

# Datenbankpraktikum 25.07.11 - 12.08.11

Nils Haldenwang, Julian Kniephoff, Oliver Vornberger  
Universität Osnabrück

Die Teilnehmer erstellen in 2-er bis 4-er Teams mit Hilfe der in der Vorlesung Datenbanksysteme vorgestellten Werkzeuge (unter anderem auch Ruby on Rails) eine webgestützte Datenbankapplikation.

Als Thema soll eine Mitfahrzentrale gewählt werden, d.h. Anfragen von Mitfahrern und Angebote von Fahrern müssen bestmöglich einander zugeordnet werden. Die Verwendung von Karten soll die Ein- und Ausgabe unterstützen. Ggf. kommen auch Smartphones zum Einsatz.

Zum Abschluß des Praktikums werden die Arbeiten präsentiert und eine schriftliche Ausarbeitung unter Verwendung von media2mult angefertigt.



## Inhaltsverzeichnis

<b>1. Backend.....</b>	<b>3</b>
1.1 Modellierung.....	3
1.2 Funktionalitäten.....	6
1.3 Trips bzw Requests erstellen.....	9
1.4 Suchfunktion.....	11
1.5 Tests.....	14
1.6 Fazit.....	19
<b>2. Controller.....</b>	<b>20</b>
2.1 Ausarbeitung.....	20
2.1.1 Aufgaben des Controllers.....	20
2.1.2 Workflow.....	21
2.1.3 Zielsetzung.....	21
2.1.4 Umsetzung.....	22
2.1.5 Fazit.....	27
<b>3. Frontend.....</b>	<b>28</b>

---

3.1 Umsetzung.....	28
3.1.1 Layout.....	28
3.1.2 CSS.....	29
3.1.3 Formulare.....	31
3.1.4 Javascript.....	32
3.1.5 Paperclip.....	35
3.1.6 Photoshop.....	35
3.1.7 GoogleMaps.....	37
3.2 Mitfahrzentrale 2.0.....	38
3.2.1 Startseite.....	38
3.2.2 Suche.....	39
3.2.3 Fahrten.....	40
3.2.4 Nachrichten.....	44
3.2.5 Bewertungen.....	44
3.2.6 Fahrzeuge.....	44
3.2.7 Profil.....	45

# 1. Backend

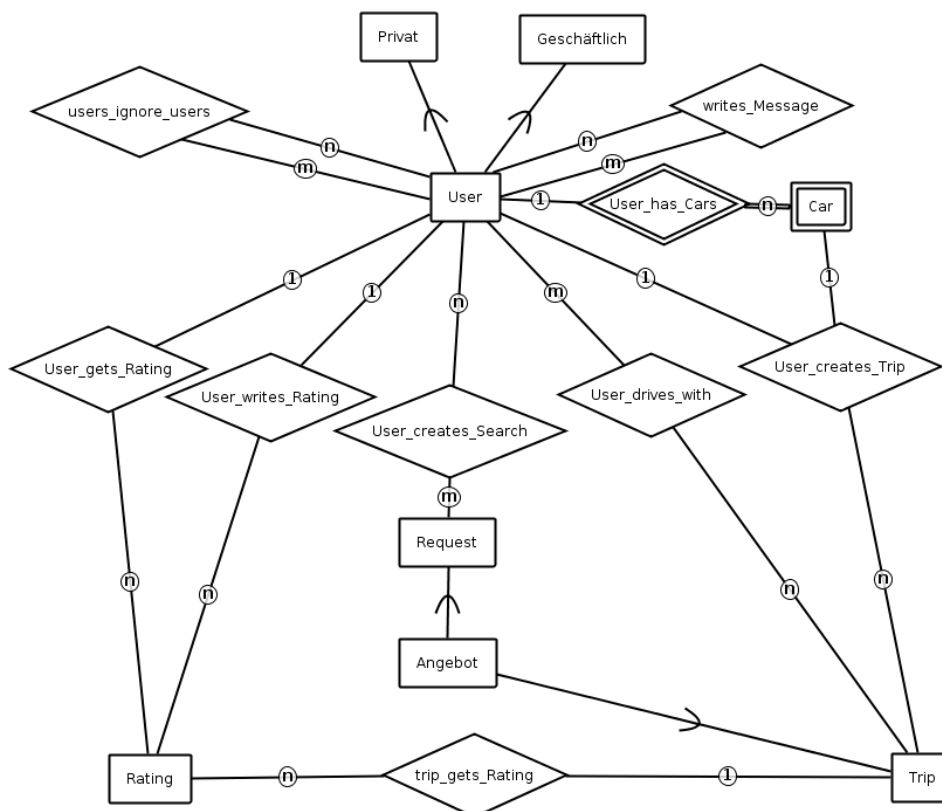
Dominik Krümpelmann, Sebastian Brockmeyer, Michael Blömer, Christoph Eichler

Die Backendgruppe stellt die untere Kommunikationsschicht mit der Datenbank dar, insofern bestand unsere Aufgabe darin, die konzeptuelle Modellierung der benötigten Features einzurichten, sowie nötige Beziehungen zwischen den Entitäten herzustellen und die Integrität der Datenbank zu sichern. Desweiteren war es an uns, mögliche Interaktionen mit der Datenbasis für die anderen Gruppen zu ermöglichen, d.h. Methoden zu implementieren, die als Schnittstellen fungieren. Dafür war eine intensive Kommunikation mit den anderen Gruppen von Nöten, um die Datenbank an die gewünschten Funktionalitäten anzupassen.

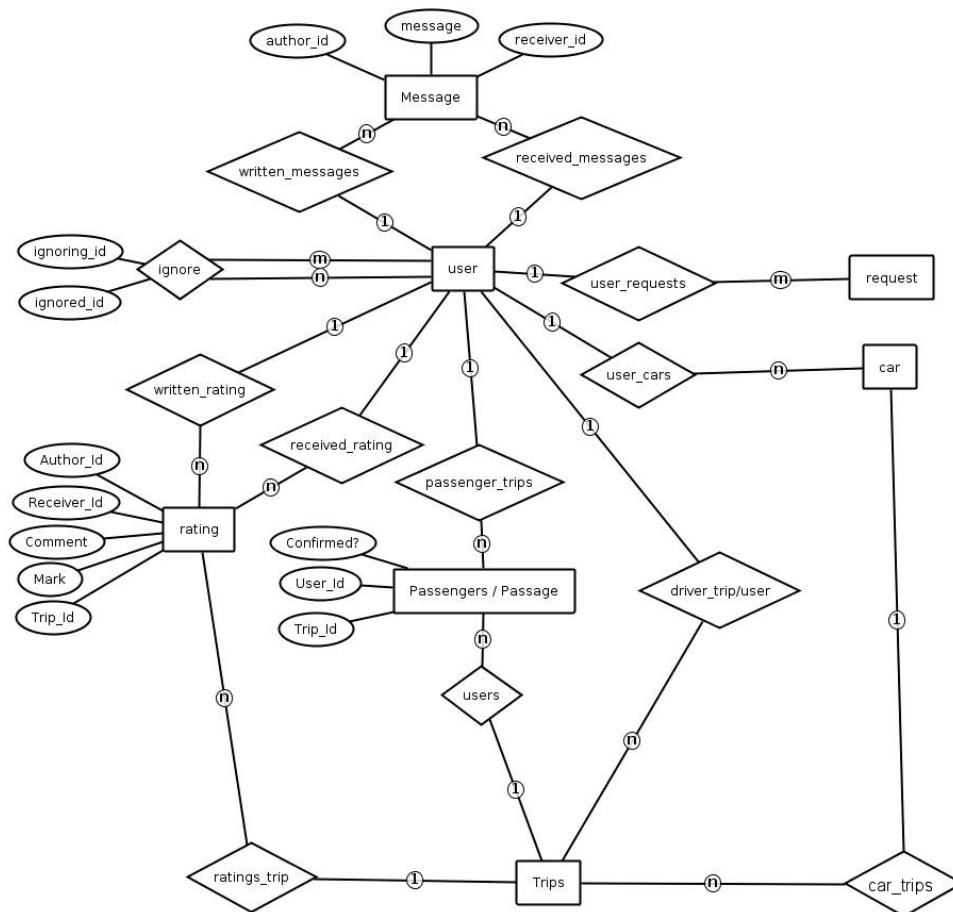
## 1.1 Modellierung

### ER-Diagramm und Grundgedanken

Zu Beginn unserer Arbeit erstellten wir ein "konventionelles", nicht an Rails angepasstes ER-Diagramm. Attribute wurden der Übersichtlichkeit halber weggelassen.



