungen des automatisierten Labelers mit denen menschlicher Labeler übereinstimmen, desto höher ist die Qualität des Programms. Um dies zu erreichen, werden Testdatensets benötigt, welche aus Tweets bestehen, die bereits von Menschen mit einem der möglichen Klassenlabels bewertet wurden. Um statistisch sinnvolle Ergebnisse zu erhalten, geschieht dies idealerweise nicht nur durch eine Person, sondern durch möglichst viele. In ein Datenset gelangen dann nur diejenigen gelabelten Tweets, bei denen die Bewertungen aller Personen zu einem gewissen Grad übereinstimmen. Im Rahmen dieser Arbeit wird mit agilen Entwicklungsmethoden eine Webapplikation entworfen und umgesetzt, welche die kollaborative Erstellung von Datensets der beschriebenen Art für den Labeler möglichst intuitiv gestaltet und die Sichtung der Eckdaten sowie die Verwaltung der Datensets für den Administrator übersichtlich und simpel hält.

## Abstract

There are many parties interested in sentiment analysis of social networks, where nowadays many people express their opinions. With the sentiment analysis of tweets this is possible to be automated, by automatically assigning mood categories, so-called Labels. The more accurate the assignment by the automatic Labeler compared to decisions of human Labelers, the higher the quality of the automatic Labeler program. To achieve this, test datasets are needed, consisting of human labeled tweets. To provide statistically significant results, the contained tweets should not be labeled by only one person, but by as many as possible. Only those tweets, labeled with the same Label by a certain amount of people, will reach the finished dataset. In this Bachelor-Thesis a webapplication will be designed and implemented by agile development methods. This webapplication will be designed to simplify the process of collaboratively creating datasets for the Labelers, and the management of the datasets and comprehension of key data for the administrator.



# Inhaltsverzeichnis

<b>1</b>	<b>Einleitung</b>	<b>1</b>
1.1	Motivation . . . . .	1
1.2	Zielsetzung und Aufbau der Arbeit . . . . .	2
<b>2</b>	<b>Technologien und Paradigmen</b>	<b>3</b>
2.1	Ruby on Rails . . . . .	3
2.1.1	Ruby . . . . .	3
2.1.2	Don't Repeat Yourself . . . . .	5
2.1.3	Convention over Configuration . . . . .	7
2.1.4	Model-View-Controller Prinzip . . . . .	8
2.1.5	MVC-Beispiel aus der Applikation . . . . .	9
2.1.6	Zusammenfassung . . . . .	13
2.2	Agile Softwareentwicklung . . . . .	13
2.2.1	Die Paradigmen . . . . .	14
2.2.2	Anwendbarkeit auf diese Arbeit . . . . .	15
2.2.3	Agilität in Rails . . . . .	16
2.3	Unit Testing . . . . .	16
2.3.1	Unit Testing in Rails . . . . .	17
2.4	Zusammenfassung . . . . .	19
<b>3</b>	<b>Die Applikation</b>	<b>21</b>
3.1	Anforderungen . . . . .	21
3.2	Front-End . . . . .	22
3.2.1	Pageflow . . . . .	23
3.2.2	Benutzeroberfläche . . . . .	26
3.3	Back-End . . . . .	29
3.3.1	Die Modelklassen . . . . .	30
3.4	Datensets . . . . .	31
3.4.1	Import . . . . .	32
3.4.2	Export . . . . .	32
3.5	Labelprozess . . . . .	33
3.6	AJAX . . . . .	36
3.6.1	AJAX auf der Dataset Show Page . . . . .	37
3.6.2	AJAX auf der Labelpage . . . . .	39
3.7	Testphase . . . . .	40
3.7.1	Laufzeitoptimierung . . . . .	40
3.7.2	Rückmeldung der Tester . . . . .	45
<b>4</b>	<b>Reflexion, Zusammenfassung und Fazit</b>	<b>47</b>
	<b>Literaturverzeichnis</b>	<b>48</b>





# 1 Einleitung

Heutzutage ist das Internet ein Ort des sozialen Austausches. Eine Studie des Bitkom <sup>1</sup> ergab, dass etwa zwei Drittel aller Internetnutzer in Sozialen Netzwerken aktiv sind. Eines der populäreren Sozialen Netzwerke ist Twitter, welches aktuell circa 250 Millionen aktive Nutzer hat <sup>2</sup>. Dort verfassen diese regelmäßig Kurznachrichten, sogenannte Tweets. Darin sind Meinung, Stimmung et cetera der Nutzer enthalten. Es sammelt sich also eine große Menge an Informationen auf Twitter an. Diese Daten sind nutzbar, zum Beispiel im Wahlgeschehen der Politik (Diakopoulos and Shamma (2010)) oder im Marketing (Saif et al. (2012)). Dort sind sie beispielsweise einsetzbar für personalisierte Werbung und Produktplatzierungen. Außerdem können mithilfe dieser Daten Vorhersagen für den Aktienmarkt getroffen werden (Bollen et al. (2011)).

## 1.1 Motivation

Wie ist es möglich sinnvoll auf die auf Twitter angesammelte Datenmenge zuzugreifen? Über die Twitter API <sup>3</sup> lässt sich eine große Menge an Tweets exportieren. Doch um die darin enthaltenen Informationen nutzen zu können, müssen diese erst noch herausgearbeitet werden. Dafür kann ein maschinelles Labeling-Verfahren eingesetzt werden, also ein Programm, welches automatisiert Tweet bewerten kann. An dieser Idee wird aktuell in der Arbeitsgruppe Medieninformatik an der Universität Osnabrück geforscht. Im Bewertungsprozess werden den Tweets sogenannte *Labels* zugewiesen, welche zum Beispiel Stimmungen wie positiv und negativ sein können. Des Weiteren können Tweets in Kategorien eingeordnet werden. Zum maschinellen Labeln eignen sich zum Beispiel ein Naiver Bayes-Klassifikator (Han et al. (2006)) oder eine Support Vector Maschine (Joachims (2002)).

Um die Qualität eines maschinellen Labeling-Verfahrens zu bewerten, werden jedoch Referenzdaten benötigt. Ein solches Verfahren ist dann besonders gut, wenn es eine Entscheidung möglichst so trifft, wie ein Mensch sie treffen würde. Das heißt, man benötigt Datensets bestehend aus von Menschen gelabelten Tweets, die mit den Ergebnissen des automatisierten Verfahrens verglichen werden können. Solche Datensets müssen bestimmte Qualitätsmerkmale aufweisen. Um statistisch signifikant zu sein, sollten möglichst nur Tweets enthalten sein, die von verschiedenen Menschen gleich gelabelt wurden.

Ein System zur Erstellung von solchen Datensets muss verschiedene Bedingungen erfüllen. Es muss einfach zu bedienen und aussagekräftig sein. Außerdem sollte es möglichst

---

<sup>1</sup>[http://www.bitkom.org/de/themen/36444\\_77780.aspx](http://www.bitkom.org/de/themen/36444_77780.aspx)

<sup>2</sup>[de.statista.com/statistik/daten/studie/232401/umfrage/monatlich-aktive-nutzer-von-twitter-weltweit-zeitreihe/](http://de.statista.com/statistik/daten/studie/232401/umfrage/monatlich-aktive-nutzer-von-twitter-weltweit-zeitreihe/)

<sup>3</sup><https://dev.twitter.com/>

variabel eingesetzt werden können. Besonders der Prozess des Zuweisens der Labels sollte intuitiv und benutzerfreundlich gestaltet sein. Für die Umsetzung eines solchen Systems bietet sich eine Webapplikation an, da sie plattformunabhängig ist und eine zentrale Verwaltung bereitstellt.

## 1.2 Zielsetzung und Aufbau der Arbeit

Die Zielsetzung dieser Arbeit ist es, mit möglichst geeigneten Technologien und Konzepten eine Webapplikation zu entwickeln, mit der kollaborativ Testdatensets zur Stimmungsanalyse von Tweets erstellt werden können. Diese muss den oben genannten Anforderungen gerecht werden. Daher gliedert sich die Arbeit in zwei grundlegende Teilbereiche. Im ersten Teil werden Technologien und Konzepte für die Applikation festgelegt. Im Anschluss wird dann mithilfe dieser die Webapplikation entwickelt.

Das Kapitel 2 erläutert zunächst den Einsatz des Webapplication Frameworks Ruby on Rails für die Umsetzung der Applikation. Im Anschluss wird die Methodik der Agilen Softwareentwicklung vorgestellt. Dabei werden die grundlegenden Paradigmen der Methodik auf die Kompatibilität mit dem gewählten Framework hin untersucht. Zum Schluss wird auf den Einsatz von Unit Tests im Entwicklungsprozess eingegangen.

Im Kapitel 3 wird dann die Umsetzung der Applikation mithilfe der gewählten Methoden und Technologien dokumentiert. Dabei wird vor allem Wert gelegt auf die Darstellung der Entscheidungen bezüglich der Usability der Applikation.

Die Arbeit wird durch eine Zusammenfassung und ein Fazit in Kapitel 4 abgeschlossen.

## 2 Technologien und Paradigmen

In diesem Kapitel sind die Technologien und Paradigmen aufgeführt, welche für dieses Softwareprojekt als sinnvoll erachtet und dementsprechend nach Möglichkeit angewendet wurden. Zu Beginn wird das verwendete Web Application Framework **Ruby on Rails** mit der zugrundeliegenden Programmiersprache **Ruby** und seinen Paradigmen vorgestellt. Anschließend wird die Entwicklungsmethode der **Agilen Softwareentwicklung** präsentiert. Beendet wird dieses Kapitel mit einer Ausführung über die Methodik des **Unit Testing**.

### 2.1 Ruby on Rails

Ruby on Rails ist ein in der Programmiersprache **Ruby** geschriebenes Webapplication Framework, welches seit 2004 der Öffentlichkeit bekannt ist. Es wurde von David Heinemeier Hansson entworfen, um Webentwicklern ihre Arbeit zu erleichtern. Laut Hansson ist das Entwickeln von Webapplikationen mit weit verbreiteten Technologien wie Java, PHP und .NET zu mühselig. Um diese Problematik zu lösen, hat Hansson Rails entwickelt, mit dessen Hilfe die Webentwicklung viel einfacher sei (vgl. Ruby et al. (2013)). Die Kernphilosophie von Rails besteht aus den folgenden Prinzipien: **Don't Repeat Yourself** (DRY), **Convention Over Configuration** (COC) und dem **Model-View-Controller Prinzip** (MVC). In diesem Kapitel wird zunächst die zugrundeliegende Programmiersprache **Ruby** vorgestellt. Danach werden die drei Prinzipien **DRY**, **COC** und **MVC** erläutert.

#### 2.1.1 Ruby

Ruby ist eine plattformunabhängige General Purpose Programmiersprache, die 1995 von dem Japaner Yukihiro Matsumoto entworfen wurde. Die Sprache ist dynamisch und objektorientiert und wurde konzipiert, um Programmierern dabei zu helfen, während des Entwickelns produktiver zu sein. Matsumoto möchte, dass sie ihre Arbeit genießen können und dabei glücklich sind, da er die Auffassung vertritt, dass dies die Produktivität und Qualität der Arbeit fördere. Matsumoto habe Ruby so entwickelt, dass die Syntax möglichst gut lesbar und schreibbar sei (vgl. Thomas et al. (2013)). Der Entwickler beschreibt seine Sprache mit den Worten:

Ruby is simple in appearance, but is very complex inside, just like our human body<sup>1</sup>.

---

<sup>1</sup>Yukihiro Matsumoto in der Ruby-Talk-Mailingliste am 12.Mai 2000

Die Sprache hat einen Garbage Collector, Entwickler müssen also nicht auf Speicher-  
verwaltung achten. Ruby lässt sich nahezu auf jeder Plattform nutzen. Entsprechende  
Literatur (Thomas et al. (2013)) und Tutorials<sup>2</sup> können zur ausführlichen Einführung in  
die Programmiersprache Ruby genutzt werden. Im Folgenden sollen jedoch die wichtigs-  
ten Besonderheiten der Sprache zum besseren Codeverständnis aufgeführt werden.

In Ruby wird das Ende einer Anweisung mit einem Zeilenumbruch markiert. Semikolons  
sind in dieser Sprache im Gegensatz zu bekannten Sprachen wie C++ und Java nicht  
nötig, können aber eingesetzt werden. Ist ein Ausdruck eindeutig, können Klammern  
vernachlässigt werden. Dies gilt sowohl für Definitionen von Methoden als auch für  
deren Aufruf. Der Rückgabewert einer Methode wird durch den zuletzt berechneten  
Wert bestimmt. Wie simpel Ruby ist, wird klar, wenn man den Aufbau des typischen  
"Hallo, Welt!" Programms in dieser Sprache betrachtet (Listing 1). Es besteht nur aus  
der Zeile, die die Ausgabe definiert.

```
1 puts "Hallo, Welt!"
```

**Listing 1:** "Hallo, Welt!" Programm in Ruby

Normalerweise ist eine Variable in Ruby lokal, ein @ vor dieser macht sie jedoch zur  
Instanzvariable. In Ruby werden statt for-Schleifen häufig sogenannte Blöcke eingesetzt.  
Ein Beispiel dafür ist in Listing 2 aufgeführt. Die Zeile 3 geht jeden Namen in `names`  
durch, schreibt ihn in die Variable `name` und führt damit den Code innerhalb des Blockes  
aus. Mit dem Schlüsselwort `end` wird der Block beendet. Dieses Schlüsselwort wird auch  
verwendet, um beispielsweise if-Anweisungen zu beenden. Mit `#{}` können in Ruby  
Variablen in Strings eingebunden werden.

```
1 names = ["Susi", "Willi", "Thorben", "Sören"]
2
3 names.each do |name|
4     puts "Goodnight, #{name}!"
5 end
6
7 # Ausgabe:
8 #
9 # Goodnight, Susi!
10 # Goodnight, Willi!
11 # Goodnight, Thorben!
12 # Goodnight, Sören!
```

**Listing 2:** Ein sogenannter Block in Ruby

In Ruby wird Duck Typing eingesetzt, sodass der Typ eines Objektes also nicht abhängig  
von der Klasse, sondern von den Methoden ist. Es existieren außerdem keine primitiven  
Datentypen in Ruby. Variablen müssen in dieser Sprache nicht deklariert werden.

---

<sup>2</sup><https://www.ruby-lang.org/de/documentation/quickstart/>

Die mit am häufigsten in Ruby verwendete Datenstruktur ist der Hash. Ein Hash ist eine Sammlung von Schlüsseln und Werten. Wie bei einem Array mit Integern als Indizes auf bestimmte Werte zugegriffen werden kann, kann bei einem Hash der Schlüssel als Index genutzt werden. Schlüssel und Werte können von beliebigen Objekt-Typen sein.

### 2.1.2 Don't Repeat Yourself

Bei dem Softwaredesign-Paradigma *Don't Repeat Yourself* dreht sich alles darum, jede mögliche Form von Redundanz zu vermeiden (Hunt and Thomas (1999), Kapitel 2). Ein Beispiel für ungewollte Redundanz wären Code-Duplizierungen. Die Problematik von Redundanzen besteht darin, dass Änderungen an redundanten Informationen aufwendig sind oder zu Inkonsistenz führen. Vor allem in Rails ist dies ein Problem, da Rails Agilität in der Entwicklung vorsieht, diese jedoch durch Redundanz nur erschwert und eingeschränkt möglich ist.

Listing 3 zeigt beispielhaft den Ausschnitt eines Controllers zur Verwaltung eines Datensets, welcher nicht DRY konform ist. Sowohl in der Controller-Action `show` als auch in der Action `edit` wird der lokalen Variable `description` ein String zugewiesen, welcher aus einer Verkettung aus dem String "Name: " und dem Namen des aktuellen Datensets `@dataset` besteht. Dieser Code ist also ein Duplikat.

```
1  class DatasetsController < ApplicationController
2
3  def show
4    @dataset = getDataset() # liefert aktuelles Datenset
5    @description = "Name: " + @dataset.name
6    # weiterer Code, Verwendung von @description
7  end
8
9  def edit
10   @dataset = getDataset() # liefert aktuelles Datenset
11   @description = "Name: " + @dataset.name
12   # weiterer Code, Verwendung von @description
13  end
14 end
```

**Listing 3:** Ausschnitt eines nicht DRY-konformen Datenset-Controllers.

Die Problematik dieser Tatsache zeigt folgendes Beispiel: Angenommen im weiteren Verlauf der Entwicklung würde ein Datenset ein weiteres Attribut `instruction` bekommen. Die Beschreibung des Datensets `description` solle nun angepasst werden, sodass alle Attribute des Datensets aufgeführt wären. Also müsste es um die Verkettung des Strings "Instruction: " mit der Anleitung `@dataset.instruction` ergänzt werden. In diesem Code-Beispiel müsste diese Änderung dann sowohl in Zeile 5 in der Controller-Action `show`, als auch in Zeile 8 in der Controller-Action `edit` vorgenommen werden. Dies macht den Aufwand der Änderung größer als notwendig und bringt das Risiko mit sich, dass bei der Änderung ein Fehler gemacht wird. Ein solcher Fehler

könnte zum Beispiel sein, dass in Zeile 11 in der Action `edit` ein Leerzeichen zwischen dem Wort `Instruction:` und der Anleitung des Datensets vergessen wird. Dies hätte zur Folge, dass die Variable `description` in inkonsistentem Zustand ist.