Im weiteren Verlauf zeigte sich jedoch, dass Features wie Generalisierung, schwache Entitäten und n:m-Beziehungen mit Attributen sich schlecht bzw. gar nicht in Rails umsetzen lassen. Z.B. mussten wir die ursprüngliche Beziehung *writes\_message* in eine Entität umwandeln, um ein Text-Attribut anhängen zu können. Vorteil bei dieser Transformation war, dass die Beziehungen transparenter wurden, d.h. der zu schreibende Rubycode lässt sich nahezu 1:1 aus dem Diagramm ablesen.



## Umsetzung in Ruby on Rails

### Migrations

Migrations stellen in Rails ein besonderes Feature dar, das gewährleistet, dass alle Mitentwickler und Anwender durch Ausführen des Befehls `rake db:migrate` das gleiche Datenbankschema erstellen, wie von uns erdacht. Gerade in unserem Fall unter Benutzung von *SQLite3* stellt dies einen enormen Vorteil dar, da es keine zentrale Datenbank gibt. Rails erkennt über die Versionsnummern, die am Anfang einer jeden Migration stehen, die chronologisch korrekte Ausführung. Dabei werden die *self.up*-Methoden einer Migration aufgerufen, wodurch Rails die angegebenen Tabellen über automatisch generierte SQL-Statements in die Datenbank schreibt. Die *self.down*-Methoden invertieren diesen Vorgang. Dadurch ist es möglich, mithilfe des Befehls `rake db:migrate VERSION=x` die Datenbank auf einen als funktionierend bekannten Stand *x* zurückzusetzen.

```

1 class AddDobToUsers < ActiveRecord::Migration
2   def self.up
3     add_column :users, :birthday, :date
4     remove_column :users, :age
5   end
6   def self.down
7     remove_column :users, :birthday
8     add_column :users, :age, :integer
9   end
10 end

```

## Beziehungen

Das Modellieren von Beziehungen gestaltet sich in Rails, solange man sich an die Konventionen hält, als äußerst komfortabel. Leider gab es viele Situationen, in denen wir diesen Vorteil nicht nutzen konnten. Dies war beispielsweise bei User und Message der Fall, da zwei Beziehungen zwischen diesen Entitäten bestehen, um zwischen geschriebenen und erhaltenen Nachrichten zu unterscheiden. Anhand dieser Beziehung zeigen wir exemplarisch, wie wir dieses Problem gelöst haben:

```

223 has_many :received_messages, :class_name => "Message",
224         :foreign_key => "receiver_id", :dependent => :destroy
225
226 has_many :written_messages, :class_name => "Message",
227         :foreign_key => "writer_id", :dependent => :nullify
228

```

Das Keyword *has\_many* gibt an, dass es sich um eine 1:n-Beziehung handelt. An zweiter Stelle steht der Name der Beziehung, dann folgt der Name der Entität, auf die referenziert wird, mit *foreign\_key* wird der Fremdschlüssel angegeben und an letzte Stelle wird die dynamische Integrität gesichert. In diesem Fall werden beim Löschen eines Users also seine erhaltenen Nachrichten gelöscht, bei seinen geschriebenen wird die Referenz nur auf *nil* gesetzt, damit diese für den Empfänger noch lesbar bleiben. Ist *u* ein User, so kann man nun mit *u.received\_messages* und *u.written\_messages* direkt auf seine erhaltenen bzw. geschriebenen Nachrichten zugreifen.

## Integrität

Wie oben schon gezeigt, gibt es in Ruby on Rails einfache Mittel, dynamische Integrität sicherzustellen. *:dependent => :destroy* kommt dabei dem SQL-Statement *on delete cascade* gleich, *:dependent => :update* entspricht *on update cascade* und *:dependent => :nullify* wirkt wie *on delete set null*. Bei der statischen Integrität gibt es ebenfalls von Rails gegebene Methoden, die die angesprochenen Attributfelder auf Vorhandensein, bestimmte Wertebereiche, Einzigartigkeit und vieles mehr verifizieren. Möchte man spezifischere Benutzerprobleme abfangen (bspw. darf ein Mitfahrer nicht gleichzeitig Fahrer sein), müssen dafür eigene Validation-Methoden implementiert werden. Diese werden beim Erstellen eines Objekts initial aufgerufen, um ggf. eine falsche Eingabe abzufangen.

```

validate :confirmed_not_nil, :not_his_own_driver, :
       :no_double_insert

```

```

def not_his_own_driver
  if self.trip.user == self.user
    errors.add(:field, 'driver '+ user.id.to_s + ' must not
  be his own passenger')
  end
end

```

## 1.2 Funktionalitäten

Wie im ER-Diagramm ersichtlich, sind für die Implementierung unserer Applikation zahlreiche Entitäten notwendig. Im folgenden möchten wir etwas genauer auf die verschiedenen Entitäten, deren Bedeutung und Funktionalitäten eingehen.

### Sichtbarkeiten

Der User stellt das Zentrale Modell unserer Applikation dar. Für jeden neu registrierten Nutzer wird also ein Userobjekt angelegt, an dem zahlreiche Informationen gespeichert werden. So sind, direkt in der User Tabelle, Informationen über dessen Adresse, Geburtsdatum, Beitrittsdatum und Login gespeichert. Weiterhin verfügt das Usermodel über boolesche Datenfelder, mit denen der jeweilige Nutzer selbst die Sichtbarkeit seiner Daten für Dritte festlegen kann. Das Sicherstellen korrekter Eingaben wird dabei von Validations übernommen. Es ist z.B. nicht möglich in die Spalte "Postleitzahl" Buchstaben einzugeben. Weiterhin kann ein User das Profil eines anderen jedoch nur dann einsehen, wenn er mit ihm über einen Trip in Verbindung steht. Sei x ein User, so wird im Programm zuerst ermittelt, mit welchen Usern x in Verbindung steht. Dies geschieht mit Hilfe der Methode `get_visible_users`, die alle angezeigten User festlegt. Wollte x sich nun deren Profil anzeigen lassen, würden ihm alle die Datenfelder angezeigt, deren `visibility_Datenfeld` auf true steht.