Eine verbesserte Version dieses Datenset-Controllers ist in Listing 4 dargestellt. Hier wurde die Verkettung des Strings für die Beschreibung des Datensets in eine private Methode `get_description` ausgelagert. Diese Methode wird jetzt jeweils in Zeile 4 in der `show`-Action und in Zeile 9 in der `edit`-Action aufgerufen und der Rückgabewert der `get_description` Methode wird der Variable `description` zugewiesen. Wenn in diesem Fall Datensets um das Attribut `instruction` erweitert würden, müsste lediglich der Code der Methode `get_description` angepasst werden.

```
1 class DatasetsController < ApplicationController
2   def show
3     @dataset = getDataset() # liefert aktuelles Dataset
4     description = get_description(@dataset)
5   end
6   def edit
7     @dataset = getDataset() # liefert aktuelles Dataset
8     description = get_description(@dataset)
9   end
10  private
11    def get_description(dataset)
12      return "Name: " + dataset.name
13    end
14  end
```

**Listing 4:** DRY-konforme Version des Controllers aus Listing 3

Dieses Beispiel zur Verdeutlichung des DRY-Paradigmas ist ein allgemeines Beispiel. Eine typische Anwendung des Paradigmas in Rails sind die `ActiveRecord` Models. Die Definition eines Models in Rails kann nur aus der Definition der Klasse an sich bestehen. Welche Attribute das Model besitzt, hängt davon ab, welche Attribute in der Zugehörigen Datenbanktabelle stehen.

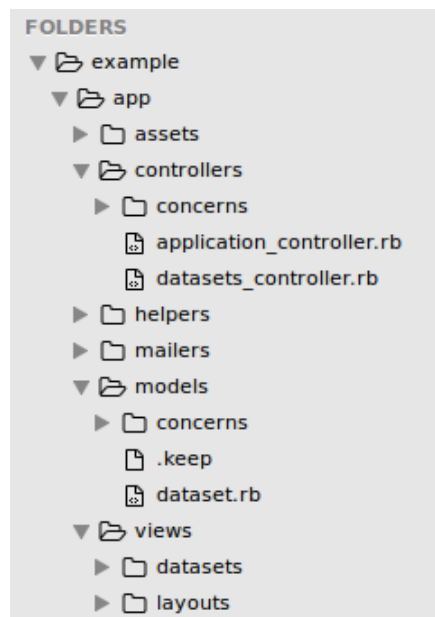
```
1 class User < ActiveRecord::Base
2   end
3   # Erzeugen eines Users
4   user = User.create(mail: "mbeutel@uos.de")
5   # Abrufen des Attributs
6   puts "Die Email-Adresse lautet: " + user.mail
7   # Ausgabe: Die Email-Adresse lautet mbeutel@uos.de
```

**Listing 5:** User Model in Rails

Eine solche „leere“ Klasse ist in Listing 5 zu sehen, in der das Model `User` definiert wird. In dieser Klasse werden keine Attribute definiert. Wurde die Datenbank jedoch so angelegt, dass ein Benutzer eine Email-Adresse besitzt, kann nach Anlegen eines Users auf dieses Attribut zugegriffen werden (Zeile 8), obwohl es in der Klasse nicht definiert wurde. Dies wird in Rails von `ActiveRecord::Base` realisiert, wovon jedes Model erbt.

### 2.1.3 Convention over Configuration

Das Design-Prinzip *Convention over Configuration* verfolgt das Ziel, auf möglichst geschickte Art und Weise viele Konventionen vorzugeben, damit der Entwickler während des Designprozesses nur noch eine begrenzte Anzahl an Entscheidungen treffen muss (Chen (2006)). Es soll die Komplexität der Konfiguration eines Projektes reduzieren. Ein typisches Rails-Beispiel für *Convention over Configuration* sind die Namensgebung und die Ordnerstruktur.



**Figure 2.1:** Der Konvention entsprechende Ordnerstruktur

In der Abbildung 2.1 ist ein Teil der Ordnerstruktur einer Rails-Applikation namens `example` zu sehen. In dem Ordner `example` befindet sich der Ordner `app`. Dieser Ordner enthält alle Applikationsdateien. Im Ordner `app/assets` befinden sich die Javascript-Dateien und die Stylesheets. Die Ordner `app/controllers`, `app/models` und `app/views` enthalten ihrem Namen entsprechend jeweils die Controller, die Models und die View-Dateien der Applikation.

Dieses Beispiel beinhaltet jeweils einen Controller, ein Model und eine View zur Darstellung eines Datensets. In Rails ist es Konvention Klassen auf Englisch und im CamelCase, die zugehörigen Dateien mit den Klassendefinitionen im snake\_case zu benennen. CamelCase bezeichnet die Schreibweise von zusammengesetzten Worten ohne Zwischenraum, aber einem Großbuchstaben am Anfang jedes Wortes. Bei der snake\_case

Schreibweise werden Worte durch einen Unterstrich getrennt und ohne Großbuchstaben geschrieben. Die Modelklasse für Datensets heißt `Dataset` (Datenset) und ist in der Datei `dataset.rb` im `app/models` Ordner zu finden. Der Name des zugehörigen Controllers beinhaltet den Modelnamen im Plural, in diesem Beispiel heißt er also `DatasetsController` und liegt in `app/controllers/datasets_controller.rb`. Die View Templates für Datensets befinden sich unter `app/views/datasets`. Der Name eines View Files ergibt sich durch den Namen der zugehörigen Controller-Action, also zB. `app/views/datasets/edit.html`, benannt nach der `DatasetsController`-Action `edit`.

### 2.1.4 Model-View-Controller Prinzip

Das Model-View-Controller Prinzip ist ein Softwarearchitekturmodell, welches die zu entwickelnde Applikation in drei Bestandteile gliedert: das Model, die View und den Controller (Fowler). Im Model sind die Logik und die Regeln der Applikation enthalten. Außerdem kommuniziert das Model mit der Datenbasis. Die View ist dafür zuständig, die Modeldaten als Output darzustellen. Dieser Output ist nicht eindeutig, da dieselben Daten auf verschiedenste Weisen dargestellt werden können, zum Beispiel abhängig davon, wer die Daten sieht. Außerdem stellt die View Elemente bereit, mit denen der Nutzer interagieren kann. Im Controller geschieht die Steuerung der Applikation. Er nimmt Interaktionen des Benutzers entgegen und passt entsprechend View oder Model an. Die Unterteilung in die drei Komponenten hat den Zweck, möglichst voneinander unabhängige Teile zu schaffen. Dies erleichtert das Testen und erhöht die Wartbarkeit. Zudem erreicht man eine starke Modularität, wodurch es möglich ist, einzelne Teile im Nachhinein in anderen Projekten zu verwenden oder im eigentlichen Projekt auszutauschen.

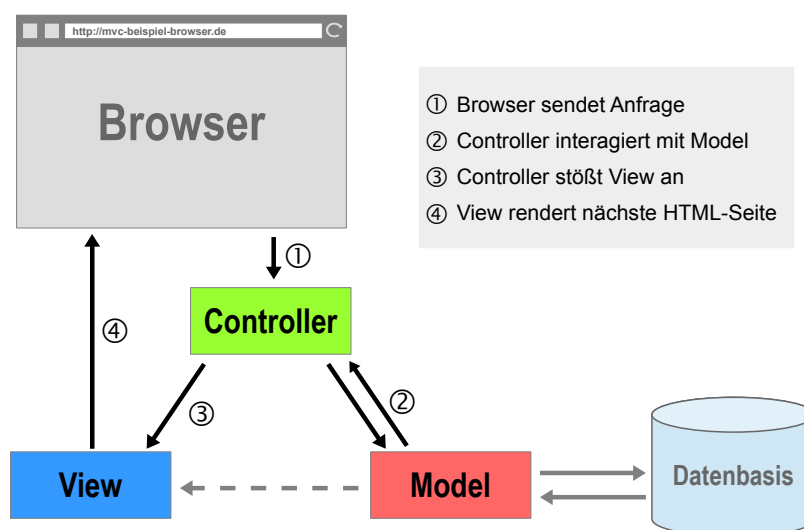


Figure 2.2: Model-View-Controller Architektur



Die Abbildung 2.2 zeigt den typischen Verlauf einer Anfrage eines Browsers in einer Model-View-Controller Architektur. Die Anfrage wird vom Browser an den Controller gestellt. Dieser interagiert mit dem Model. Das Model tauscht, falls nötig, Daten mit der Datenbasis aus. Zusätzlich stellt es der View Daten bereit. Nachdem der Controller fertig ist, stößt er die View an, welche den HTML-Code für die nächste Seite im Browser generiert.

## Model-View-Controller Prinzip in Rails

In Rails gehört die Model-View-Controller Architektur zu den grundlegenden Prinzipien. Schon eine neu generierte, leere Applikation enthält die Ordner `app/controllers`, `app/models` und `app/views`. In diesen Ordnern liegen dann pro Klasse der Controller, das Model und die View-Templates. Die Rails Konventionen geben also vor, dass Model, View und Controller als funktionale Einzelteile gebaut werden, die bei Ausführung der Applikation durch die intelligente Nutzung von Defaults zu einem funktionierenden Ganzen zusammengesetzt werden. Es ist also keine externe Konfiguration nötig, um die drei Einzelteile miteinander zu vereinen.

Eine Rails Applikation handhabt eine eingehende Anfrage eines Browsers, indem diese zuerst zu einem Router geleitet wird. Dieser entscheidet wohin die Anfrage weitergeleitet wird. Die Anfrage besteht aus einem Pfad und einer Methode. Im Idealfall gibt es eine Controller-Methode, die im Rails Jargon als Action bezeichnet wird, welche von der Anfrage gefordert wird. Der Aufruf dieser Controller-Action kann dann auf mitgelieferte Daten zugreifen, andere Actions aufrufen, mit dem Model interagieren und falls nötig, der View Daten bereitstellen. Diese rendert dann die nächste HTML-Seite im Browser.

### 2.1.5 MVC-Beispiel aus der Applikation

Um die Anwendung des Model-View-Controller Prinzips in dieser Applikation zu verdeutlichen, wird im Folgenden ein Beispiel eingeführt, welches das Ändern der Email-Adresse eines Benutzers darstellt. Im Anschluss werden beispielhaft Model, View und Controller für die Klasse User (Benutzer) vorgestellt.

Bei Aufruf der Seite `https://tweabler.informatik.uos.de/users/3/edit_mail` im Browser wird die Kombination aus Methode `GET` und Pfad `users/3/edit_mail` an den Router weitergeleitet. In der Routing-Komponente wird nun bestimmt, dass die Methode `edit_mail()` im `users_controller` aufgerufen werden soll.

Der Aufruf der Methode `edit_mail()` liest aus dem Pfad heraus, dass die Methode für den User mit der `id = 3` aufgerufen wurde. In der Methode wird nun das Model `User` benachrichtigt, es solle die Daten des Users mit der entsprechenden ID laden. Dann unterrichtet der Controller die View, welche das HTML-Formular für die `edit_mail()` Methode mithilfe der vom Controller bereitgestellten Daten rendert. Das gerenderte Formular erscheint im Browser und ist bereit dafür, die Benutzereingaben entgegen zu nehmen.

## Model

In Rails wird im Allgemeinen mit relationalen Datenbanken gearbeitet. Rails Applikationen basieren jedoch auf Objektorientierung. Daher ist Object-Relational Mapping (ORM) nötig, das Abbilden von den Datenbanktabellen auf Objekte. Eine Tabelle zu einer Rails-Klasse ist mit dem pluralisierten Namen im snake\_case benannt. Für die Klasse `User` ist also die Datenbanktabelle `users` vorgesehen. Zeilen der Tabelle entsprechen Objekten der Klasse, Spalten entsprechen den Attributen des Objekts.

In Rails-Klassen gibt es verschiedene Methoden auf Klassenlevel, die auf dem Datenbanklevel Operationen ausführen, um die Abbildung von Tabelle auf Objekt vorzunehmen. Es können Objekte erstellt werden, die den Zeilen der Tabelle entsprechen. Auf diesen Objekten können wiederum Operationen ausgeführt werden, die auf dieser Zeile agieren.

```
1 user = User.find(1) #Returns User with ID = 1
2 puts user.name     #Returns the Name of User with ID = 1
3
4 users = User.all   #Returns all Users in a Collection
5 users.each do |user| #Iterates over every User in users
6   puts user.name   #Returns the Name of current user
7 end
```

**Listing 6:** ORM in Rails

In Listing 6 ist ein Beispiel aufgeführt. In Zeile 1 wird auf die Klassenmethoden von `User` zugegriffen, um die Ausprägung der Klasse zu finden, für welche `id = 1` gilt. Diese wird als Objekt vom Typ `User` zurückgegeben und der lokalen Variable `user` zugewiesen. Zeile 2 greift nun auf das Attribut `name` des Objekts zu und gibt es auf der Konsole aus. In Zeile 4 werden alle Ausprägungen der Klasse `User`, in Form einer Collection von Objekten, der lokalen Variable `users` zugewiesen. Über diese Collection kann dann, wie in den Zeilen 5-7 zu sehen, iteriert werden, um jeweils auf die Attribute der einzelnen Objekte zuzugreifen.

In Listing 7 ist das User Model dieser Applikation zu sehen. In Rails erben Model-Klassen von `ActiveRecord::Base`, welches die ORM-Funktionalitäten zur Verfügung stellt. In dem Model werden zwei grundlegende Schritte ausgeführt: die Validierung und das Festlegen der Abhängigkeiten.

In den Zeilen 4-6 werden die Validierungen vorgenommen. Zeile 4 legt fest, dass ein User immer ein Passwort haben muss. Zeile 5 definiert, dass ein User immer eine einzigartige Email-Adresse braucht. In Zeile 6 wird das Format dieser Email-Adresse mithilfe eines regulären Ausdrucks überprüft. Diese Validierungen werden immer ausgeführt, wenn die Daten des Models verändert werden. Fällt die Änderung jedoch durch eine der Validierungen, wird sie nicht ausgeführt und eine Fehlermeldung wird zurückgegeben. Durch die Validierungen kann also festgelegt werden, wie eine gültige Ausprägung dieses Models auszusehen hat.

```
1  # Model for User
2  class User < ActiveRecord::Base
3    # only valid with unique name, password and mail-adress
4    validates :password, presence: true
5    validates :mail, presence: true, uniqueness: true
6    validates :mail, format: {with: /\S*\S*\[\.\]\w*/}
7    # Dependencies
8    has_many :roles, dependent: :destroy
9    has_many :datasets, through: :roles
10   has_and_belongs_to_many :entries
11 end
```

**Listing 7:** User Model

In den Zeilen 8-10 sind die Abhängigkeiten dieses Models geregelt. Zeile 8 legt fest, dass ein User viele Ausprägungen der Model-Klasse `Role` hat und dass diese Ausprägungen gelöscht werden sollen, falls der zugehörige User gelöscht wird. In Zeile 9 ist eine Abhängigkeit zur Model-Klasse `Dataset` über ein drittes Model `Role` definiert. Solche Abhängigkeiten über ein drittes Model werden häufig eingesetzt, um eine many-to-many Beziehung aufzubauen, bei der die Beziehung selbst Attribute hat. In Zeile 10 wird eine direkte many-to-many Beziehung mit dem Model `Entry` definiert.

## View

In Rails arbeiten View und Controller sehr eng zusammen: Der Controller sendet Daten an die View, die View sendet die generierten Ergebnisse an den Controller. Infolge dieser engen Beziehung ist der Rails-Support von View und Controller in der einzelnen Komponente `Action Pack` gebündelt.

Die simpelste Form von einer View ist statischer HTML-Code. Typischerweise wird aber eher HTML mit dynamischem Inhalt benötigt. In Rails wird dies mithilfe von View-Templates erreicht. Die häufigste Form von Template-Schema ist Embedded Ruby (ERB). In ERB ist es möglich, Ruby Code im HTML Code unterzubringen, um den dynamischen Inhalt zu generieren. Für die View-Templates in dieser Applikation wurde das ERB-Schema verwendet.

Ein beispielhaftes View-Template ist in Listing 8 zu sehen. Dieses Template generiert in Zeile 8 die Überschrift *Change Mail* und darunter ein Formular (Zeile 11-19). Das Formular besteht aus einem Eingabefeld für eine Email-Adresse (Zeile 14) und einem Button zum Absenden des Formulars (Zeile 17). Nach dem Drücken des Buttons wird eine Anfrage mit der Methode `PATCH` und dem Pfad `users/update_mail/` an den Router gesendet, welcher die Methode `edit_mail()` im `UserController` aufruft.

```
1 <!DOCTYPE html>
2 <html>
3   <head>
4     <title>Labeling</title>
5   </head>
6   <body>
7     <div class="page-header">
8       <h1>Change Mail</h1>
9     </div>
10
11     <%= form_for(@user,url: {action: "update_mail"}) do |f| %>
12       <div class="field">
13         <%= f.label "Mail" %><br />
14         <%= f.text_field :mail %>
15       </div>
16       <div class="action_container">
17         <%= f.submit %>
18       </div>
19     <% end %>
20   </body>
21 </html>
```

**Listing 8:** User View-Template im ERB-Schema

## Controller

In Listing 9 ist ein Ausschnitt des `UsersControllers` zu sehen. In Zeile 1 erbt der `UsersController` vom `ApplicationController`. Dieser `ApplicationController` enthält alle Methoden, die für jeden Controller der Applikation zugänglich sein sollen. Er bringt außerdem die Funktionalitäten zur Kommunikation von View und Controller mit. Ansonsten beinhaltet der Ausschnitt des Controllers nur die einzelne Methode `update_mail()`, welche durch das Abschicken des in Listing 8 vorgestellten Formulars aufgerufen wird.

Wenn eine Controller-Action durch das Absenden eines Formulars aufgerufen wird, steht dieser der Inhalt des Formulars im Hash `params` zur Verfügung. Um diese Parameter nicht unkontrolliert zu lassen, gibt es die private Methode `user_params()`, welche ausgewählte Parameter durch einen Sicherheitsmechanismus führt und danach im Hash `user_params` unterbringt. In `session[:user_id]` steht die `id` des aktuell eingeloggten Users. In Zeile 6 wird dieser aktuelle User geladen und in der Instanzvariable `@user` gespeichert. In den Zeilen 8-9 wird dann das Attribut `mail` des aktuellen Users mit der übergebenen `mail` aus `user_params` geupdated, die Methode `update_attributes` durchläuft dabei die Validierungen des Models. War das Aktualisieren des Attributs erfolgreich, wird die `show` Action des Users aufgerufen. Darin wird der aktualisierte User gerendert (Zeile 9). Andernfalls wird das Formular erneut gezeigt (Zeile 15).

```
1 class UsersController < ApplicationController
2   # called after submit of edit-mail form
3   def update_mail
4     if not session[:user_id].nil?
5       # current User
6       @user = User.find(session[:user_id])
7       # tries to update mail with passed parameter
8       if @user.update_attributes(:mail, user_params[:mail])
9         redirect_to @user
10      else
11        # if not successful
12        render "edit_mail"
13      end
14    else
15      redirect_to login_path
16    end
17  end
18 end
```

**Listing 9:** Ausschnitt des Users Controllers

### 2.1.6 Zusammenfassung

Insgesamt ist festzuhalten, dass Ruby on Rails für dieses Projekt sehr geeignet ist. Das Framework unterstützt die Agile Softwareentwicklung. Ruby Code ist außerdem sehr leicht zu lesen und die durch Rails vorgegebenen Konventionen und die Model-View-Controller Architektur vereinfachen den Designprozess. Für Außenstehende ist es entsprechend leichter, sich später in die Applikation einzuarbeiten. Außerdem wurden sowohl die Programmiersprache Ruby als auch das Webapplication Framework Ruby on Rails dafür entwickelt, dem Programmierer die Arbeit möglichst leicht und angenehm zu gestalten. Es kann sich also auf die produktive Umsetzung der Kundenwünsche konzentriert werden.

## 2.2 Agile Softwareentwicklung

Die Agile Softwareentwicklung ist eine Methodik zur Softwareentwicklung, bei der ein flexibler, schlank gehaltener Entwicklungsprozess im Vordergrund steht. Dieser Begriff wurde erstmals im *Agilen Manifest* (Alliance (2001)) vorgestellt. Dieses Manifest wurde im Jahr 2001 von 17 Entwicklern geschrieben, welche sich in Utah trafen, um leichtgewichtige Entwicklungsmethoden zu untersuchen.

### 2.2.1 Die Paradigmen

In der Agilen Softwareentwicklung gibt es vier Hauptparadigmen, die das Grundkonzept der Entwicklungsmethode bilden.

- **Menschen und Interaktionen** vor Prozessen und Werkzeugen
- **Funktionierende Software** vor umfassender Dokumentation
- **Zusammenarbeit mit Kunden** vor ursprünglichen Leistungsbeschreibungen
- **Eingehen auf Veränderungen** vor Festhalten an einem Plan

Zu betonen ist, dass die Punkte auf der rechten Seite nicht zu vernachlässigen sind. In der Agilen Softwareentwicklung wird auch auf diese Punkte Rücksicht genommen, nur stehen sie gegenüber den Punkten auf der linken Seite deutlich im Hintergrund (vgl. Chelimsky et al. (2010), S. 115).

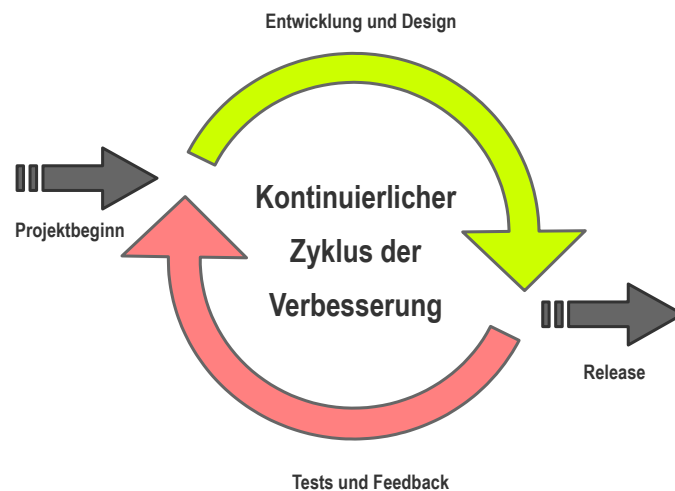
Das erste Paradigma stellt die im Projekt eingespannten Menschen und deren Interaktionen in den Vordergrund. Motivation ist ein entscheidender Punkt, denn motivierte Menschen sind produktiver. Das Projekt wird um die involvierten Menschen herum gebaut und ruht damit auf deren Schultern. Es sollte also Vertrauen herrschen. Kommunikation sollte möglichst ohne Umwege, also am besten in Form von direkten Treffen, geschehen. Im Projekt involvierte Teams sollten außerdem Selbstorganisation betreiben.

Das zweite Paradigma betont den Grad der Relevanz funktionierender Software in der Agilen Softwareentwicklung. Dieser Grundsatz soll bewirken, dass der Kunde möglichst schnell Ergebnisse in Form von bereits benutzbarer Software sieht. Es geht dabei darum, nicht alle möglichen Projektteile gleichzeitig zu beginnen und erst am Schluss gleichzeitig fertig zu stellen, sondern funktionale Einzelteile nacheinander zu entwickeln. Ein Programm, das schon ein gewisses Maß an Funktion zeigt, lässt sich dem Kunden besser vorführen und ist anschaulicher als Entwürfe oder Dokumente über den Projekthergang. Der Fortschritt des Projekts wird daran gemessen, wie groß der Anteil funktionierender Software ist. Es wird außerdem immer Wert darauf gelegt, gutes Design zu betreiben und technisch hochwertig zu arbeiten.

Der dritte Punkt der Paradigmen bezieht sich auf die Zusammenarbeit mit dem Kunden. In der agilen Softwareentwicklung ist bekannt, dass ein Kunde seine Ansprüche an das Ergebnis eines Softwareprojektes während des Entwicklungsprozesses häufig noch anpasst. Durch das zweite Paradigma kann der Kunde schon früh feststellen, ob der bisherige Stand seinen Ansprüchen genügt und weitere Ideen bilden. Es wird als wichtig angesehen, möglichst häufig mit dem Kunden Rücksprache zu halten und dessen Meinung in der Relevanz hoch einzustufen. Auch hier wird auf direkte Kommunikation gesetzt. Die enge Zusammenarbeit zwischen Entwickler und Kunde bewirkt, dass das Projekt in allen Phasen von den Wünschen des Kunden geprägt wird.

Der letzte Punkt bildet, vor allem im Zusammenhang mit den anderen Punkten, das Kernstück der Agilen Softwareentwicklung. Wie schon die Bezeichnung *agil* andeutet, spielt Flexibilität eine große Rolle in dieser Entwicklungstechnik. Es ist nicht nötig, zu Beginn des Projekts einen bis ins Detail ausgearbeiteten Plan zu erstellen. Es ist

ausreichend, sich auf Grundfunktionen und erste Ideen zu einigen, die häufigen Rücksprachen mit dem Kunden ergeben dann jeweils die nächsten möglichen Schritte. Die Abbildung 2.3 stellt den Zyklus dar, der sich durch das Abwechseln von *Entwicklungs- und Designphasen* und *Test- und Feedbackphasen* ergibt. Dieser Zyklus wird vom Start des Projektes bis hin zum Release der Software - und oft sogar darüber hinaus - fortgeführt.



**Figure 2.3:** Zyklus der Verbesserung in der Agilen Softwareentwicklung

Durch diese Paradigmen ergibt sich der Vorteil, dass am Ende des Projekts kein *Big Bang* eintritt. Ein *Big Bang* bezeichnet die Problematik eines am Ende stehenden Softwareprojektes, welches erst vollständig implementiert und danach in Betrieb genommen, getestet und dem Kunden vorgeführt wird. Dieser Zeitpunkt ist häufig kritisch, da Fehler nicht selten sind. Zudem ist der Kunde oft nicht zufrieden, weil er während des Projektes keine Chance auf Mitsprache hatte und eventuelle Änderungen jetzt sehr aufwändig sind. Bei der Agilen Softwareentwicklung tritt diese Problematik nicht auf, denn jede neue Funktionalität wird so bald wie möglich getestet und dem Kunden vorgeführt. Zudem sind die einzelnen Teile der Software in der Regel funktional unabhängig voneinander. Dies begünstigt auch spätere Änderungen einzelner Programmteile.

### 2.2.2 Anwendbarkeit auf diese Arbeit

Da das dieser Arbeit zugehörige Softwareprojekt nicht von einem Team von Entwicklern, sondern einer Einzelperson durchgeführt wird, steht die Frage offen, welche Aspekte der Agilen Softwareentwicklung für einen Einzelentwickler relevant bzw. umsetzbar sind. Das erste Paradigma ist für einen Einzelentwickler nicht so maßgebend wie die übrigen Paradigmen, da die Frage nach Vertrauen und Kommunikation im Team nicht auftaucht. Die anderen Paradigmen lassen sich jedoch direkt auch auf Einzelentwickler anwenden.

Folgende Punkte wurden also für dieses Projekt entsprechend der Paradigmen der Agilen Softwareentwicklung aufgenommen:

- Häufige Rücksprache mit dem Betreuer und Auftraggeber der Arbeit (Kunde)
- Umsetzen von unabhängigen Einzelfunktionalitäten
- Direktes Testen dieser Einzelfunktionalitäten
- Direktes Vorführen der Einzelfunktionalitäten gegenüber dem Kunden
- Direktes anwenden des Feedbacks des Kunden
- Flexibilität gegenüber (neuen) Anforderungen und Wünschen des Kunden

### 2.2.3 Agilität in Rails

Das für diese Arbeit verwendete Webapplication-Framework Rails unterstützt einige Aspekte der Agilen Softwareentwicklung schon von sich aus. In Rails ist es möglich, durch einen einzigen Konsolenbefehl einen mitgebrachten Server zu starten, auf dem dann im Browser der Wahl die Applikation unter `localhost:3000` aufgerufen werden kann. Änderungen im Code werden auch bei laufendem Server übernommen, sodass theoretisch nach einer Änderung ohne Neustart des Servers die veränderte Funktion der Applikation betrachtet und getestet werden kann. Dies unterstützt auch die Zusammenarbeit mit dem Kunden, da die Applikation diesem in jedem Zustand vorgeführt werden kann. Dieser sieht die Applikation schon im Browser, wo sie letztendlich für Anwender sichtbar sein wird. Er bekommt also schon einen guten Eindruck davon, wie die Applikation bei Benutzung funktioniert und wirkt. Dadurch, dass Rails auf den Paradigmen **COC** und **DRY** sowie dem Designpattern **MVC** beruht, sind Änderungen im Code wenig aufwändig und riskieren keine Inkonsistenz. Außerdem wurde im Rails-Framework erstmals ein „*integrated full-stack testing framework*“ zur Verfügung gestellt (Chelimsky et al. (2010), S. 277). Mithilfe dieses Frameworks wird es dem Entwickler unter anderem einfach gemacht, Unit Tests einzubauen.

## 2.3 Unit Testing

Da in der Agilen Softwareentwicklung das häufige Testen der Applikation eine große Rolle spielt, wurden in dieser Applikation sogenannte Unit Tests eingesetzt (Osherove (2014)). Unit Tests werden verwendet, um funktionale Einzelteile einer Applikation zu testen. Sie werden eher in frühen Testphasen angesetzt und testen die detailreichsten, grundlegendsten Komponenten der Software.

Ein Vorteil von Unit Tests ist, dass mit ihrer Hilfe Einheiten nach einer Änderung an der Applikation auf die gewollte Funktionalität überprüft werden können. Dies ist gerade in der Agilen Softwareentwicklung von Vorteil, wo zu jedem Zeitpunkt des Projekts Änderungen möglich sein sollten, ohne dabei die Grundfunktionalität des Programms zu verändern. Auch wenn eine Applikation später von einer anderen Person als den ursprünglichen Entwicklern angepasst werden soll, sind Unit Tests hilfreich, da die Person



damit testen kann, ob ihre Änderungen die Funktionalität des Programms beeinträchtigen, ohne sich in die gesamte Software einarbeiten zu müssen.

### 2.3.1 Unit Testing in Rails

In Rails ist es üblich, die Models mit Unit Tests zu versehen, denn diese entsprechen in Rails gewissermaßen den kleinsten Komponenten der Applikation. Das Rails Framework stellt schon beim Start des Projektes Unterstützung für Tests bereit. Im Ordner `test/models` wird für jedes Model automatisch ein leerer Unit Test erzeugt, der dann bei Bedarf vom Entwickler mit Inhalt gefüllt werden kann. Üblicherweise wird ein Unit Test mit verschiedenen Einzeltests gefüllt.

```
1 class UserTest < ActiveSupport::TestCase
2   # Test that no users with empty attributes are valid
3   test "user attributes must not be empty" do
4     user = User.new
5     assert user.invalid?
6     assert user.errors[:mail].any?
7     assert user.errors[:password].any?
8   end
9
10  #test that no users without unique mail are valid
11  test "user mail must be unique" do
12    user = users(:one)
13    user2 = User.new(mail: user.mail,
14                    password: user.password_digest,
15                    password_confirmation: user.password_digest)
16    assert user2.invalid?
17    assert user2.errors[:mail].any?
18  end
19
20  #test that no users without valid mail format are valid
21  test "mail format must be valid" do
22    user = User.new(mail: "hallo.de",
23                  password: "1",
24                  password_confirmation: "1")
25    assert user.invalid?
26    assert user.errors[:mail].any?
27  end
28 end
```

**Listing 10:** Unit Test für Model User

Wenn die Tests mit Inhalt gefüllt wurden, ist es möglich, in einem Terminal im Projektordner das Kommando `rake test:models` auszuführen. Dieses Kommando lässt jeden

Unit Test durchlaufen und gibt auf der Kommandozeile Auskunft über den Erfolg oder Misserfolg der einzelnen Tests in der Unit Test Klasse.

Sind alle Tests erfolgreich durchlaufen worden, ist dies zwar keine Garantie für ein fehlerloses Model, wurden jedoch die Tests sinnvoll aufgebaut und die Testdaten in den Fixtures sinnvoll gewählt, erhöht dies die Wahrscheinlichkeit einer geringen Fehlerrate drastisch und kann den weiteren Testverlauf deutlich verkürzen.

Durch die Simplizität von Unit Tests und die Unterstützung durch das Rails Framework sind die Kosten für Unit Tests eher gering, wohingegen der Nutzen groß ist. Daher wurde dieser Testmechanismus im weiteren Verlauf des Projektes zur Unterstützung des Konzepts der Agilen Softwareentwicklung eingesetzt.

Ein Beispiel für einen solchen Unit Test ist in Listing 10 aufgeführt. Dieser Unit Test testet das Model `User` aus Listing 7, welches in Abschnitt 2.1.5 vorgestellt wurde. Im Prinzip werden in dem Unit Test jetzt die Validierungen aus dem Model überprüft. Für jede Validierung im `User`-Model gibt es eine Testmethode im zugehörigen Unit Test.

## Test Fixtures

Eine `Fixture` ist eine Umgebung, in der sich Tests ausführen lassen. In Rails ist es möglich, solche `Fixtures` im YAML-Format anzulegen. Es liegt je Model eine `Fixture`-Datei im Ordner `test/fixtures`. Für das Model `User` ist die `Fixture`-Datei in Listing 11 aufgeführt.

```
1 one:
2   mail: first.example@uos.de
3   password_digest: <%= BCrypt::Password.create('secret') %>
4
5 two:
6   mail: second@example@uos.de
7   password_digest: <%= BCrypt::Password.create('secret') %>
```

**Listing 11:** Test Fixture für User-Model im YAML-Format

Es werden die zwei Datensätze `one` and `two` erstellt, in denen jeweils die User-Attribute `mail` und `password` festgelegt werden. In Listing 10 ist in Zeile 12 zu sehen, wie der Datensatz `one` verwendet wird. Der Aufruf der Methode `users(:one)` liefert ein User-Objekt mit den in der `Fixture` definierten Attributen zurück und weist es der lokalen Variable `user` zu.