**Profil bearbeiten**

Mit \* gekennzeichnete Felder müssen ausgefüllt sein

Name*	<input type="text"/>	
E-Mail*	<input type="text"/>	
Neues Passwort	<input type="password"/>	
Neues Passwort wiederholen	<input type="password"/>	
Geburtsdatum*	11   8   2011	sichtbar <input type="checkbox"/>
Geschlecht	Männlich	
Strasse und Hausnummer*	<input type="text"/>	sichtbar <input type="checkbox"/>
PLZ*	<input type="text"/>	sichtbar <input type="checkbox"/>
Ort*	<input type="text"/>	sichtbar <input type="checkbox"/>
Telefon	<input type="text"/>	sichtbar <input type="checkbox"/>
InstantMessengerDaten	<input type="text"/>	sichtbar <input type="checkbox"/>
Bild hochladen	<input type="text"/> <input type="button" value="Durchsuchen"/>	
Passwort*	<input type="password"/>	

Sichtbarkeitseinstellungen in der View

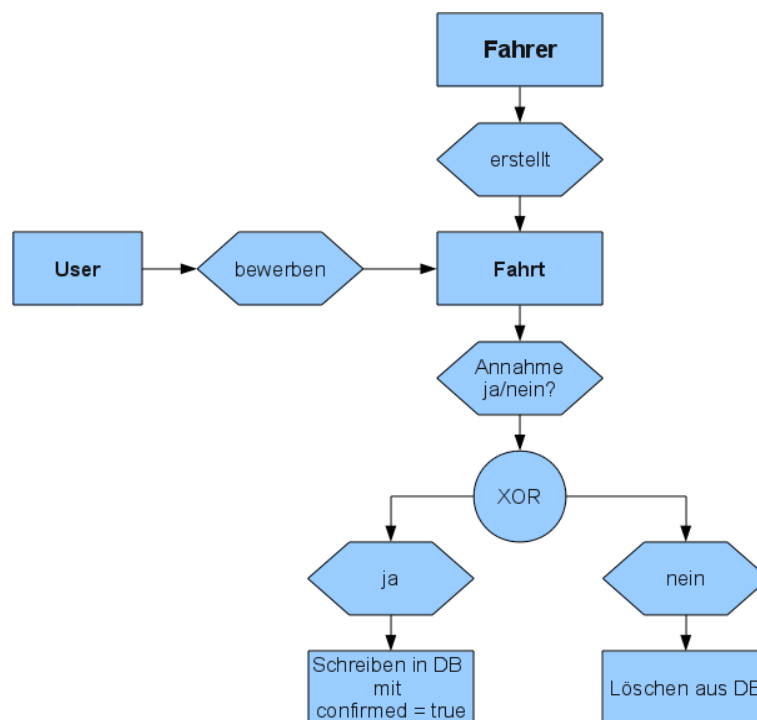
### Nachrichten

Weiterhin können Nutzer anderen Nutzern Nachrichten schreiben. Dies läuft über die Message-Entität ab. Diese besitzt eine Spalte für die ID des Authors, des Empfängers und ein Textfeld für die Nachricht selbst. Es wurden einige Methoden in die Userklasse implementiert um eine formatierte Ausgabe in der View zu gewährleisten. Ein Nutzer kann sich z. B. seine geschriebenen, sowie die empfangenen Nachrichten anzeigen lassen. Nachrichten die er bereits gelöscht hat, werden für ihn nicht mehr angezeigt, bleiben jedoch in der Datenbank bestehen, damit der jeweilige Counterpart sie trotzdem einsehen kann. Erst wenn die Nachricht von beiden Seiten als gelöscht markiert worden ist, wird sie aus der Datenbank entfernt. Zusätzlich wurde eine Methode implementiert, die die Anzahl neuer Nachrichten angibt.

## Mitfahren

Möchte ein Nutzer bei einer Fahrt mitfahren, muss er sich zuerst bewerben. Seine Id wird dann mithilfe der Methode *bewerben*, der man einen Trip übergibt, in die Passenger Entität mit der dazu gehörigen Trip Id geschrieben. Die Passenger Entität stellt also eine Jointable zwischen Mitfahrern und Fahrten dar. Diese n:m Relation wurde in eine eigene Entität mit zwei 1:n Beziehungen aufgelöst, da wir zusätzlich zu der eigentlichen Relation ein boolsches Datenfeld brauchen, in dem vermerkt ist, ob ein Nutzer vom Fahrer des Trips angenommen ist, oder noch auf Annahme wartet. Wenn er sich auf eine Fahrt beworben hat, wird systemintern über den Controller eine Nachricht an den jeweiligen Fahrer der entsprechenden Fahrt verschickt. Der Fahrer hat nun zwei Möglichkeiten:

- Er nimmt den Bewerber an, d.h. in der Passenger Entität wird das *confirmed*-Datenfeld über die Methode *accept* an entsprechender Stelle auf *true* gesetzt
- Er lehnt den Bewerber ab, d.h. der vollständige Eintrag wird mithilfe der Methode *delete* aus der Passenger Relation gelöscht.



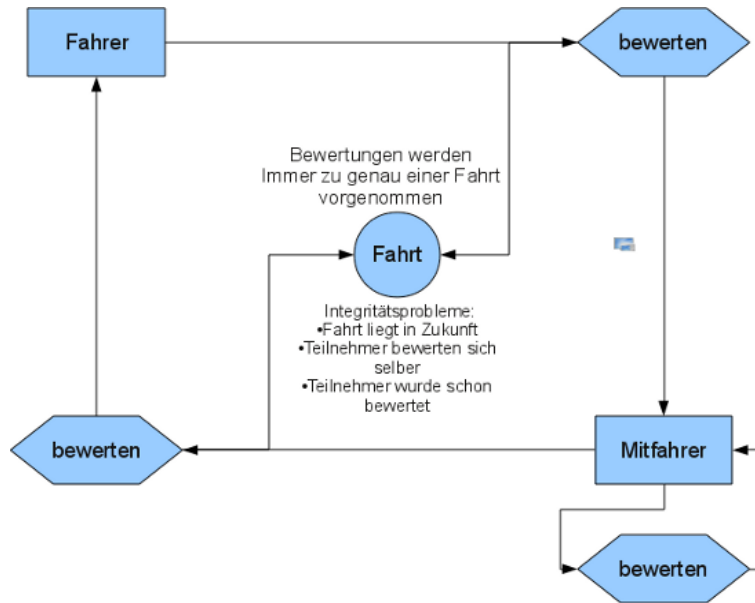
## Fahren

Jeder Nutzer kann Fahrten erstellen und so, für andere sichtbar, als Anbieter in der Applikation erscheinen. Tut er dies, so wird über eine von uns implementierte Methode die Routendistanz und Routendauer berechnet (s. Punkt 3). Nun steht dieser vom Nutzer erstellte Trip in der Datenbank und kann bei Anfragen angezeigt werden. Voraussetzung für den Fahrer ist lediglich, dass er ein Fahrzeug angelegt hat.

## Bewerten

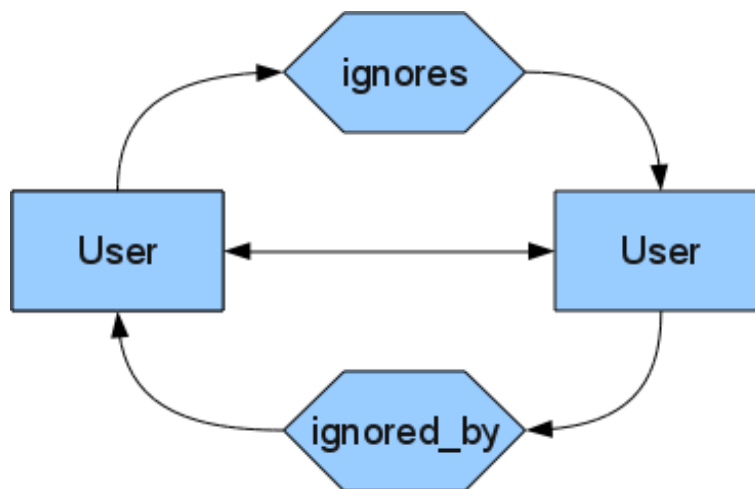
Während der Planungsphase hatte unsere Gruppe die Idee, dass eine Bewertungsfunktion ähnlich der großer führender Internetversandhäuser, gegeben sein müsste. Es sollte möglich sein als Mitfahrer andere Mitfahrer und den Fahrer zur entsprechenden Fahrt zu bewerten und auch umgekehrt, als Fahrer seine Mitfahrer zu bewerten. Dazu wurde in Rails zunächst eine Rating Entität erstellt, die über die Primärschlüssel *Trip\_Id*, *Author\_Id* und *Receiver\_Id* verfügt. D.h. die ursprünglich ternäre n:m-Relation wurde in drei 1:n-Beziehungen aufgelöst. Somit können bei einer Fahrt an der 5 Personen teilgenommen haben (inclusive Fahrer) bis zu 20 Ratings erstellt werden. Der Nutzer selber kann sich anhand der Spalten *Author\_Id* und *Receiver\_Id*, die für