## Unit Test des User-Models

Um den Aufbau eines Unit Tests näher zu beleuchten, wird hier derjenige des User-Models (Listing 10) im Detail erklärt. Dieser Unit Test besteht aus den drei Tests "user attributes must not be empty", "user mail must be unique" und zuletzt

"mail format must be valid". Die Namen der Tests ergeben sich aus der zu testenden Funktionalität.

Der erste Test überprüft, ob die Validierungen im User-Model wirklich sicherstellen, dass ein User nicht mit leeren Attributen erstellt werden kann. Dafür wird in Zeile 4 ein leerer User erstellt. In den Zeilen 5-7 wird sichergestellt, dass der User nicht gültig ist und dass Fehlermeldungen für die beiden leeren Parameter geworfen wurden.

Im zweiten Test wird kontrolliert, dass kein User mit einer schon für einen anderen User verwendeten Email-Adresse erstellt werden kann. Dafür wird ein User mithilfe der User-Fixture `one` erstellt. Danach wird ein weiterer User mit den gleichen Attributen wie der erste User erstellt. In Zeile 16 wird der zweite User auf Ungültigkeit überprüft, Zeile 17 testet auf eine vorhandene Fehlermeldung für das Attribut `mail`.

Zuletzt wird im dritten Test noch der Ausschluss von Emails mit ungültigem Format garantiert. Es wird ein User mit gültigem Passwort, aber ungültiger Email-Adresse erstellt. Auch bei diesem User wird die Ungültigkeit und die Fehlermeldung des Attributes `mail` revidiert.

## 2.4 Zusammenfassung

Durch das Webapplication Framework *Ruby on Rails* und die damit einhergehenden Paradigmen, durch die Methodik der *Agilen Softwareentwicklung* und durch das *Unit Testing* werden die Grundzüge der Applikation bereits definiert. Wie auf diesen Grundzügen aufgebaut und daraus die Applikation entwickelt wurde, wird im folgenden Kapitel ausführlich erläutert.



## 3 Die Applikation

In diesem Kapitel wird die Umsetzung der Applikation vorgestellt. Dafür werden zuerst die Anforderungen an die Applikation veranschaulicht. Im Anschluss werden sowohl Front- als auch Back-End der Applikation erläutert. Danach wird genauer auf die Umsetzung von Datensets in der Applikation eingegangen. Im darauf folgenden Abschnitt wird die Seite der Webapplikation vorgestellt, auf der der Labelingprozess stattfindet. Am Schluss werden noch der Einsatz von AJAX und die Testphase der Applikation aufgegriffen.

### 3.1 Anforderungen

In der Anfangsphase wurden die ersten Anforderungen an die Applikation in einem Gespräch mit dem Auftraggeber festgelegt, um die grundlegende gewünschte Funktionalität der Applikation zu bestimmen. Im Anschluss wurden diese ausformuliert und zu einem Gesamtbild zusammengesetzt, welches im Folgenden vorgestellt wird.

**User und Rollen:** Es soll in der Applikation User in drei verschiedenen möglichen Rollen geben: **Admin**, **Datenset-Admin** und **Labeler**. Einen Admin gibt es nur einmal in der gesamten Applikation, der User mit dieser Rolle ist mit allen Rechten ausgestattet und ist der Einzige, der neue Datensets anlegen und diese mit Tweets füllen kann. Ein Admin kann zu einem Datenset Datenset-Admins hinzufügen. Diese können das Datenset verwalten und dem Datenset Labeler hinzufügen. Labeler haben die geringsten Rechte. Sie können nur labeln, aber keine sonstigen Änderungen an Datensets vornehmen. Der Admin besitzt seine Rolle immer für alle Datensets. Die anderen Rollen werden jeweils nur für ein bestimmtes Datenset vergeben. Das bedeutet, dass ein und derselbe User unterschiedliche Rollen für unterschiedliche Datensets einnehmen kann.

**Crosslabelzahl:** Die Crosslabelzahl bezeichnet die Anzahl der Labels, mit der ein Tweet im Datenset versehen sein muss, um bereit für den Export zu sein. Sie sagt allerdings nichts darüber aus, ob der Tweet auch entsprechend oft mit dem gleichen Label versehen wurde.

**Relabelanteil:** Der Relabelanteil hat Einfluss darauf, welche Tweets für den Labelingprozess ausgewählt werden. Er ist in Prozent angegeben, ist der Anteil hoch, werden viele Tweets gelabelt. Das bedeutet, es werden viele Tweets für das Labeln ausgewählt, die schon gelabelt wurden. Ist der Anteil gering, werden mehr Tweets gewählt, die noch nicht mit Labels versehen wurden.

**Labeling:** User können einem Tweet ein Label und eine (optionale) Kategorie zuteilen. Jeder User kann einen Tweet nur einmalig labeln. Ein Tweet wird höchstens der Crosslabelzahl entsprechend oft gelabelt. Der nächste zu labelnde Tweet wird abhängig vom Relabelanteil entweder aus den bereits gelabelten Tweets oder aus noch nicht gelabelten Tweets ausgewählt. Labels und Kategorien können als Garbage-Label bzw. -Kategorie eingestellt werden. Tweets, welche mit einem solchen Label oder einer solchen Kategorie bewertet werden, sind vom weiteren Labeling-Prozess ausgeschlossen.

**Datensets:** Eine Menge von Tweets bildet in der Applikation ein Datenset. Für ein Datenset soll einstellbar sein, wie hoch die Crosslabelzahl und der Relabelanteil sind. Außerdem können dem Datenset mögliche Labels und Kategorien zugewiesen werden, mit welchen enthaltene Tweets gelabelt werden können.

**Usability:** Die gesamte Applikation sollte möglichst nutzerfreundlich gestaltet werden. Die Website sollte übersichtlich gestaltet sein. Admin und Datensetadmin sollten schnell einen Überblick über ein Datenset erlangen können. Dafür sollen Statistiken bzw. Diagramme für das Datenset einsehbar sein. Für die Labeler sollte der Labelingprozess möglichst intuitiv und simpel gestaltet sein.

**Im- und Export:** Der Admin soll Dateien mit noch nicht gelabelten Tweets importieren können. Sowohl der Admin als auch Datenset-Admins sollen die Daten im Datenset exportieren können. Der Export soll möglichst variabel gestaltet werden, damit der Nutzer das exportierte Datenset ganz nach seinen Wünschen gestalten kann.

**Variabilität** Die gesamte Applikation soll möglichst variabel gestaltet werden, damit sie später in verschiedenen Bereichen und von verschiedenen Anwendern benutzt werden kann. Admins und Datensetadmins sollten viele Einstellungsmöglichkeiten bezüglich der Ausprägung der Datensets haben. Dies beinhaltet, dass mögliche Labels und Kategorien je Datenset einzeln einstellbar sind. Außerdem sollen Labels selbst mit einer Farbe, passend zum Labeltext, versehen werden können. Auch die Einstellbarkeit von Crosslabelzahl und Relabelanteil sowie die Variabilität des Exports unterstützen die variable Einsetzbarkeit der Applikation.

## 3.2 Front-End

In diesem Abschnitt wird das Front-End der Applikation vorgestellt. Dafür wird zunächst der Pageflow vermittelt. Im Anschluss wird die Benutzeroberflächengestaltung begründet und anhand einiger Beispiele anschaulich gemacht.

### 3.2.1 Pageflow

Da der Pageflow der Applikation relativ umfangreich ist und sich für die drei möglichen Rollen der User unterscheidet, wird dieser für die Rollen einzeln dargestellt, wobei für die mit mehr Rechten ausgestattete Rolle jeweils auch der gesamte Pageflow der Rollen mit weniger Rechten zur Verfügung steht.

#### Labeler Pageflow

Die Rolle mit den geringsten Rechten ist der **Labeler**. Der Pageflow der Labeler ist in Abbildung 3.1 zu sehen. Die Seite, auf der man startet, wenn die Applikation aufgerufen wird, ist die **Login Page**. Dort ist ein Formular zur Anmeldung mit Email-Adresse und Passwort für den User zu sehen. Drückt man auf den *Submit* Button, gelangt man bei gültigen Anmeldedaten zur **Welcome Page**, bei ungültigen Daten wird man erneut zur **Login Page** geleitet.

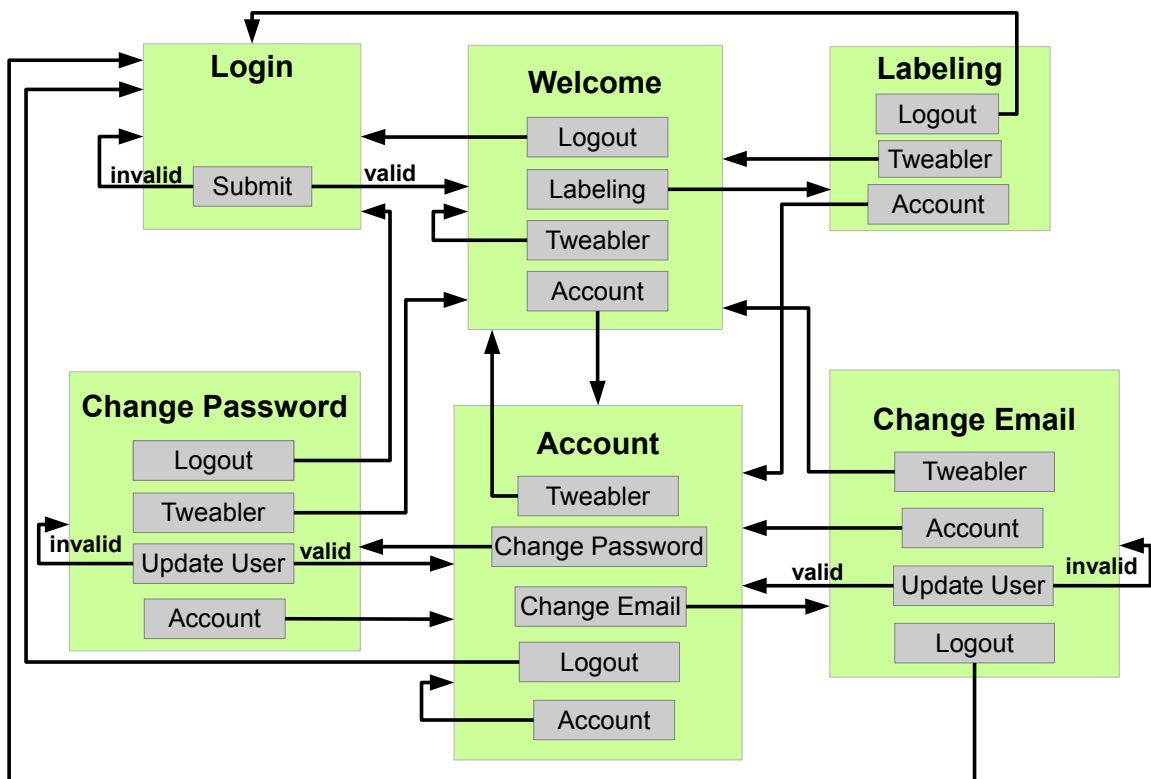


Figure 3.1: Pageflow der Applikation für Benutzer in der Rolle Labeler

Ist ein Benutzer angemeldet, erscheinen die Buttons *Account* und *Logout* auf der rechten Seite der Navigationsleiste an der oberen Kante der Website. Auf der linken Seite der Navigationsleiste gibt es den Button *Tweabler* (Name der Applikation). Wählt der Benutzer *Logout*, wird er wieder zur **Login Page** geleitet. Der Button *Tweabler* führt von jeder Seite zurück zur **Welcome Page**.

Der Button *Account* leitet den Benutzer zur **Account Page** weiter. Hier werden die Email-Adresse und die Rollen des Benutzers angezeigt. Außerdem kann man von hier aus über *Change Password* zur **Change Password Page** gelangen. Dort wird ein Formular zum ändern des Passworts gerendert. Drückt man dort auf den *Update User* Button, wird man bei gültiger Eingabe zurück zur **Account Page**, bei ungültiger Eingabe erneut zur **Change Password Page** geleitet. Ähnlich ist der Pageflow bei dem Button *Change Email* auf der **Account Page**. Er leitet zur **Change Email Page** weiter, wo ein Formular zur Anpassung der Email-Adresse gerendert wird. Bei gültiger Eingabe führt der Button *Update User* zur **Account Page**, bei ungültiger wieder zur **Change Email Page**.

Auf der **Welcome Page** sind für den Benutzer alle für ihn zulässigen Datensets gelistet. Zu jedem dieser Datensets gibt es einen Button *Labeling*, der den Benutzer zur **Label Page** des jeweiligen Datensets leitet.

### DatasetAdmin Pageflow

Ein Benutzer mit der Rolle **DatasetAdmin** hat mehr Rechte als ein Labeler, aber weniger Rechte als ein Admin. Daher gilt der Page Flow aus Abbildung 3.1 auch für Datasetadmins. In Abbildung 3.2 ist der Teil des DatasetAdmin Pageflows dargestellt, welcher über den der Labeler hinausgeht. Dabei wurden die Buttons der Navigationsleiste (vgl. 3.2.1) in der Darstellung der Übersichtlichkeit halber vernachlässigt, sie sind jedoch von jeder Seite aus nutzbar.

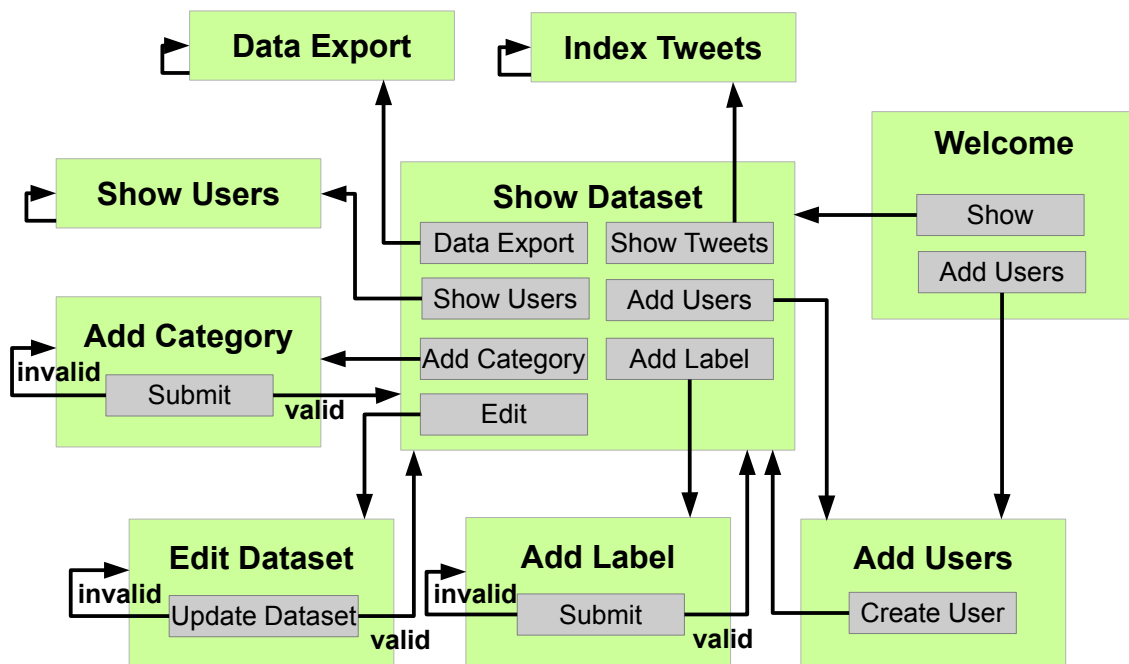


Figure 3.2: Pageflow der Applikation für Benutzer in der Rolle DatasetAdmin



Auch für Datenset Admins ist auf der **Welcome Page** jedes Datenset gelistet, für das er eine Rolle einnimmt. Ist diese Rolle **DatasetAdmin**, erscheinen zusätzlich zum *Labeling* Button (vgl. 3.2.1) die Buttons *Show* und *Add Users* für dieses Datenset.

Der *Show* Button leitet den Benutzer weiter zur **Show Dataset Page**. Diese zeigt die Details eines Datensets, wie zum Beispiel die Beschreibung, die Crosslabelzahl, einstellbare Labels et cetera. Außerdem sind einige weitere Seiten über diese Seite erreichbar. Über die Buttons *Data Export*, *Show Users* und *Show Tweets* lassen sich jeweils die **Data Export Page**, **Show Users Page** und die **Index Tweets Page** erreichen. Ersterer ist für den Datenexport zuständig, die beiden anderen sind jeweils dafür da, die Benutzer bzw. Tweets des Datensets aufzulisten. Von allen drei Seiten gelangt man nur herunter, indem man den *Tweabler* Button nutzt (vgl. 3.2.1).

Die Buttons *Add Category* und *Add Label* auf der **Show Dataset Page** rufen die **Add Category Page** bzw. die **Add Label Page** auf. Auf diesen Seiten lassen sich dem Datenset mit einem Formular Kategorien bzw. Labels hinzufügen. Sind die Eingaben im Formular gültig, wird zurück zur **Show Dataset Page** geleitet, im anderen Fall rufen die Seiten sich selbst nochmal auf.

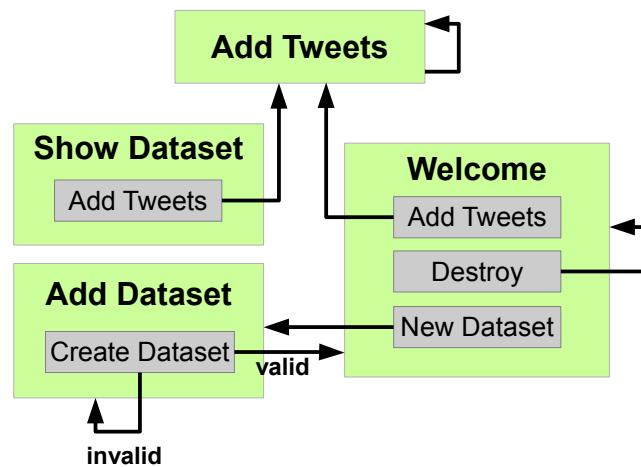
Mit dem *Edit Dataset* Button erreicht der Benutzer die **Edit Dataset Page**. Hier kann das Datenset mithilfe eines Formulars editiert werden. Es können der Name, die Beschreibung, die Crosslabelzahl, die Anleitung und die Relabelzahl verändert werden. Sind die Änderungen gültig, wird zurück zur **Show Dataset Page** geleitet, im anderen Fall wird auch hier das Formular erneut angezeigt.

Sowohl von der **Show Dataset Page** als auch von der **Welcome Page** lässt sich über den Button *Add Users* die **Add Users Page** aufrufen, auf der es möglich ist, dem Datenset mithilfe eines Formulars neue Benutzer hinzuzufügen. Drückt der Benutzer den *Create User* Button, wird er zurück zur **Show Dataset Page** geleitet.

## Admin Pageflow

Der Benutzer mit der Rolle **Admin** hat alle Rechte der Applikation, für ihn gelten sowohl Pageflow 3.1 als auch 3.2. Zusätzlich ist in Abbildung 3.3 der restliche Pageflow des Admins zu sehen.

Auf der **Welcome Page** werden dem Admin für jedes Datenset zusätzlich zu den schon vorgestellten Buttons noch *Add Tweets* und *Destroy* angezeigt. Der *Add Tweets* Button ist auch auf der **Show Dataset Page** vorhanden. Über *Add Tweets* gelangt der Admin zur **Add Tweets Page**, wo er mit einem Formular Tweets für das Datenset importieren kann. Mit *Destroy* kann ein Datenset gelöscht werden, der Button ruft wieder die **Welcome Page** auf. Zusätzlich gibt es auf der **Welcome Page** noch den *New Dataset* Button. Mit diesem gelangt man zur **Add Dataset Page**, wo mit einem Formular ein neues Datenset erstellt werden kann. Bei gültigen Eingaben wird zurück zur **Welcome Page** geleitet, bei ungültigen Eingaben wird das Formular erneut gezeigt.



**Figure 3.3:** Pageflow der Applikation für den Benutzer in der Rolle Admin

### 3.2.2 Benutzeroberfläche

Das grundsätzliche Design der Website ist auf Übersichtlichkeit ausgerichtet. Durch einen simplen und eindeutigen Seitenaufbau soll sich der Benutzer gut zurecht finden. Farblich wurden schlichte Töne wie Weiß und Grau eingesetzt. Beispielhaft für das allgemeine Design der Website ist in Abbildung 3.4 die Welcome Page in der Ansicht des Admins zu sehen.

Tweabler Account Logout

---

**Datasets**


Name	Description	Number of Tweets	Crosslabel-Number	Relabel-Initial					
PosNegNoSpam	Datenset zum Benchmarken von Algorithmen.	1327	2	65% - 35%	Labeling	Show	Add Tweets	Add Users	Destroy
Testset	Datenset zum Testen der Applikation	9	4	30% - 70%	Labeling	Show	Add Tweets	Add Users	Destroy

New Dataset

**Figure 3.4:** Welcome Page der Applikation in der Ansicht des Admins

Um Übersichtlichkeit und Aussagekraft zu unterstützen, wurden auf der Dataset Show Page verschiedene Elemente eingesetzt. Zum einen gibt es interaktive Kontrollelemente, mit denen sich die Zahl der Tweets bezüglich der Crosslabelzahl des Datensets oder der Anzahl der identischen erhaltenen Labels für das Datenset anzeigen lassen. Diese Kontrollelemente sind in Abbildung 3.5 zu sehen.

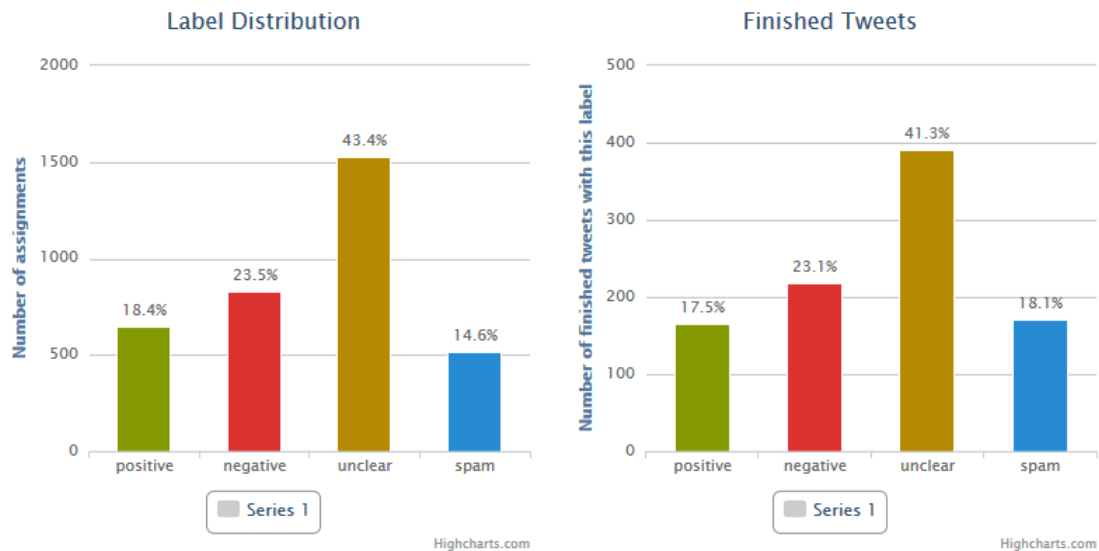
Zum anderen werden auf der Dataset Show Page zwei Diagramme angezeigt, zu sehen in Abbildung 3.6. Im linken Diagramm wird für jedes Label die Anzahl der Zuweisungen

With Crosslabel-Number 2  count of finished tweets: **1467**

Tweets with at least 2  identical labels: **861**

**Figure 3.5:** Interaktive Kontrollelemente auf der Dataset Show Page

im Labelprozess angezeigt. Über dem jeweiligen Balken steht die Prozentzahl für das Label, fährt man mit der Maus über den Balken wird die absolute Zahl angezeigt. Das rechte Diagramm zeigt für jedes Label die Anzahl der fertigen Tweets; das heißt die Anzahl der Tweets, die bereits von einer der Crosslabelzahl entsprechenden Anzahl an Benutzern mit diesem Label bewertet wurden. Auch hier sind sowohl Prozentzahl als auch die absolute Zahl einsehbar.



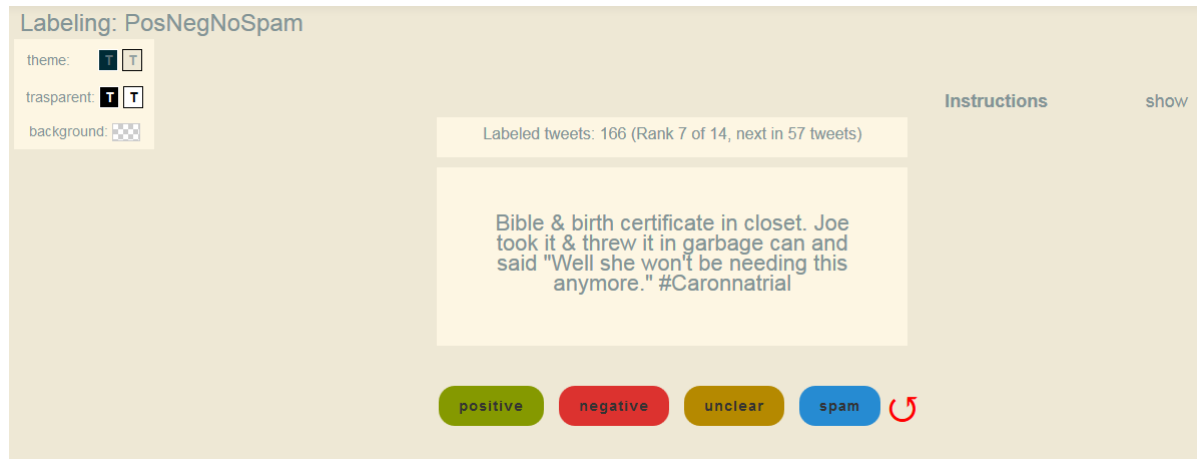
**Figure 3.6:** Diagramme auf der Show Dataset Page

Auf der Label Page wurde nicht das allgemeine Design der Website angewendet. Da Benutzer lange auf dieser Seite arbeiten können sollten, musste hier besonders viel Wert auf die Usability der View gelegt werden. Die weiße Hintergrundfarbe und die starken Kontraste des allgemeinen Designs wären auf Dauer während des Labelingprozesses zu anstrengend für die Augen, daher wurde hier anders vorgegangen.

Die View auf der Label Page wurde so gestaltet, dass der Benutzer sie in einem gewissen Maß an seine Bedürfnisse anpassen kann. So kann der Benutzer die Website an seine Situation und seine Vorlieben angleichen und ist flexibel. Das Default-Design der Labelpage ist in Abbildung 3.7 dargestellt. Es ist in den Farben des hellen Solarized Farbschemas<sup>1</sup> gehalten. Das Solarized Farbschema gilt als besonders augenfreundlich, es

<sup>1</sup><http://ethanschoonover.com/solarized>

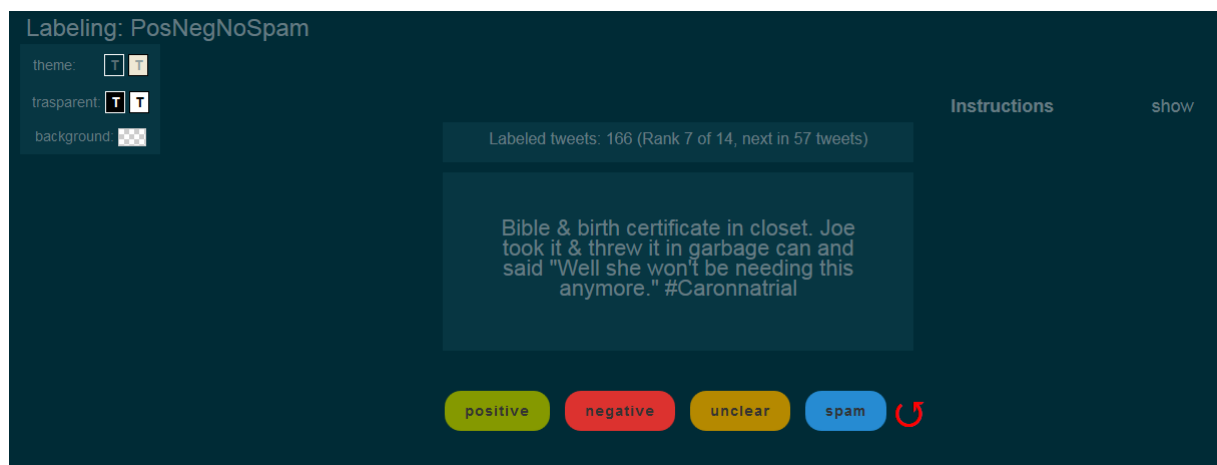
wurde von Ethan Schoonover vor allem für Programmierer entwickelt, die häufig lange an Bildschirmen arbeiten müssen. Die Default Farben für Labels entsprechen ebenfalls Farben aus dem Solarized Farbschema.



**Figure 3.7:** Label Page mit den Default Farben

Oben links in der Ecke der Label Page ist ein Kasten mit den Einstellmöglichkeiten der Seite zu sehen. Die beiden Buttons hinter *theme:* stellen jeweils das dunkle bzw. helle Solarized Farbschema ein. Das dunkle Solarized Farbschema ist in Abbildung 3.8 zu sehen.

Die Buttons hinter *transparent:* stellen einen transparenten dunklen bzw. hellen Modus ein. Mit dem letzten Button lässt sich dann die Hintergrundfarbe individuell einstellen, welche durch die transparenten Flächen durchscheint. In der Abbildung 3.9 ist der dunkle transparente Modus mit einer grünen Hintergrundfarbe zu sehen, der helle transparente Modus mit einem hellen Orangeton als Hintergrund ist in Abbildung 3.10 zu sehen.



**Figure 3.8:** Label Page mit den dunklen Solarized Farben

Auf der Label Page befinden sich auf der rechten Seite die Anleitungen für Datenset, Labels und, falls vorhanden, Kategorien. Diese werden beim ersten Aufruf der Seite angezeigt. Der Nutzer kann diese mit dem Button *hide* ausblenden. Diese Einstellung