ihn als Fremdschlüssel fungieren, seine jeweils geschriebenen und empfangenen Bewertungen anzeigen lassen. Durch die Methode *get\_waiting\_ratings* können in der View weiterhin alle Fahrten mit ihren Nutzern ausgegeben werden, für die der jeweils aktuelle Nutzer noch keine Bewertung geschrieben hat. Aus der Anzahl aller erhaltenen Bewertungen berechnen wir einen Durchschnittswert, der dem Nutzer als Mitfahrer-, bzw. Fahrerrating ausgegeben wird. D.h. die erhaltenen und auch geschriebenen Bewertungen werden strikt nach der Rolle (Fahrer oder Mitfahrer) des Nutzers im jeweiligen Trip getrennt. Die Differenzierung zwischen Nutzern, die Fahrer in einem Trip waren und nachträglich nur Mitfahrer bewerten, von diesen aber als Fahrer bewertet werden und Nutzern, die Mitfahrer waren und nachträglich sowohl den Fahrer als auch weitere Passagiere bewerten und von diesen als Mitfahrer bewertet werden, erreichen wir durch die Methoden *get\_own\_driver\_ratings* und *get\_own\_passenger\_ratings*. Erst nachdem ein Trip in die Datenbank gespeichert wurde, wird durch diese Methoden die Unterscheidung zwischen fahrer- und mitfahrerspezifischen Bewertungen vorgenommen.



## Ignorieren

Zusätzlich können Nutzer andere Nutzer ignorieren. D.h. dass der Ignorierte aus der Sicht des Ignorierenden komplett aus der Datenbank "entfernt" wird. Dieser Eindruck entsteht in der Implementierung durch das Verändern von Sichtbarkeiten. Fahrten ignorierte Nutzer werden nicht angezeigt, Ignorierte können sich nicht auf eigens erstellte Fahrten bewerben, dem Ignorierten keine Nachrichten schreiben und nicht sein Profil einsehen.



Die Umsetzung dieses Features erfolgte in Rails durch eine Selbstreferenzierende n:m-Beziehung. Ein User reverenziert über die Beziehungen *ignoring* und *ignored* durch die Join-Tabelle *ignore* auf einen anderen. Durch



die jeweiligen Beziehungen ist dabei eindeutig, ob ein User einen anderen ignoriert oder von diesem ignoriert wird. Die Umsetzung in Rails gestaltete sich als schwierig, da wir, um eine solche Beziehung umzusetzen, eine Konvention von Rails brechen mussten, was dazu führte, dass wir die benötigten Keywords von Hand hinzufügen mussten.

```
has_and_belongs_to_many :ignorings, :class_name => "User", :join_table => "ignore", :foreign_key =>
  "ignored_id", :association_foreign_key => "ignoring_id"
has_and_belongs_to_many :ignoreds, :class_name => "User", :join_table => "ignore", :foreign_key =>
  "ignoring_id", :association_foreign_key => "ignored_id"
```

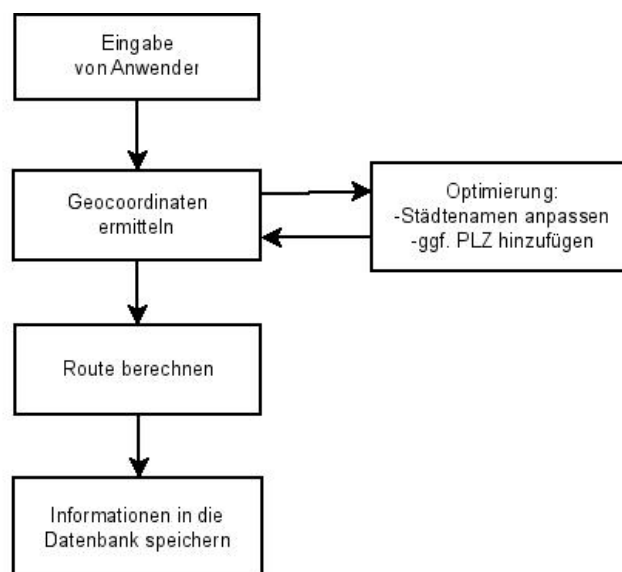
`:ignorings` bzw. `:ignoreds` gibt den Namen der Beziehung an, `:class_name` die Entität auf die referenziert wird, `:join_table` die Tabelle in der Fremdschlüssel und eigene Id in Beziehung gesetzt werden, `:foreign_key`, also der Fremdschlüssel, gibt die ID des Objektes an, dass (am Beispiel von `ignorings`) ignoriert wird und `:association_foreign_key` legt fest, in welcher Spalte die ID des ignorierenden Nutzers steht.

Anhand dieser Beziehung ist es dann z.B. auch möglich alle, den User ignorierenden User auszugeben und diese zu zählen, da dieser Wert später in der Bewertungsfunktion unter Punkt 4 noch benötigt wird.

## 1.3 Trips bzw Requests erstellen

Eine der Hauptfunktionalitäten unserer Mitfahrzentrale besteht darin, eine Fahrt oder ein Fahrtgesuch zu erstellen. Dies wurde durch eine Trip- bzw Request-Entität verwirklicht, die unter Anderem die Datenfelder PLZ, Ort und Straße für Start- und Zieladresse beinhalten. Bevor ein Trip erstellt wird muss das Model also prüfen, ob alle benötigten Daten zugrundeliegen, die für spätere Funktionalitäten verlangt werden.

Das folgende Schaubild zeigt den Ablauf, der eingehalten werden muss, damit ein Request, bzw. ein Trip erstellt werden kann:



Im folgenden wird der Ablauf zur Verdeutlichung beispielsweise erläutert:

### Die vom Nutzer getätigte Eingabe wird geocodiert.

```
temp = Geocoder.coordinates(params[:address_start])
if temp != nil
  @trip.starts_at_N = temp[0]
  @trip.starts_at_E = temp[1]
```

Hierbei kann es vorkommen, dass ein Nutzer nur eine Postleitzahl oder eine Stadt ohne dazugehörigen Straßennamen und Hausnummer eingibt. Durch das Kontaktieren der Controller von Google Maps mithilfe des Programms Geocoder wird die Eingabe automatisch auf eine eindeutige Adresse vervollständigt. D.h. Geocoder sucht für eine nicht eindeutige Eingabe, wie z.B. *Köln* (ohne Straßennamen und PLZ) die zentrumsnahen Koordinaten, welche dann in die dazu gehörigen Datenfelder geschrieben werden.

## Optimierung der Eingabe.

Da es möglich sein sollte, dem Anwender eine vielfältige Eingabe zu ermöglichen, lag es an uns, die restlichen Spalten der Entität zu ergänzen, vervollständigen oder die Eingabe zu korrigieren. Dazu mussten wir ein sogenanntes *reverse geocoding* vornehmen, dass uns zu den im Vorfeld ermittelten Koordinaten alle Informationen liefert, die genau dieser Punkt enthält.