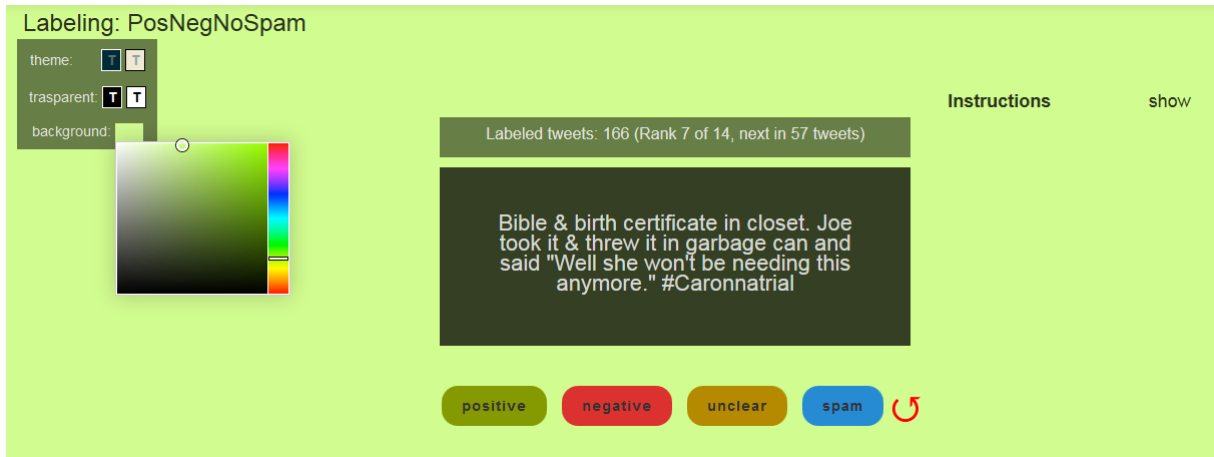


Figure 3.9: Label Page im dunklen transparenten Modus

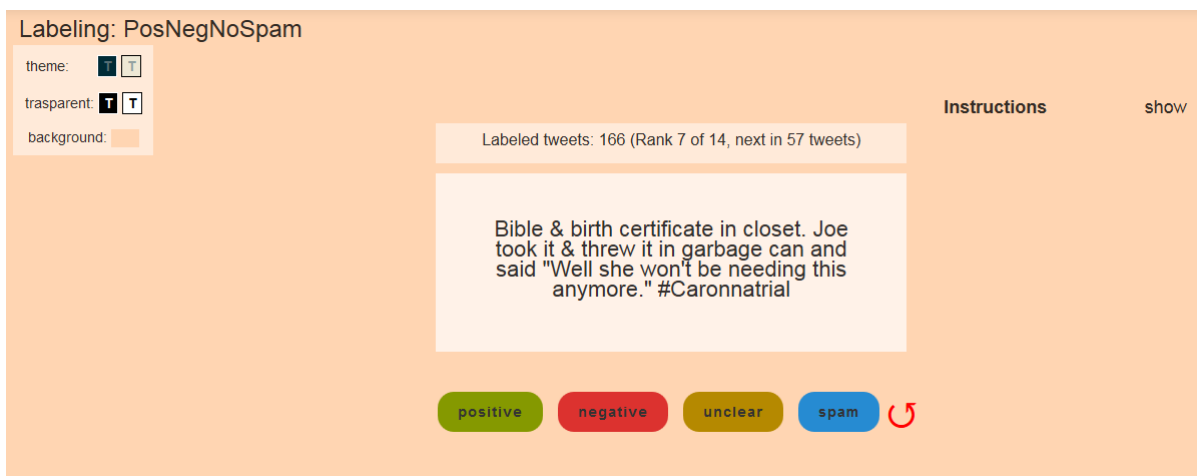


Figure 3.10: Label Page im hellen transparenten Modus

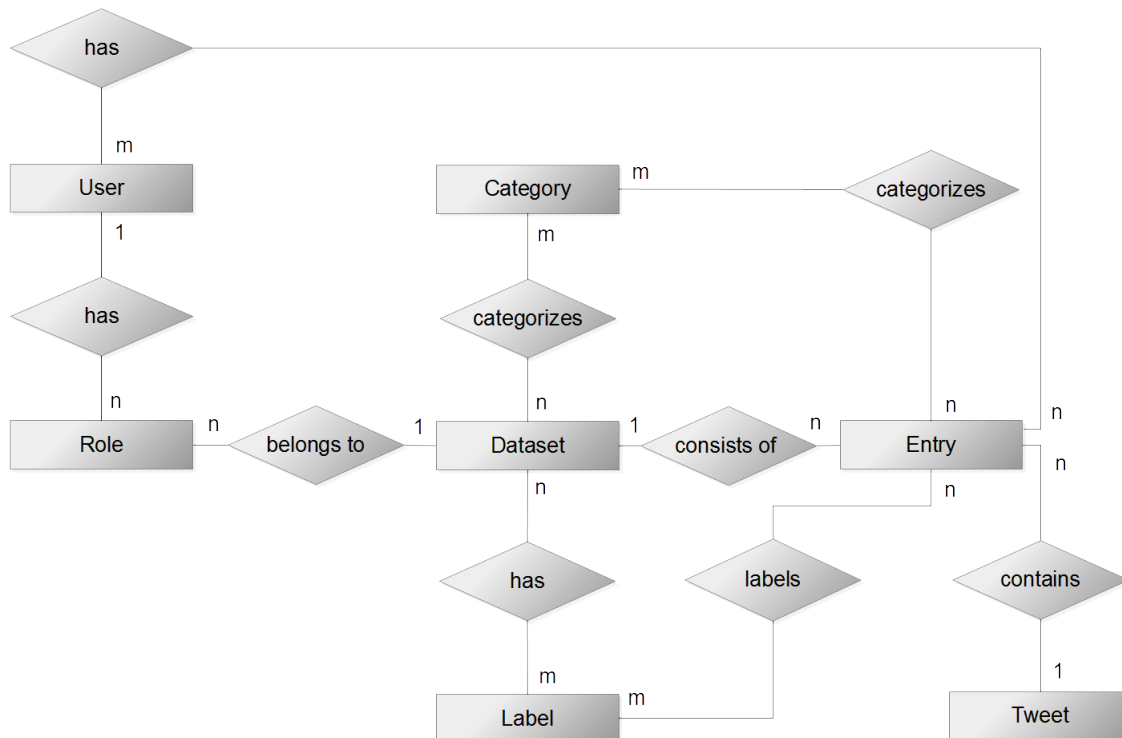
wird gespeichert für den nächsten Aufruf der Seite. Der Benutzer kann jederzeit mit *show* die Anleitungen wieder einblenden lassen.

### 3.3 Back-End

In diesem Abschnitt der Arbeit wird das Back End der Applikation vorgestellt. Dafür werden Datenbankschema und zugrundeliegende Modelklassen der Applikation präsentiert. Der Übersichtlichkeit dienend wird die zeitliche Abfolge der Entwicklung der Modelklassen und des Datenbankschemas vernachlässigt und beides als Ganzheit dargestellt.

In Rails ergibt sich das Datenbankschema implizit durch Festlegung der Modelklassen und deren Beziehungen untereinander. Ein Model erhält je eine Tabelle in der Datenbank. Um eine direkte  $n:m$ -Beziehung abzubilden, wird eine weitere Tabelle benötigt. Im Allgemeinen wird eine  $n:m$ -Beziehung jedoch durch ein Join Model umgesetzt, insbesondere wenn die Beziehung eigene Attribute benötigt. Dieses Join Model erhält ebenfalls

eine Tabelle mit den IDs der abzubildenden Modelklassen als Fremdschlüssel in der Datenbank. In Abbildung 3.11 sind die wichtigsten Modelklassen der Applikation mit ihren Relationen dargestellt. Einige Join Models wurden in dieser Abbildung der Übersichtlichkeit halber vernachlässigt, werden jedoch in Abschnitt 3.3.1 vorgestellt.



**Figure 3.11:** Vereinfachte Fassung eines Entity-Relationship Diagramms der Applikation mit Kardinalitäten in der Chen-Notation

### 3.3.1 Die Modelklassen

Ein **Dataset** besteht aus vielen **Entries** (Einträge). Ein Eintrag ist immer nur einem Dataset zugehörig. Dieses kann ohne Einträge existieren, zum Beispiel wenn noch keine Daten importiert wurden. Ein Eintrag muss jedoch immer einem Dataset zugewiesen sein.

In einem Eintrag ist genau ein **Tweet** enthalten. Einträge werden als Zwischenglied zwischen Dataset und Tweet benötigt, um bei multiplen Datensets die Daten der Tweets, welche in verschiedenen davon enthalten sind, nicht mehrfach in der Datenbank vorhalten zu müssen und damit Redundanzen zu vermeiden. Für Tweets ist diese Beziehung optional, ein Eintrag kann jedoch ohne Tweet nicht bestehen.

Einem Dataset können mehrere **Labels** und **Categories** (Kategorien) zugewiesen werden. Ein Label kann mehreren Datensets zugewiesen werden, gleiches gilt für Kategorien. Die Beziehung zwischen Dataset und Label bzw. Kategorie ist optional. Die Zuweisung von Labels bzw. Kategorien zu einem Dataset bedeutet, dass sie für die über die

Einträge im Datenset enthaltenen Tweets im Labelprozess für die Bewertung zur Verfügung stehen. Diese Beziehung ist eine **n:m**-Beziehung, welche durch die Join Models **LabelRange** (Label Palette) und **CategoryRange** (Kategorien Palette) umgesetzt wurde.

Labels und Kategorien können einem Eintrag zugewiesen werden. Diese Beziehung ist eine optionale **n:m**-Beziehung, als Join Model für Labels wird **Rating** (Bewertung) eingesetzt, für eine Kategorie ist es die **Categorization** (Kategorisierung). Diese Beziehung stellt die Bewertung des im Eintrag enthaltenen Tweets mit dem Label bzw. der Kategorie dar.

Für die Benutzerverwaltung wird das Model **User** (Benutzer) verwendet. Ein Benutzer kann viele **Roles** (Rollen) haben, welche wiederum einem Datenset zugewiesen sind. Eine Rolle gehört immer zu einem Benutzer und einem Datenset. Ein Benutzer kann für verschiedene Datensets je unterschiedliche Rollen einnehmen.

Die einzige **n:m** Beziehung der Applikation, welche ohne Join Model abgebildet wurde, ist die zwischen Benutzer und Eintrag. Diese optionale Beziehung speichert, welche Einträge im Datenset der Benutzer schon bewertet hat, jedoch nicht womit sie bewertet wurden.

## 3.4 Datensets

Das Ziel dieser Applikation ist es, geeignete Datensets für die Stimmungsanalyse von Tweets zu erstellen. Sie bilden somit das Zentrum der Applikation. Daher wird die Datenset Klasse der Applikation in diesem Abschnitt näher beleuchtet.

Ein Datenset hat in dieser Applikation entsprechend der Anforderungen (vgl. 3.1) die folgenden Attribute:

- name:string Name des Datensets
- description:text Beschreibung des Datensets
- relabel:integer Relabelanteil des Datensets in Prozent
- crosslabel:integer Crosslabelzahl des Datensets
- instruction:text Anleitung für das Labeln des Datensets

Wie bereits in Abschnitt 3.3.1 erwähnt, werden einem Datenset durch Einträge Tweets zugewiesen. Den Einträgen werden während des Labelprozesses Kategorien und Labels zugewiesen. Dieser Labelprozess wird in Abschnitt 3.5 genauer erläutert.

### 3.4.1 Import

Um ein Datenset mit Tweets zu füllen, müssen die Tweets mithilfe des Import Formulars importiert werden. Dafür wird in einem Dateiauswahldialog eine CSV-Datei mit dem Format `original_id`, `content` ausgewählt. Diese wird als temporäre Datei im Browser gespeichert. Dann kann sie zeilenweise vom CSV-Manager Controller ausgelesen und zu Tweets verarbeitet werden. Dass die Verarbeitung zeilenweise vorgenommen wird, hat den Vorteil, dass die Datei nicht erst ganz in den Speicher geladen werden muss, was bei großen Dateien sehr lange dauern würde. Bei der Verarbeitung wird ein Tweet mit `content` und `originalID` angelegt und ein Eintrag erstellt, der den Tweet dem Datenset zuordnet. Falls schon ein Tweet mit der eingelesenen `originalID` in der Datenbank vorhanden ist, wird nur der Eintrag angelegt, der diesen dem Datenset zuordnet.

### 3.4.2 Export

Um die Datensets der Applikation später als Testdatensets für die Stimmungsanalyse von Tweets verwenden zu können, müssen diese zuerst exportiert werden. Dafür ist das Formular auf der Export Page vorgesehen, welches in Abbildung 3.12 zu sehen ist.

<b><u>Labels:</u></b>		<b><u>Categories:</u></b>		<b>Crosslabel-Number:</b>	<input type="text" value="2"/>
positive	<input checked="" type="checkbox"/>	Unspecified	<input type="checkbox"/>		
negative	<input checked="" type="checkbox"/>			<b>Matching-Number:</b>	<input type="text" value="2"/>
unclear	<input type="checkbox"/>				
spam	<input type="checkbox"/>			<b>Filename:</b>	<input type="text" value="Tweabler_Export.csv"/>
<input type="button" value="Submit"/>					

**Figure 3.12:** Formular für den Datenexport

Es wird eine CSV-Datei im Format: `original_id`, `content`, `label`, `category` erstellt. Um den Export möglichst variabel zu gestalten, kann der User verschiedene Einstellungen vornehmen. Es kann ausgewählt werden, welche Labels in den Exportdaten vorhanden sein dürfen, ebenso können Kategorien ausgewählt werden. Mit *Unspecified* gibt man hier an, dass auch Tweets ohne Kategorie exportiert werden sollen. Zudem kann unabhängig von der Einstellung im Datenset die Crosslabelzahl für den Export eingestellt werden. Außerdem kann die Mindestzahl identischer Labels für einen Tweet eingestellt werden, der im Datenset landet. Unter Filename kann der User dann noch den Namen der CSV-Datei einstellen.



## 3.5 Labelprozess

Im Labelprozess wird ein Datenset mit den Werten gefüllt, die später als Vergleichswerte für maschinelle Labeler genutzt werden sollen. Es werden also den Tweets, welche dem Datenset über Einträge zugeordnet sind, Labels und optional auch Kategorien zugewiesen. Dafür steht die Label Page zur Verfügung.

Die Seite für den Labelprozess ist wie folgt aufgebaut (siehe Abbildung 3.7): In der Mitte wird der Tweet angezeigt, der gelabelt werden soll. Darunter befinden sich die Buttons mit den Labels, aus denen gewählt werden muss. Falls Kategorien auswählbar sind, stehen sie links in einer Auswahlbox zur Verfügung. Oben über dem Tweet ist das Ranking des Users zu sehen. Rechts vom Tweet stehen die Anleitungen von Datenset, Labels und Kategorien zur Verfügung.

Für die User ist der Ablauf beim Labeln wie folgt:

- Der aktuell zu lesende Tweet wird angezeigt
- Optional: Die passende Kategorie wird gewählt
- Das passende Label wird angeklickt
- Der nächste Tweet wird angezeigt
- Das Ranking wird aktualisiert

Der Ablauf während des Labelprozesses im Hintergrund ist so aufgebaut:

- Laden der nötigen Daten aus dem HTML-Dokument via CoffeeScript
- Weiterleiten dieser Daten an die Methode `nextTweet` im Page Controller via AJAX
- `nextTweet` versieht den aktuellen Tweet mit dem Label und der Kategorie
- `nextTweet` ruft die Methode `ranking` zum Aktualisieren der Ranking-Daten auf
- `nextTweet` ruft die Methode `nextID` zum Bestimmen des nächsten zu labelnden Tweets auf
- `nextTweet` liefert die gesammelten Daten im JSON Format an den CoffeeScript-Interpreter zurück
- Der CoffeeScript-Interpreter aktualisiert die Label Page um die Daten, unter anderem werden der neue Tweet angezeigt und das Ranking angepasst

Um einen versehentlichen Klick auf das falsche Label ausgleichen zu können, kann der User jederzeit den zuletzt gelabelten Tweet noch einmal aufrufen und die Bewertung ändern.

Die Methode `nextID` im PageController ist dafür zuständig, abhängig vom Relabelanteil des Datensets, den neuen Tweet für den Labelprozess auszuwählen. In Listing 12 ist der Code der Methode aufgeführt. In dieser Methode wird zuerst eine Zufallszahl zwischen 0 und 100 generiert (Zeile 6). Diese wird dann mit dem Relabelanteil des Datensets verglichen. Ist die Zahl kleiner oder gleich groß, gelangt man in den Relabelmodus (Zeilen 7-19), ist die Zahl größer, wird der Initialmodus gewählt (Zeilen 20-32).

```

1  def nextID(datasetID)
2    if not session[:user_id].nil?
3      @nt = 0
4      @d= Dataset.find_by(id: datasetID)
5      success = false
6      random = rand(0..100)           #relabel or initial?
7      if random <= @d.relabel        #relabel mode
8        /get max. 50 possible next relabel entries/
9        @labeled = User.get_valid_relabel_ids(@user, @dataset)
10       if @labeled.count >0
11         random = rand(0..@labeled.count-1) #take one random
12         entry = Entry.find_by(id: @labeled[random])
13         if not entry.nil?
14           @nt = entry.tweet           #load related tweet
15           success = true
16         end
17       end
18     end
19     if success == false             #initial mode or no relabel tweets
20       /get max. 50 possible next initial entries/
21       @unlabeled = User.get_valid_initial_ids(@user, @dataset)
22       if @unlabeled.count > 0
23         random = rand(0..@unlabeled.count-1) #take one random
24         entry = Entry.find_by(id: @unlabeled[random])
25         if not entry.nil?
26           @nt = entry.tweet           #load related tweet
27         end
28       end
29     end
30     return @nt                       #return selected Tweet
31   else
32     redirect_to login_path
33   end
34 end

```

**Listing 12:** Die Methode nextID aus dem PageController

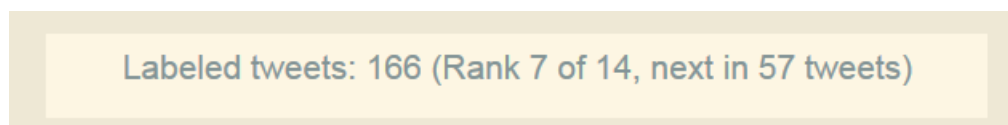
Im Relabelmodus wird die Methode `get_valid_relabel_ids` des User Models aufgerufen (Zeile 9). Diese liefert maximal 50 EntryIDs des Datensets in einem Array zurück, deren Tweets in diesem Datenset schon von anderen Usern gelabelt wurden, aber noch nicht wegen der Crosslabelzahl oder Garbage Labels herausfallen.

Sind keine solchen Tweets vorhanden, wird in den Initialmodus gewechselt. Ansonsten wird eine Zufallszahl `random` zwischen 0 und der Anzahl der zurückgelieferten IDs -1 generiert (Zeile 11). Dann wird der Eintrag mit der ID, die an der Stelle `n` im Array steht geladen, um im Anschluss dessen Tweet zu laden (Zeilen 12-15).

Der Initialmodus ist sehr ähnlich aufgebaut, allerdings wird dort anstatt der Methode `get_valid_relabel_ids` die Methode `get_valid_initial_ids` aufgerufen. Diese liefert maximal 50 EntryIDs des Datensets in einem Array zurück, deren Tweets in diesem Datenset noch nicht gelabelt wurden. Am Schluss liefert die Methode den ausgewählten Tweet zurück an `nextTweet` (Zeile 30).

## Ranking

Für den Verwender der Applikation, welcher möglichst schnell, möglichst aussagekräftige Datensets erhalten möchte, ist es wichtig, dass die User motiviert sind, viele Tweets zu labeln. Der Vergleich mit anderen Usern soll diese Motivation beim Labelprozess steigern.