Im folgenden Beispiel wird die Variable *start\_a* mit allen Informationen gefüllt, die uns Gmaps4rails für die Startkoordinaten *starts\_at\_N* (Längengrad) und *starts\_at\_E* (Breitengrad) liefert.

```
def set_address_info
  start_a = Gmaps4rails.geocode(self.starts_at_N.to_s + " " +
    self.starts_at_E.to_s + " ", "de")[0][:full_data]
```

Für die Koordinaten von beispielsweise Hörstel (52.30012439999 Nord und 7.58706790000 Ost) ergäbe sich dann folgende Ausgabe.

Danach sind wir mit entsprechender Hash- und Array-Notation an die gewünschten Informationen gekommen und haben diese in die dafür vorgesehenen Datenfelder (*zipcode*, *city*, *street*) gespeichert. Dies gestaltete sich jedoch als äußerst schwierig, da in der Google Maps API nicht angegeben war, wie die jeweiligen Hashes/Arrays formatiert sind. Letztendlich bot sich uns ein stark verschachteltes "Konstrukt" aus Hashes und Arrays: *self.zipcode = start\_a[0][:full\_data]["address\_components"][6]["long\_name"]*

```
Ruby-1.9.2-p290 :002 > Gmaps4rails.geocode("52.30012439999 7.58706790000")
=> [{"lat"=>52.2999464, "lng"=>7.5872163, "matched_address"=>"Bahnhofstraße 1, 48477 Hörstel, Ger
99, "lng"=>7.5873667, "southwest"=>{"lat"=>52.2997732, "lng"=>7.587067900000001}}, {"full_data"
me"=>"1", "types"=>["street_number"]}, {"long_name"=>"Bahnhofstraße", "short_name"=>"Bahnhofstr
short_name"=>"Hörstel", "types"=>["locality", "political"]}, {"long_name"=>"Steinfurt", "short_
political"}}, {"long_name"=>"Nordrhein-Westfalen", "short_name"=>"Nordrhein-Westfalen", "type
long_name"=>"Germany", "short_name"=>"DE", "types"=>["country", "political"]}, {"long_name"=>"4
"}}, {"formatted_address"=>"Bahnhofstraße 1, 48477 Hörstel, Germany", "geometry"=>{"bounds"=>{"n
7)}, {"southwest"=>{"lat"=>52.2997732, "lng"=>7.587067900000001}}, {"location"=>{"lat"=>52.2999464
ATED", "viewport"=>{"northeast"=>{"lat"=>52.3012977802915, "lng"=>7.588566280291501}, "southwes
7)}}, {"types"=>["street_address"]}]}
```

Des öfteren standen weiterhin gesuchte Informationen im Array an verschiedenen Stellen, sprich es kann vorkommen, dass bei größeren Städten die Postleitzahl beim Hash *address\_components* nicht im Array an Stelle 6, sondern z.B. an Stelle 7 zu finden ist. Gelöst haben wir dieses Problem, durch eine einfache, von Ruby gegebene Methode. So mussten wir jedes Array im Hash mit der Methode *include?* darauf untersuchen, an welcher Stelle jeweilige Information steht.

```
start_a["address_components"].each do |i|
  if i["types"].include?("postal_code")
    self.start_zipcode = i["long_name"]
  end
```

## Berechnung der Route

Nachdem wir alle Informationen erhalten haben, wird die Routendistanz und -dauer berechnet. Dazu wird vom Controller die Methode *set\_route* ausgerufen, in der wiederum anfänglich Google Maps kontaktiert wurde, um die Route zwischen Startpunkt und Endpunkt zu berechnen. Im weiteren Verlauf stellte sich jedoch heraus, dass Google Maps mit nur 2500 Routenberechnungsanfragen am Tag zu wenig Kapazität bietet, so dass wir in der *set\_route* Methode nun Bing-Maps verwenden.

```
def set_route
  route = Bing::Route.find(:waypoints => [self.starts_at_N.to_s + " " + self.starts_at_E.
  to_s + " ", self.ends_at_N.to_s + " " + self.ends_at_E.to_s + " "])[0]
  self.distance = route.total_distance
  self.duration = route.total_duration
end
```

## Speicherung in die Datenbank

Erst jetzt, wenn alle Datenfelder ermittelt wurden, kann der Trip, bzw. Request gespeichert werden. Falls Datenfelder nicht gesetzt sind greifen mehrere Validations und der Trip kann nicht erstellt werden.

```
def set_route
  route = Bing::Route.find(:waypoints => [self.starts_at_N.to_s + " " + self.starts_at_E.
  to_s + " ", self.ends_at_N.to_s + " " + self.ends_at_E.to_s + " "])[0]
  self.distance = route.total_distance
  self.duration = route.total_duration
end
```

Folgende Probleme können dazu führen, dass ein Trip nicht gespeichert werden kann:

- Die Eingabe vom Benutzer kann nicht geocodiert werden! (Beispiel: *Eingabe* von: "hallo" nach: "98765")
  - Validation: die Koordinatenfelder dürfen nicht Null sein
- Die Route kann nicht berechnet werden! (Beispiel: Von Nordamerika kann keine Route nach Europa berechnet werden.)
  - Bing (*Gem für Routenberechnung*) wirft eine Exception, die an Controller weitergeleitet wird, Trip wird verworfen.
- Die Eingabe vom Benutzer ist nicht eindeutig! (Beispiel: *Eingabe* von: "Ham" nach: "Ber")
  - In diesem Fall kann es vorkommen, dass die Eingabe geocodiert, als auch die Route berechnet wird, es liegt also im Nachhinein am Anwender zu entscheiden, ob er diese Route fahren will oder nicht.
- Der Startzeitpunkt des Trips liegt in der Vergangenheit.
- Der Benutzer möchte einen Trip erstellen, in dem er allerdings keine freien Sitzplätze zur Verfügung hat.

## 1.4 Suchfunktion

Zu unseren Aufgaben als Model gehörte außerdem, eine Optimierungsfunktion zu implementieren, die Fahrtgesuche (Requests) und Fahrten (Trips) zueinander in Beziehung bringt und auf Übereinstimmung überprüft. Dabei sollen nicht nur Trips und Requests zusammengeführt werden, die identisch sind, sondern auch solche, die sich nur ähnlich sind, also z.B. nicht exakt denselben Zielort haben. Die Herausforderung besteht hierbei darin, diese "Ähnlichkeit" anhand gewisser Kriterien zu bewerten und die Trips bzw. Requests dann nach dieser Bewertung sortiert auszugeben.

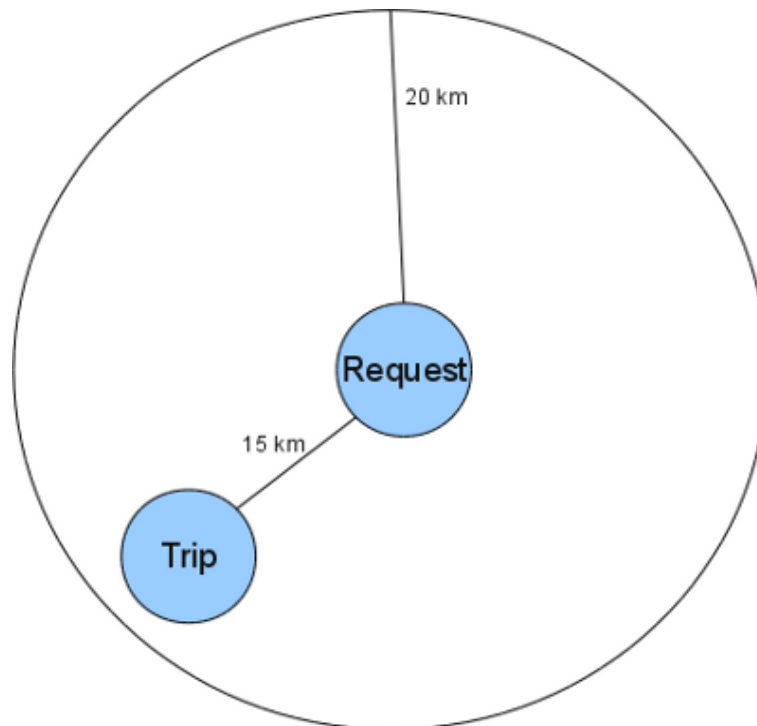
## Ablauf einer Suche

Wenn ein User eine Anfrage (Request) stellt, werden folgende Schritte abgearbeitet:

1. Grobe Selektion:
  1. Nach Start-, Ziel-, Zeitpunkt
  2. Nach Gepäck, Ignorierung
2. Bewertung der Selektion:
  1. Nach Umweg, Verzögerung
  2. Bewertung des Fahrers
  3. Anzahl Ignorierungen des Fahrers
3. Sortierung nach Bewertung

## Grobe Selektion

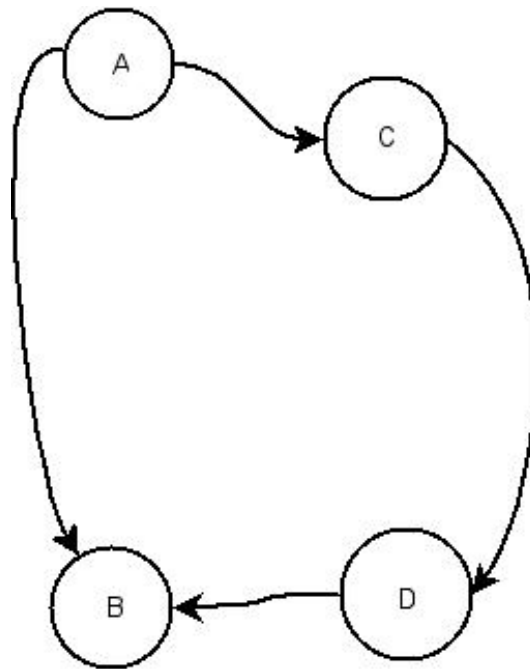
Sowohl die Start- als auch die Zieladresse sollen in einem vom Requeststeller vorher festgelegten Umkreis liegen und der Startzeitpunkt in einem vorher ausgewählten Intervall. Weiterhin wird der Wunsch nach Gepäckmitnahme berücksichtigt und Fahrten ignoriertes Benutzer werden nicht angezeigt.



Für die Berechnung der Abstände der Start- und Zieladressen verwenden wir die Methode *Calculation.distance\_between* des *Geocoders*. Diese Methode bekommt zwei Längen- und Breitengrade übergeben und errechnet den Abstand der beiden Koordinaten auf der Erdoberfläche.

## Bewertung

Ist die grobe Auswahl erfolgt, werden alle Fahrten, die diese Bedingungen erfüllen näher betrachtet. Unsere Grundidee soll dabei durch folgendes Diagramm veranschaulicht werden:



Wenn es darum geht, einen Trip, der von A nach B führt, bezüglich eines Requests von C nach D zu bewerten, dann wird die Route ACDB berechnet und mit der eigentlichen Route AB des Fahrers verglichen. Wir berechnen also stets den Umweg, den der Fahrer in Kauf nehmen müsste, wenn er den Ersteller dieses Requests mitnehmen will. An dieser Stelle soll betont werden, dass keineswegs vom Fahrer verlangt wird, diesen Umweg tatsächlich zu fahren. Es wird ihm auch nicht angeboten dies zu tun, etwa indem eine Route mit diesem Umweg angezeigt wird. Die Berechnung des Umwegs dient einzig und allein uns als Kriterium für die "Ähnlichkeit" von Trip und Request.

An dieser Stelle wird auch klar, warum wir vor der Bewertung zwangsläufig erst die Auswahl einschränken mussten. Es wird für jeden einzelnen Trip eine Umwegsberechnung mit GoogleMaps oder Bing/Maps durchgeführt. Bereits ab acht Trips liegt die Anfragedauer bei mehr als drei Sekunden, so dass es unzumutbar wäre, alle Trips in der Datenbank in die Berechnung einzubeziehen.

## An folgendem Beispiel soll die Berechnung der Bewertung eines Trips demonstriert werden:

Ein User hat einen Trip von Melle zum Flughafen Köln-Bonn angelegt. In der Datenbank werden die Länge und die Fahrtzeit des Trips gespeichert.

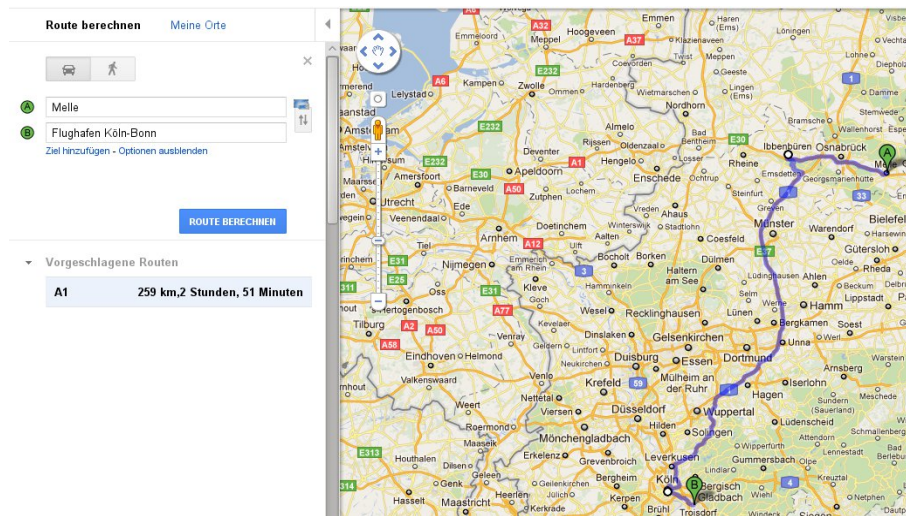
Ziel hinzufügen - Optionen ausblenden

ROUTE BERECHNEN

Vorgeschlagene Routen

A1	227 km, 2 Stunden, 8 Minuten
A43 und A1	234 km, 2 Stunden, 15 Minuten
A3	239 km, 2 Stunden, 17 Minuten

Wenn jetzt ein andere User einen Request von Ibbenbüren nach Köln anlegt, dann wird im Hintergrund folgende Route mit den Wegpunkten Ibbenbüren und Köln berechnet, aber nicht angezeigt:



Nun wird der "reine" Umweg berechnet, also die neue Distanz abzüglich der ursprünglichen:  $Weg\_Diff = 259 - 227 = 32$ . Dieser Wert wird dann durch die ursprüngliche Distanz geteilt und so der relative Umweg berechnet:  $Weg\_rel = 32 / 227 = 0,141$ . Das gleich geschieht mit der Fahrzeit:  $Zeit\_Diff = 171 - 128 = 43$  und  $Zeit\_rel = 43 / 128 = 0,336$ .

Im günstigsten Fall sind Trip und Request identisch, so dass die beiden Werte  $Weg\_rel$  und  $Zeit\_rel$  den Wert 0 haben.

Außerdem sollen noch die bisherige Bewertung des Fahrers sowie die Anzahl der User, die ihn ignorieren beachtet werden. Dazu wird die Durchschnittsbewertung des Fahrers mit 1 subtrahiert und dann durch 5 geteilt. Auf diese Weise erhält man einen Wert zwischen 0 und 1, wobei 0 aus einer Durchschnittsbewertung von 1,0 folgt und 1 aus einer von 6,0. Bei den Ignorierungen wird einfach die Anzahl der User, von denen der Fahrer ignoriert wird, durch die Gesamtzahl der User geteilt. So erhält man wiederum einen Wert zwischen 0 und 1.