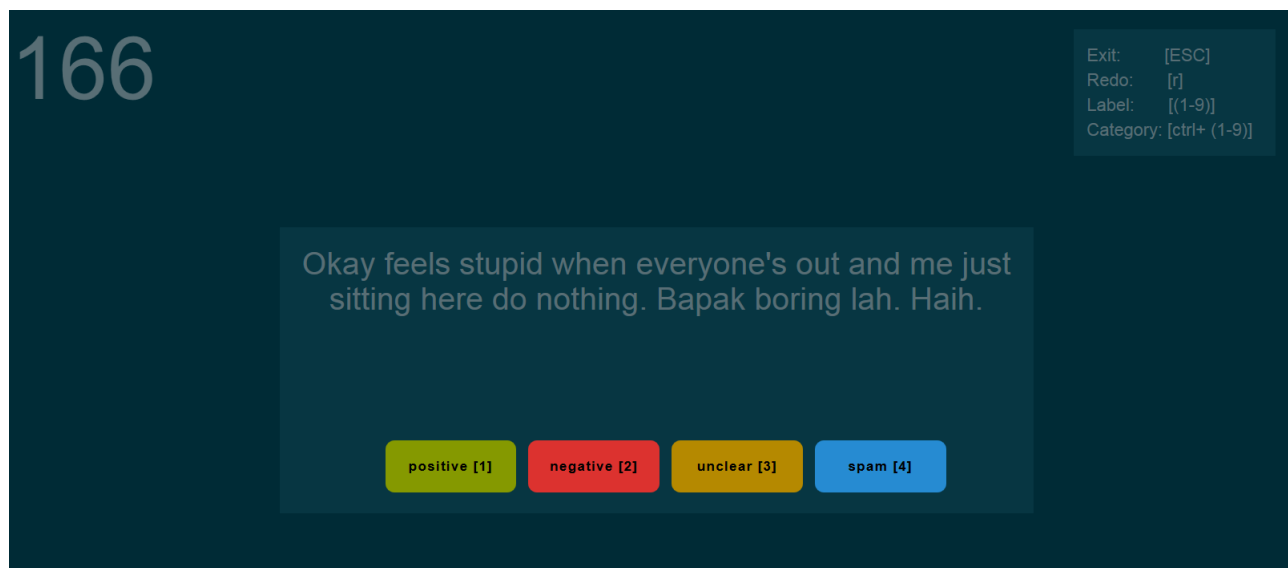
**Figure 3.13:** Ranking eines Users auf der Label Page

Um also dem User einen guten Eindruck davon zu verschaffen, wie viele Tweets er schon gelabelt hat und wie er im Verhältnis zu anderen Usern steht, wird über dem Tweet auf der Labelpage das Ranking angezeigt, zu sehen in Abbildung 3.13. Dieses beinhaltet die konkrete Zahl der aktuell vom User gelabelten Tweets, die Stelle, welche der User in der Liste der User, sortiert nach Anzahl der gelabelten Tweets einnimmt und wie viele Tweets er labeln muss, um das nächsthöhere Ranking zu erreichen. Nimmt der User den ersten Platz ein, wird dagegen die Anzahl der Tweets, die er dem User auf dem zweiten Platz voraus ist, angezeigt.

## Shortcutmode

Zusätzlich zum Labeln auf der normalen Label Page ist es dem User möglich, in den Shortcutmode zu wechseln. Er kann diesen entweder über einen Button in der unteren linken Ecke oder über die Tastenkombination `Shift+S` starten. Der Extramodus wurde speziell dazu entwickelt, den Labelprozess noch intuitiver und zeitsparender zu gestalten. Die Besonderheit des Shortcutmodes ist, dass hier die Labels und Kategorien per Tastendruck zugewiesen werden können.

Der Aufbau der Label Page im Shortcutmode ist in Abbildung 3.14 zu sehen. Als Farbschema wurde das dunkle der beiden Solarized Farbschemata ausgewählt, da die Inhalte in dessen Farben auch bei dunkler Umgebung gut zu lesen sind und die Augen nicht angestrengt werden. Bei der hellen Variante wäre eine sehr dunkle Umgebung eventuell ungünstig. Die Seite ist möglichst simpel gestaltet. Zu diesem Zweck wird die Navigationsleiste ausgeblendet. Oben links in der Ecke befindet sich die Anzahl der vom User gelabelten Tweets. Auf den Rang des Users wird hier verzichtet, um jede unnötige Ablenkung zu vermeiden. In der Mitte wird relativ groß der Tweet angezeigt. Oben rechts sind die Tasten zur Kontrolle des Shortcutmodes erklärt. Mit `ESC` kann



**Figure 3.14:** Aufbau der Label Page im Shortcutmode

der Modus verlassen werden. Mit `r` kann der zuletzt gelabelte Tweet erneut aufgerufen werden, um die Bewertung zu ändern. Mit Zahlen werden Labels ausgewählt, mit der Kombination aus `CTRL` und einer Zahl wählt man die Kategorie.

Unter dem Tweet sind die Labels bzw. die Kategorien zu sehen. Auf ihren Buttons steht die Tastenkombination, mit der das entsprechende Label bzw. die entsprechende Kategorie gewählt werden kann. Anders als beim normalen Labelmodus werden Kategorien und Labels nicht gleichzeitig angezeigt. Stehen Kategorien zur Verfügung, werden erst nur diese gezeigt. Wurde eine Kategorie ausgewählt, verschwinden die Kategorien und die Labels werden stattdessen sichtbar. Wurde dann das Label gewählt, erscheint ein neuer Tweet, die Labels verschwinden und die Kategorien sind wieder zu sehen. Falls keine Kategorien auswählbar sind, sind immer die Labels sichtbar.

## 3.6 AJAX

Mit **AJAX** (aus dem englischen, *Asynchronous JavaScript and XML*) ist eine asynchrone Datenübertragung zwischen Browser und Server möglich. Die gängigsten Browser sind in der Lage JavaScript zu interpretieren. Das macht es möglich von JavaScript ausgehende HTTP-Anfragen zu senden, nachdem eine HTML-Seite schon geladen wurde und diese zu verändern, ohne sie neu laden zu müssen. In dieser Applikation werden dafür mithilfe des REST Ansatzes über klassische HTTP Anfragen Methoden in den Model Controllern aufgerufen. Diese Anfragen liefern Daten im textbasierten JSON Format an den JavaScript Interpreter zurück, welcher mit den Daten die aktuell gezeigte HTML Seite anpassen kann. In dieser Applikation wird statt JavaScript CoffeeScript verwendet, welches zu Javascript kompiliert <sup>2</sup>.

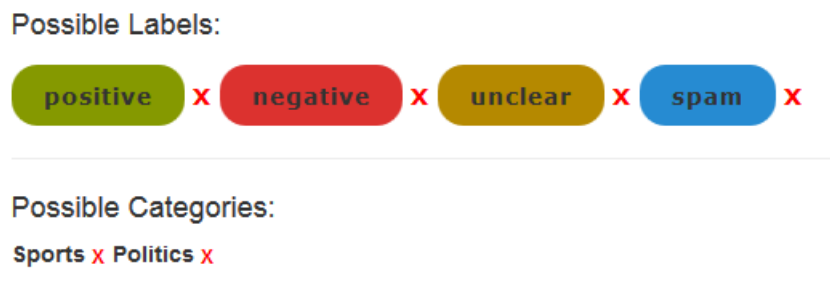
<sup>2</sup><http://coffeescript.org/>

Es ist sinnvoll AJAX an Stellen einer Webapplikation einzusetzen, an denen der Benutzer die Seite manipulieren und möglichst ohne große Verzögerung die Ergebnisse sehen können soll. In dieser Applikation wurde AJAX hauptsächlich an zwei Stellen eingesetzt: der Übersichtsseite eines Datensets (Dataset Show Page) und der Labelpage. In diesem Kapitel wird der Einsatz dieser Technologie anhand der zwei Beispiele näher erläutert.

### 3.6.1 AJAX auf der Dataset Show Page

Auf der Übersichtsseite eines Datensets wird AJAX für zwei Funktionalitäten eingesetzt. Zum einen für das Löschen von Labels und Kategorien, zum anderen für die Kontrollelemente, die Auskunft über Anzahl der Tweets im Zusammenhang mit der Crosslabelzahl bzw. der Anzahl identischer Labels geben.

Abbildung 3.15 zeigt einen Ausschnitt der Dataset Show Page mit den Labels und Kategorien des Datensets. Um Labels bzw. Kategorien zu löschen, muss der Nutzer auf das rote x neben Label/Kategorie klicken. Es erscheint ein Bestätigungsdialog, um ein versehentliches Klicken auszuschließen, da sich das Löschen nicht rückgängig machen lässt. Wurde im Dialog das Löschen bestätigt, verschwindet das Label von der Seite, ohne dass diese neu geladen werden muss. Wie dies umgesetzt wird, ist in Listing 13 zu sehen.



**Figure 3.15:** Ausschnitt der Dataset Show Page mit den Labels und Kategorien des Datensets

In den Zeilen 2-5 werden den Buttons zum Löschen der Labels und Kategorien Event Handler zugewiesen. Wird auf den Button eines Labels geklickt, soll die Methode `delLabel` mit der ID des Buttons aufgerufen werden. Ist es der Button einer Kategorie wird entsprechend `delCategory` mit der Button ID aufgerufen. In den Zeilen 7-27 ist die `delLabel` Methode zu sehen. Die Zeilen 8-9 erzeugen den Bestätigungsdialog. In den Zeilen 10-13 werden die Label ID und die Farbe des Label Buttons aus der HTML Seite geladen. Diese Daten werden in den Zeilen 14-21 zusammen mit der ID des Buttons via REST an den DatasetController gesendet. Um die verkürzte URL `/del` verwenden zu können, wurde diese in der Konfigurationsdatei für Routen `routes.rb` so definiert, dass die Anfrage an den DatasetController geleitet wird. Dieser erhält die Daten und kann mit deren Hilfe das richtige Label löschen. Allerdings sind ohne ein erneutes Laden der Seite das Label und der Button zum Löschen des Labels immer noch zu sehen. Deshalb werden diese bei erfolgreicher Antwort des Controllers in Zeile 24-27 ausgeblendet, indem sie die CSS Eigenschaft `display='none'` erhalten. Bei erneutem Aufruf der Seite

```

1 $(document).ready ->
2   $('button.xl').click ->
3     delLabel this.id
4   $('button.xc').click ->
5     delCategory this.id
6
7 delLabel = (buttonID) ->
8   if window.confirm('Are you sure? This will delete
9     all ratings of this label from dataset')
10    labelID = document.getElementById(buttonID).
11      getAttribute("data-label")
12    color = document.getElementById(buttonID).
13      getAttribute("data-color")
14    $.ajax
15      type: "POST"
16      dataType: "json"
17      url: "/dell"
18      data:
19        labelID: labelID
20        buttonID: buttonID
21        color: color
22
23      success: (data) ->
24        document.getElementById(data.color).
25          style.display="none"
26        document.getElementById(data.button).
27          style.display = 'none'

```

**Listing 13:** Ausschnitt des CoffeeScripts der Dataset Klasse

werden diese Elemente dann nicht mehr geladen. Für den User ist es von Vorteil, wenn nicht nach jedem Löschen eines Labels oder einer Kategorie die Seite neu lädt, er aber trotzdem seine Änderung direkt sehen kann.

With Crosslabel-Number **2**  count of finished tweets: **1467**

Tweets with at least **2**  identical labels: **861**

**Figure 3.16:** Kontrollelemente der Dataset Show Page

Abbildung 3.16 zeigt einen Ausschnitt der Dataset Show Page mit zwei Kontrollelementen zur Darstellung der Anzahl der Tweets im Zusammenhang mit der Crosslabelzahl im oberen, der Anzahl identischer Labels im unteren Fall. Die jeweils linken Zahlen sind

mithilfe eines Number Fields dargestellt, welches über eine direkte Eingabe oder über die Pfeiltasten auf einen neuen Wert gesetzt werden kann. Die rechten Zahlen sollen sich dann dementsprechend anpassen. Hier wäre es extrem störend, wenn jedes Mal die Seite neu laden würde. Daher ist die Verwendung von AJAX nötig gewesen.

```
1  $(document).on "change", "#x", ->
2    x = $(this).val()
3    $.ajax
4      type: "POST"
5      dataType: "json"
6      url: "/x"
7      data:
8        dataset: $(this).data('dataset')
9        x: x
10
11     success: (data) ->
12       $("#label_count").val(data.x)
```

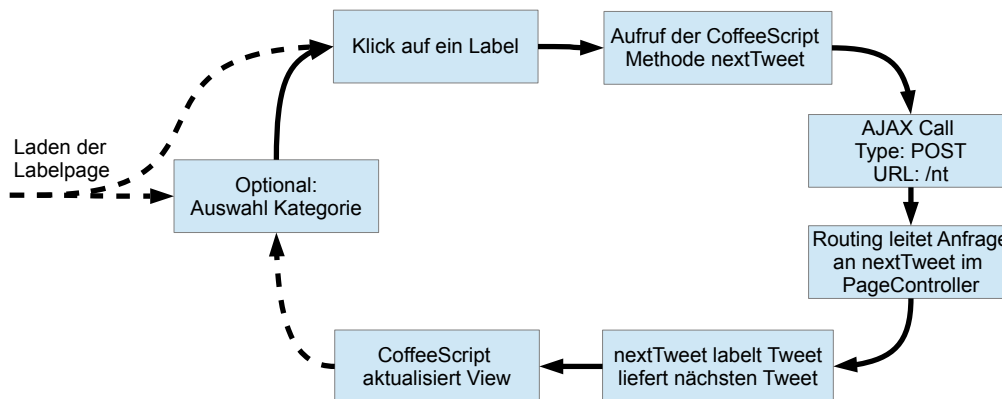
**Listing 14:** Weiterer Ausschnitt des CoffeeScripts der Datenset Klasse

In Listing 14 ist das CoffeeScript zu sehen, welches die Funktionalität des oberen Kontrollelements bereitstellt. Zeile 1 sorgt dafür, dass der Code in den Zeilen 2-12 ausgeführt wird, wenn sich das HTML Element mit `id="x"` verändert, also entweder ein neuer Wert eingegeben oder eine der Pfeiltasten gedrückt wurde. Zeile 2 lädt den Wert des Numberfields aus dem HTML Dokument und speichert ihn in `x`. Dann wird in den Zeilen 3-9 mit AJAX die Methode `x_labels` im `DatasetController` aufgerufen: Ihr werden das `x` und die ID des aktuell gezeigten Datensets übergeben. Die `x_labels`-Methode führt dann mithilfe der übergebenen Werte eine Datenbankabfrage aus und sendet das Ergebnis, die Anzahl der Tweets, welche von mindestens `x` verschiedenen Usern gelabelt wurden, im JSON Format zurück. Bei erfolgreichem AJAX Aufruf wird nun durch Zeile 12 der Wert der rechten Zahl des Kontrollelementes auf den zurückgelieferten Wert gesetzt.

### 3.6.2 AJAX auf der Labelpage

Noch wichtiger als auf der Dataset Show Page war der Einsatz von AJAX auf der Labelpage (Abbildung 3.7). Bei dieser Seite steht die Usability stark im Vordergrund. Hier wäre es für den Nutzer störend, wenn die Seite bei jedem Labelingschritt neu geladen werden müsste. Auf dieser Seite ist so ziemlich jede Funktion mithilfe von AJAX umgesetzt: das Verstecken/Anzeigen der Anleitung, die Einstellbarkeit der Viewelemente über Theme, Transparenz und Hintergrundfarbe sowie der gesamte Labelingprozess.

Der Ablauf des Labelingprozesses ist in vereinfachter Form in Abbildung 3.17 zu sehen. Wird die Labelingseite aufgerufen, ist ein labelbarer Tweet zu sehen. Stehen Kategorien zur Verfügung, kann eine ausgewählt werden, doch dieser Schritt ist optional. Wird auf einen der Labelbuttons gedrückt, ruft der Button-Listener die CoffeeScript Methode `nextTweet` auf. Hier werden die nötigen Daten gesammelt und mit AJAX and die URL



**Figure 3.17:** Vereinfachte Darstellung des Ablaufs des Labelingprozesses

`/nt` verschickt. Der Router leitet die Anfrage dann an die `nextTweet` Methode im PageController weiter. Die Methode versieht den aktuellen Tweet mit dem ausgewählten Label und, falls nötig, der gewählten Kategorie. Dann ruft sie die Methode `nextID` des PageControllers auf, welche die ID des nächsten zu labelnden Tweets zurückliefert. Außerdem werden Daten für das Ranking des Users gesammelt. Sind alle Daten zusammen, werden diese im JSON Format zurück zum CoffeeScript-Interpreter geleitet. Dort wird die View um die entsprechenden Daten aktualisiert und der Prozess kann von vorne beginnen.

## 3.7 Testphase

Zum Ende der Entwicklungszeit sollte die Applikation in einem möglichst anwendungsnahen Kontext ausführlich getestet werden. Dafür wurde die Applikation auf dem Server der Universität Osnabrück bereitgestellt. Es wurde ein Datenset angelegt, welches in etwa der Größe der Datensets in der Endanwendung entspricht. Die Applikation wurde den Tutoren aus dem Wintersemester 2014/15 der Veranstaltung Informatik A zugänglich gemacht, welche die Rolle von Labelern zugewiesen bekommen haben und den Auftrag erhielten, möglichst viele Tweets zu labeln und über eventuelle Fehler und Unstimmigkeiten Bericht zu erstatten.