Hat in unserem Beispiel der Fahrer des Trips also eine durchschnittliche Bewertung von 2,1 und wird er von 2 Usern ignoriert bei einer Gesamtuseranzahl von 100, so erhalten wir noch die Werte  $Note\_rel = (2,1 - 1) / 5 = 0,22$  und  $Igno\_rel = 2 / 100 = 0,02$ .

Diese vier Relativwerte werden dann nach dem Vorbild der euklidischen Norm verrechnet, so dass man einen aussagekräftigen Wert über die Güte des Trips erhält.

$$\sqrt{Weg\_rel^2 + Zeit\_rel^2 + Note\_rel^2 + Igno\_rel^2}$$

## 1.5 Tests

## Test-Driven-Development

### Allgemein:

Testgetriebene Entwicklung ist eine Methode, die häufig bei der agilen Entwicklung von Computerprogrammen eingesetzt wird. Bei der testgetriebene Entwicklung erstellt der Programmierer Software-Tests konsequent vor den zu testenden Komponenten.



## Gründe:

Nach klassischer Vorgehensweise werden Tests parallel zum und unabhängig vom zu testenden System entwickelt oder sogar nach ihm. Dies führt oft dazu, dass nicht gewünschte und erforderliche Testabdeckung erzielt wird. Ein weiterer Nachteil klassischer Tests ist, dass der Entwickler das zu testende System und seine Eigenheiten selbst kennt und dadurch aus Betriebsblindheit unversehens "um Fehler herum" testet.

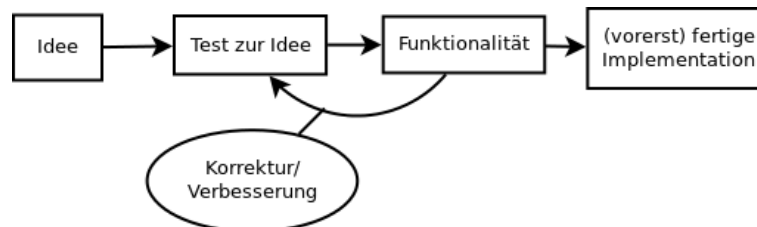
## Vorgehensweise:

Unit-Tests und mit ihnen getestete Units werden stets parallel entwickelt. Die eigentliche Programmierung erfolgt in Iterationen. Eine Iteration hat drei Hauptteile:

1. Schreibe Tests für das erwünschte fehlerfreie Verhalten, für schon bekannte Fehlschläge oder für das nächste Teilstück an Funktionalität, das neu implementiert werden soll.
2. Ändere/schreibe den Programmcode, bis nach dem anschließenden Test alles bestanden wird.
3. Räume dann im Code auf (Refactoring)! Natürlich wieder mit anschließendem Testen. Ziel des Aufräumens ist es, den Code schlicht und verständlich zu machen

Diese drei Schritte werden so lange wiederholt, bis die gewünschte Funktionalität erreicht oder der bekannte Fehler bereinigt ist und dem Entwickler keine sinnvollen Tests weiter einfallen. Die so behandelte programmtechnische Einheit (Unit) wird dann als (vorerst) fertig angesehen. Die gemeinsam mit ihr geschaffene Tests bleiben erhalten, um auch zukünftige Umsetzungen daraufhin testen zu können, ob das gewünschte Verhalten fortbesteht!

## Diagramm zur Vorgehensweise:



## Kritik:

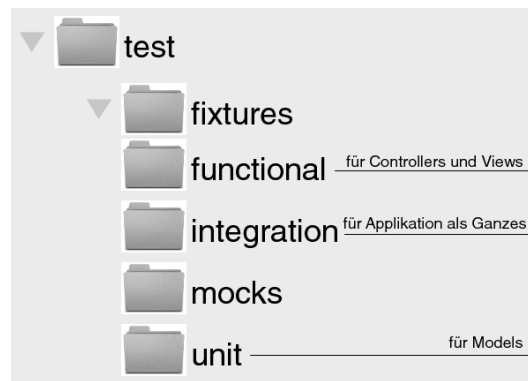
Auch die Methode der testgetriebenen Entwicklung kann falsch eingesetzt werden und dann scheitern. Programmierern, die noch keine Erfahrung dabei besitzen, erscheint sie manchmal schwierig oder gar unmöglich. Sie fragen sich, wie man etwas testen soll, das doch noch gar nicht vorhanden ist. Auswirkungen kann sein, dass sie die Prinzipien dieser Methode vernachlässigen, was zur extremen Programmierungsschwierigkeiten oder sogar dessen Zusammenbruch zur Folge haben kann.

## TDD - Umsetzung in Ruby on Rails:

Ruby on Rails unterscheidet zwischen drei Arten von Tests:

- Unit-Test - Hiermit werden hauptsächlich Models getestet.
- Functional-Tests - Setzt den Fokus auf das Testen von Controllern und Views
- Integration-Tests - Dient zum Testen der Gesamtfunktionalität der Rails-Applikation.

Für jeden Test-Typ gibt es ein eigenes Verzeichnis im Projekt.



Das Verzeichnis "fixtures" ist für Testdaten bestimmt. Im Verzeichnis "mocks" werden sogenannte Mock-Objekte (Attrappen-Objekte) abgelegt. Mock-Objekte sind Objekte, die nur die Funktion eines echten Objekts imitieren, aber nicht die Original-Funktionalität bieten. Dies ist z.B. sinnvoll, wenn ein Objekt Online-Zugriffe macht, die für die Tests zu aufwendig wären.

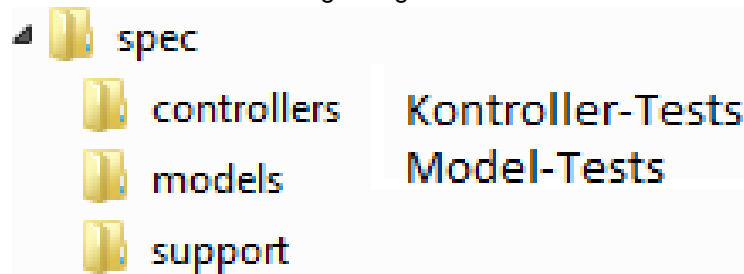
## TDD - Alternative Rspec gem

### Was sind RubyGems?

RubyGems ist das offizielle Paketsystem. Mit ihm hat der Anwender die Möglichkeit, mehrere Versionen eines Programmes, Programmteiles oder einer Bibliothek gesteuert nach Bedarf einzurichten, zu verwalten oder auch wieder zu entfernen.

### Rspec - Allgemein

Rspec ist ein Gem, das als alternative TDD-Umgebung dient. Das Verzeichnis sieht wie folgt aus.



### Bsp-Rspec

```
it "Kontrolle ob eine richtige e-mail zulässig ist" do
  #Erzeugen gültiger e-mail Adressen
  address = %w[user@foo.com THE_USER@foo.bar.org first.last@foo.jp]
  #Iteriere durch alle e-mail Adressen und teste sie
  address.each do |address|
    valid_email_user = User.new(@attr.merge(:email=>address))
    #falls sie nicht gültig sind schmeiße eine Exception
    valid_email_user.should be_valid
  end
end
```



## Seed-Data:

### Allgemein:

Die Seed-data dient als Hilfe um eine Datenbank mit Informationen bzw. Daten zu füllen. Im Allgemeinen ist sie eine Ruby-Datei mit einer Rails-Umgebung und vollem Zugriff auf alle Klassen. D.h. es besteht die Möglichkeit Objekte wie in der rails Konsole zu erzeugen. Dadurch ist eine schnelle und einfache Erstellung gegeben.