### 3.7.1 Laufzeitoptimierung

Nach ein paar Tagen der Benutzung stellte sich ein Optimierungsbedarf der Applikation heraus, da die Label Page sehr langsam geworden war. Wenn ein Label für den aktuell gezeigten Tweet ausgewählt wurde, musste bis zu 3 Sekunden darauf gewartet werden, bis der neue Tweet geladen war und erschien. Da ein angenehmer Labelingprozess für den Anwender voraussetzt, dass die Seite möglichst kurze Ladezeiten hat, musste die Applikation dahingehend untersucht werden, was die zeitliche Verzögerung verursacht hatte.



Abbildung 3.18 zeigt die Konsolenausgabe der Methode `next_tweet` des `Page Controllers`, welche den aktuellen Tweet mit dem ausgewählten Label versieht und einen neuen Tweet lädt, zum Zeitpunkt, als die Verzögerung festgestellt wurde. Die Methode hatte in etwa eine Ladezeit von 1,7 Sekunden, manchmal sogar mehr. Es musste also diese Methode auf die Ursache der zeitlichen Verzögerung hin untersucht werden.

```
I, [2015-03-25T15:14:28.602898 #14860] INFO — : Started POST "/nt" for
131.173.13.63 at 2015-03-25 15:14:28 +0100
I, [2015-03-25T15:14:28.606251 #14860] INFO — : Processing by
PageController#next_tweet as JSON
I, [2015-03-25T15:14:28.606357 #14860] INFO — : Parameters:
{"dataset"=>"2", "label"=>"2", "tweet"=>"10393", "category"=>"null"}
I, [2015-03-25T15:14:30.418278 #14860] INFO — : Completed 200 OK in
1812ms (Views: 0.1ms | ActiveRecord: 1555.0ms)
```

**Figure 3.18:** Konsolenausgabe bei Laden des nächsten Tweets auf der Labelpage inklusive Ladezeit vor Verbesserungsmaßnahmen.

Die Untersuchung der Methode `next_tweet` ergab, dass die in ihrem Rumpf aufgerufene Methode `nextID` die Verzögerung verursachte. Diese Methode ist dafür zuständig die ID des nächsten zu labelnden Tweets auszuwählen, unter Berücksichtigung von Relabel- oder Initialmodus, schon vom User gelabelten Tweets und durch Garbagelabel vom Labelingprozess ausgeschlossene Tweets. Um gültige Relabel- bzw. Initialmodus Tweets zu erhalten, müssen entsprechende Daten aus der Datenbank geladen werden. Zum Zeitpunkt des Optimierungsbedarfs wurden diese Queries mithilfe von Rails ActiveRecord Query Interface formuliert, zu sehen in Listing 15. Dort werden für die Initialmodus

```
1 @unlabeled = Entry.includes(:ratings).where(
2     :ratings => {:id => nil},
3     dataset_id: datasetID).pluck(:id)-
4     forbidden
5
6 @labeled = Entry.where(dataset_id: @d.id).
7     pluck(:id) - forbidden - @unlabeled
```

**Listing 15:** Queries zur Vorbereitung der Auswahl eines nächsten Tweets für die Labelpage zum Zeitpunkt des laufzeittechnischen Optimierungsbedarfs

Tweets in `@unlabeled` die Tabellen `entry` und `rating` zusammengeführt und alle IDs der Einträge dieses Datensets genommen, welche kein Rating besitzen. Dann wird davon `forbidden` abgezogen. `forbidden` enthält zu dem Zeitpunkt alle IDs der Einträge, deren Tweets von diesem User nicht gelabelt werden dürfen. Für die Relabel Tweets werden in `@labeled` alle IDs der Einträge dieses Datensets gespeichert, um danach die Eintrag IDs der noch nicht gelabelten Tweets aus `@unlabeled` und die IDs in `forbidden` abzuziehen.

Dieses Vorgehen ist jedoch, wie der Test im Nachhinein ergeben hat, eher ineffizient. Zum einen werden mehrere Queries getätigt. Des Weiteren werden erst alle IDs geladen, um dann davon einige wieder auszusortieren. Enthält die Datenbank viele Daten, sind die Ladezeiten entsprechend lang. Zudem ist das `ActiveRecord`-Interface meist langsamer als direkte SQL Abfragen.

Um dieses Problem zu beheben, sollten die benötigten Eintrag IDs auf effizienterem Weg geladen werden. Da die beiden Abfragen für die Laufzeit kritisch sind, sollten sie je in eine eigene Methode ausgelagert werden. Weil die Abfragen vom User abhängig sind, wurden diese in das User Model verlagert.

```
1 test "must return valid ids" do
2   # Anlegen des Datensets,
3   # der Tweets, Einträge, User und Labels
4
5   # Anlegen der Ratings
6   # tweet1 has one label from user1 - valid user2
7   Rating.create(entry_id: entries[0].id,
8                 label_id: label1.id, count: 1)
9   user1.entries << entries[0]
10  # tweet2 has two labels from user1 and user 2
11  Rating.create(entry_id: entries[1].id,
12               label_id: label1.id, count: 1)
13  Rating.create(entry_id: entries[1].id,
14               label_id: label2.id, count: 1)
15  user1.entries << entries[1]
16  user2.entries << entries[1]
17  # Anlegen weiterer Ratings
18  .
19  #Testen der Methoden
20  valid_initial_user1 =
21    User.get_valid_initial_ids(user1, dataset)
22  assert [entries[8].id, entries[9].id] -
23         valid_initial_user1 == []
24  assert valid_initial_user1.count == 2
25 end
```

**Listing 16:** Ausschnitt des Tests `must return valid ids` im Unit Test für das User Model

Um die kritischen Methoden geeignet zu entwickeln und zu testen, wurde zunächst der Unit Test für den User entsprechend um einen Test für die beiden Methoden erweitert. Ein Ausschnitt davon ist in Listing 16 zu sehen. Darin wurden zunächst mithilfe von Fixtures ein Datenset und zwei User angelegt. Außerdem einige Tweets und entsprechend viele Einträge, welche die Tweets dem Datenset zuordnen. Danach wurden, möglichst alle Fälle abdeckend, einige Ratings angelegt. Dann wurden entsprechend für die zwei User die neuen Methoden aufgerufen und die zurückgelieferten IDs mit den erwarteten Werten verglichen. Mit diesem Unit Test war die grundlegende Funktionalität der neuen

Methoden überprüfbar, mithilfe von Rails Time-Klasse konnten die Laufzeiten getestet werden.

```
1 def User.get_valid_relabel_ids(user, dataset)
2   user_entries = "(" + user.entries.pluck(:id).uniq.
3                 join(',') + ")"
4   sql = "SELECT e.id
5         FROM entries e
6         INNER JOIN ratings r
7         ON e.id = r.entry_id
8         INNER JOIN labels l
9         ON l.id = r.label_id
10        WHERE e.dataset_id = #{dataset.id}
11        AND e.id not in #{user_entries}
12        GROUP BY e.id
13        HAVING SUM(r.count) < #{dataset.crosslabel}
14        AND MAX(CASE WHEN l.garbage = 'f'
15                THEN 0 ELSE 1 END) = 0
16        LIMIT 50"
17   valid_relabel_ids = ActiveRecord::Base.connection.
18                       execute(sql).map{|row| row[0]}
19 end
20
21 def User.get_valid_initial_ids(user, dataset)
22   sql = "SELECT e.id
23         FROM entries e
24         LEFT JOIN ratings r ON e.id = r.entry_id
25         WHERE e.dataset_id = #{dataset.id}
26         AND r.entry_id is NULL
27         GROUP BY e.id
28         LIMIT 50"
29   valid_initial_ids = ActiveRecord::Base.connection.
30                       execute(sql).map{|row| row[0]}
31 end
```

**Listing 17:** Ausschnitt des Tests `must return valid ids` im Unit Test für das User Model

Die Methoden mit den optimierten, direkten SQL Abfragen sind in Listing 17 zu sehen. Schon alleine der Wechsel zur direkten SQL Abfrage beschleunigt den Prozess. Dazu kommt das Beschränken der Menge der IDs auf 50 und die Tatsache, dass nur tatsächlich erlaubte Einträge auch geladen werden. Die Vermutung lag nah, dass die Optimierung nun abgeschlossen war, doch die Ausgabe auf der Konsole zeigte immer noch eine hohe Zeitangabe, zu sehen in Abbildung 3.19.

Dass die Abfragen noch immer so langsam waren, ließ vermuten, dass ein weiteres Problem bestand. Bei weiterer Untersuchung stellte sich heraus, dass die neuen Queries keine Indizes verwendet haben, dadurch erhöhte sich die Laufzeit im Zusammenhang mit der Anzahl der Datenbankeinträge rapide.

Durch das geschickte Setzen von Indizes auf die verwendeten Tabellen (Listing 18) konnte die Laufzeit dann deutlich verringert werden, zu sehen in Abbildung 3.20. Die Laufzeit der Abfrage beträgt weniger als ein Zehntel der Laufzeit vor Setzen der Indizes, lediglich circa 200 Millisekunden. Durch diese Änderung den der Datenbank war die Problematik also behoben.

```

l, [2015-03-26T10:11:01.122436 #21610] INFO — : Processing by
PageController#next_tweet as JSON
l, [2015-03-26T10:11:01.122527 #21610] INFO — : Parameters:
{"dataset"=>"2", "label"=>"3", "tweet"=>"10101", "category"=>"null"}
l, [2015-03-26T10:11:03.002599 #21610] INFO — : Completed 200 OK in
1880ms (Views: 0.1ms | ActiveRecord: 1727.8ms)

```

**Figure 3.19:** Konsolenausgabe bei Laden des nächsten Tweets auf der Labelpage inklusive Ladezeit nach ersten Verbesserungsmaßnahmen.

```

1  add_index "entries", ["dataset_id"],
2     name: "index_entries_on_dataset_id"
3  add_index "entries_users", ["entry_id"],
4     name: "index_entries_users_on_entry_id"
5  add_index "entries_users", ["user_id"],
6     name: "index_entries_users_on_user_id"
7  add_index "ratings", ["entry_id"],
8     name: "index_ratings_on_entry_id"

```

**Listing 18:** Ausschnitt des Datenbankschemas der Applikation

```

l, [2015-03-30T09:00:18.984457 #18538] INFO — : Completed 200 OK in
226ms (Views: 0.1ms | ActiveRecord: 163.9ms)
l, [2015-03-30T09:00:23.827578 #18527] INFO — : Started POST "/nt" for
95.90.202.104 at 2015-03-30 09:00:23 +0200
l, [2015-03-30T09:00:23.881211 #18527] INFO — : Processing by
PageController#next_tweet as JSON
l, [2015-03-30T09:00:23.881289 #18527] INFO — : Parameters:
{"dataset"=>"2", "label"=>"3", "tweet"=>"4169", "category"=>"null"}

```

**Figure 3.20:** Konsolenausgabe bei Laden des nächsten Tweets auf der Labelpage inklusive Ladezeit nach setzen der Indizes.

### 3.7.2 Rückmeldung der Tester

Die allgemeine Rückmeldung der Tester war sehr positiv. Die Wirkung des Designs der Labelpage war wie erwünscht. Das Design wurde als sehr *"übersichtlich"* beschrieben. Die Tester finden es gut, dass die Farben der Labelpage einstellbar sind und waren insbesondere über die beiden Solarized Farbschemata erfreut. Ebenfalls wurde bemerkt, dass die Ausblendbarkeit der Anleitungen auf der Label Page positiv aufgefallen ist. Auch der Shortcutmodus wurde gelobt. Es wurde die Annahme bestätigt, dass der Extramodus den Labelprozess beschleunigt und weniger anstrengend gestaltet. Auch das Ranking ist sehr gut angekommen; es würde die Nutzer zum weiteren Labeln ansprechen.

Da die Tester nur Zugriff auf die Seite in der Rolle Labeler hatten, beschränkten sich die Rückmeldungen hauptsächlich auf die Labelseite. Da hier jedoch das Design der Seite besonders wichtig war, ist die positive Rückmeldung der Tester eine wichtige Bestätigung.



## 4 Reflexion, Zusammenfassung und Fazit

Mit der entstandenen Applikation wurden alle Anforderungen erfüllt. Alle gewünschten Funktionalitäten wurden umgesetzt und Ergänzungen wie zum Beispiel der Shortcutmodus wurden zusätzlich eingebaut. Häufige Rücksprache mit dem Endanwender hat sichergestellt, dass die Anwendung dessen Ansprüche in vollem Maße erfüllen kann.

Die Usability der Applikation ist entsprechend im Vordergrund geblieben. Dass diese in einem Maß vorhanden ist, das den Ansprüchen genügt, bestätigt der Test, dessen Rückmeldungen sehr positiv waren.

Zwar wurde die Applikation speziell für den Endanwender in der Arbeitsgruppe entwickelt, dennoch ist sie in vielen Punkten variabel gestaltet. Zudem ist es dank der sehr gut lesbaren Programmiersprache Ruby und der stark nach Konvention gehenden Architektur der Applikation relativ einfach, sich in diese bei Bedarf einzuarbeiten. Daher kann die Applikation auch in anderen Bereichen eingesetzt werden. Zwar ist die Applikation nicht für die Erstellung von Datensets mit etwas anderem als Tweets geeignet, doch vermutlich ließe sich auch dies durch die modulare Gestaltung der Applikation leicht anpassen. Die ausgewählten Methoden und Technologien sowie das Webapplication Framework Ruby on Rails haben den Gestaltungsprozess so einfach gestaltet wie möglich. Die während des Entwicklungszeitraumes gemachten Erfahrungen lassen mich den Aussagen der Entwickler von Ruby und Ruby on Rails über die positiven Effekte der Sprache und des Frameworks zustimmen.

Um die Qualität der entwickelten Applikation endgültig bewerten zu können, muss der Einsatz dieser in Zukunft beobachtet werden. Weitere Rückmeldungen der Nutzer werden mehr Auskunft über die Usability der Anwendung und die Qualität der erstellten Datensets geben. Auch die Ergebnisse der Verwendung der Datensets in der Forschung hinsichtlich der Stimmungsanalyse von Tweets werden die endgültige Qualitätsbewertung der Applikation unterstützen.

In Zukunft ist es möglich die Applikation dahingehend zu erweitern, dass für einen Tweet mehrere Stimmungen zugewiesen werden können. Dies kann nötig sein, wenn ein Tweet mehrere Entitäten enthält, deren Stimmungen unterschiedlich zu bewerten sind. In der jetzigen Applikation kann ein Tweet immer nur eine Stimmung in Form eines Labels zugewiesen bekommen. Um das Problem mit mehreren Entitäten in einem Tweet zu beheben, könnte eine Funktionalität eingebaut werden, die es erlaubt mehrere Entitäten im Tweet auszuwählen und konkret diesen Entitäten einzeln Labels zuzuweisen.

Die gewählten Technologien und Methoden haben den Entwicklungsprozess stark unterstützt, die entwickelte Applikation konnte allen an sie gestellten Anforderungen gerecht werden und der erfolgreiche Test lieferte ein direkt in der Forschung nutzbares Datenset.





## Literaturverzeichnis

- A. Alliance. Manifesto for agile software development, 2001. URL <http://agilemanifesto.org/>.
- J. Bollen, H. Mao, and X. Zeng. Twitter mood predicts the stock market. *Journal of Computational Science*, 2(1):1–8, 2011.
- D. Chelimsky, D. Astels, Z. Dennis, A. Hellesoy, B. Helmkamp, and D. North. *The RSpec Book: Behaviour-Driven Development with RSpec, Cucumber, and Friends*. The Pragmatic Programmers, LLC., 2010.
- N. Chen. Convention over configuration, 2006. URL <http://softwareengineering.vazexqi.com/files/pattern.html>.
- N. A. Diakopoulos and D. A. Shamma. Characterizing debate performance via aggregated twitter sentiment. In *Proceedings of the SIGCHI Conference on Human Factors in Computing Systems*, pages 1195–1199. ACM, 2010.
- M. Fowler. Gui architectures. URL <http://martinfowler.com/eaDev/uiArchs.html>.
- J. Han, M. Kamber, and J. Pei. *Data mining: concepts and techniques*. Morgan Kaufmann, 2006.
- A. Hunt and D. Thomas. *The Pragmatic Programmer*. Addison Wesley, 1999.
- T. Joachims. *Learning to classify text using support vector machines: methods, theory and algorithms*. Kluwer Academic Publishers, 2002.
- R. Osherove. *The Art of Unit Testing, 2nd Edition*. Manning, 2014.
- S. Ruby, D. Thomas, and D. Heinemeier Hansson. *Agile Web Development with Rails 4*. The Pragmatic Programmers, 2013.
- H. Saif, Y. He, and H. Alani. Alleviating data sparsity for twitter sentiment analysis. In *The 2nd Workshop on Making Sense of Microposts*. 2012.
- D. Thomas, C. Fowler, and A. Hunt. *Programming Ruby*. The Pragmatic Programmers, 2013.







## Erklärung

Ich versichere, dass ich die eingereichte Bachelor-Arbeit selbstständig und ohne unerlaubte Hilfe verfasst habe. Anderer als der von mir angegebenen Hilfsmittel und Schriften habe ich mich nicht bedient. Alle wörtlich oder sinngemäß den Schriften anderer Autoren entnommenen Stellen habe ich kenntlich gemacht.

Osnabrück, den 13. April 2015

(Miriam Beutel)