### Bsp-Code:

Im folgenden ist ein kleiner Ausschnitt der Seed-Data zu sehen.

```
#Erstellen eines Passengers
ps1 = Passenger.new :user_id => 2, :trip_id => 1, :confirmed => true
ps1.save!
```

### Vorteile:

Durch die Seed-data ist eine Testumgebung gegeben, die flexibel an den Änderungen der Applikation anpassbar ist. Dadurch besteht für alle Gruppen eine gute, schnelle und einfache Testumgebung zu Verfügung.

### Nachteile:

Beim erstellen der Seed-data, durch rake db:seed, wird jedes mal auf Online-Diensten zugegriffen, falls sie verwendet werden. Außerdem besteht ein hoher Wartungsaufwand. D.h. bei Änderungen an Tabellen werden jedesmal alle Datensätze der Tabelle angepasst.

## Fixtures

### Allgemein:

Mit Fixtures werden Beispieldaten für Tests generiert. Das Default-Format dafür ist YAML. Die Dateien dafür finden sich im Verzeichnis test/fixtures/ und werden mit rails generate automatisch mit erstellt. Sie können aber natürlich auch eigene Dateien definieren. Alle Fixtures werden per Default bei jedem Test neu in die Test-Datenbank geladen. Mit rails extract\_fixtures besteht die Möglichkeit aus der Seed-Data fixtures zu erstellen.

### Vorteile:

Einmaliges benutzen von Online-Diensten bei der Erstellung. Danach können sie schnell und einfach mittels rake db:load:fixtures in der Datenbank geladen werden.

## Nachteile:

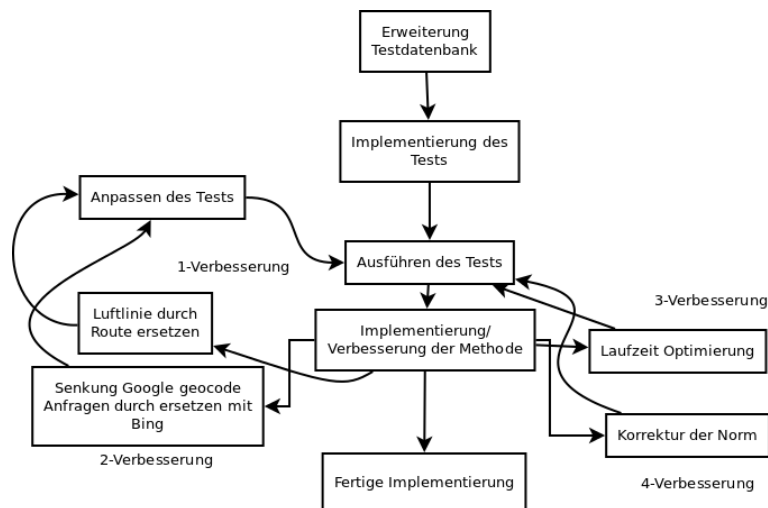
Jedoch sind fixtures statisch, d.h. zum Beispiel falls Daten bei Google verändert werden, sind die Daten in den fixtures solange veraltet, bis neue erstellt werden.

## Bsp. Code:

```
passengers_001:
  id: 1
  user_id: 2
  trip_id: 1
  confirmed: t
  created_at: '2011-08-11 07:12:25.106788'
  updated_at: '2011-08-11 07:12:25.106788'
```

## Beispiel: TDD-Anwendung

Im folgenden ist eine grafische Darstellung einer TDD Entwicklung ,anhand der Methode `get_sorted_requests`, zu sehen.



## Massentest:

### Ziele:

Unser Ziel war es die Funktionen und Laufzeit einiger Methoden zu testen (explizit Methoden aus der Optimierung).

### Umsetzung:

Dieser Test wurde durch eine normale Ruby-Datei, die in der Seed-data eingebunden wird mittels `require "massentest"`, umgesetzt.

## Problem bei der Umsetzung:

Bei der Umsetzung des Massentests gab es Probleme bei den Google geocode Anfragen, da sie auf 2500 Anfragen pro Tag begrenzt sind. Außerdem war es schwierig eine zufällige Ortswahl zur implementieren.

## Lösung

Da die Anfragen begrenzt sind, haben wir bei der Optimierung BING anstatt Google verwendet. Dadurch gibt es bei den Optimierungsmethoden 47500 Anfragen pro Tag mehr. Außerdem wurden zusätzliche Attribute *distance* und *duration* eingefügt und Attribute aufgespalten um weitere Google-Anfragen zu minimieren. Ebenso wurden Optimierungsmethoden verbessert um weitere Anfragen zu senken. Für die zufällige Ortswahl wurde ein Stadtverzeichnis als Array in einer Ruby-Datei angelegt. Dadurch ist es möglich diese einzubinden und mittels `rand(n)` (n Natürliche Zahl) zufällig Orte zu wählen.

## Folgen des Massentests:

- Laufzeitoptimierungen (Model)
- Behebung von Anzeige-Fehlern (View)
- Methoden korrigiert bzw. verbessert (Model und Controller)

## 1.6 Fazit

Alles in Allem ließen sich die an uns gestellten Aufgaben gut erfüllen. Trotz der Probleme, auf die wir im Verlauf unseres Praktikums stießen: z.B. die eingeschränkte Nutzbarkeit von Routenberechnungsdiensten, konnten wir eine lauffähige Version, die weitgehend den ursprünglich gesetzten Features gerecht wird, erstellen. Des weiteren war es uns am Ende der Programmierarbeit sogar möglich einen größeren Stresstest durchzuführen und so die Laufzeit zu überprüfen. Im Nachhinein lässt sich also festhalten, dass, grade durch die gute gruppeninterne Kommunikation, die Aufgabe in zufriedenstellender Weise erfüllt wurde.

## 2. Controller

Henning Strüber und Dominik Stegmann

Zusammenspiel der View und des Models gewährleisten.

### 2.1 Ausarbeitung

Dominik Stegmann und Henning Strüber

#### Inhalt

- Aufgaben des Controllers
- Workflow
- Zielsetzung
- Umsetzung
  - Konkrete Umsetzung
  - Rollenkonzept
  - Rechteverwaltung
- Fazit

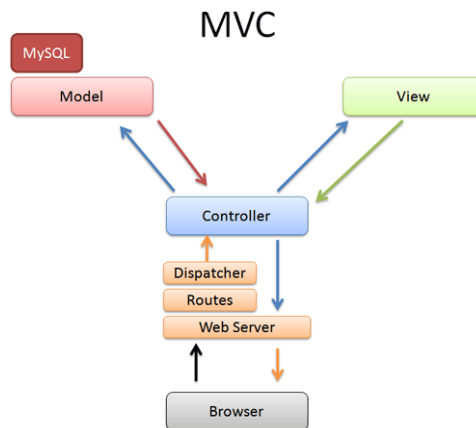
#### 2.1.1 Aufgaben des Controllers

Der Controller ist für die Koordination von Anwender und Applikation zuständig. Er nimmt Benutzereingaben der View entgegen und verarbeitet diese in Zusammenspiel mit der Applikationslogik, die vom Model in Form von Methoden zur Verfügung gestellt wird. Es liegt außerdem im Aufgabenbereich des Controllers dafür zu sorgen, dass das Routing einwandfrei funktioniert, d.H., dass die, vom Anwender gewünschte Funktion korrekt zu der, dafür zuständigen Controllerklasse und Controllermethode gelangt. Innerhalb der Klassen verarbeitet der Controller die Interaktion grundlegend mit 4 Methoden. Create, Read, Update und Destroy (CRUD). Die von Ruby standardmäßig zur Verfügung gestellten Methoden sind dementsprechend:

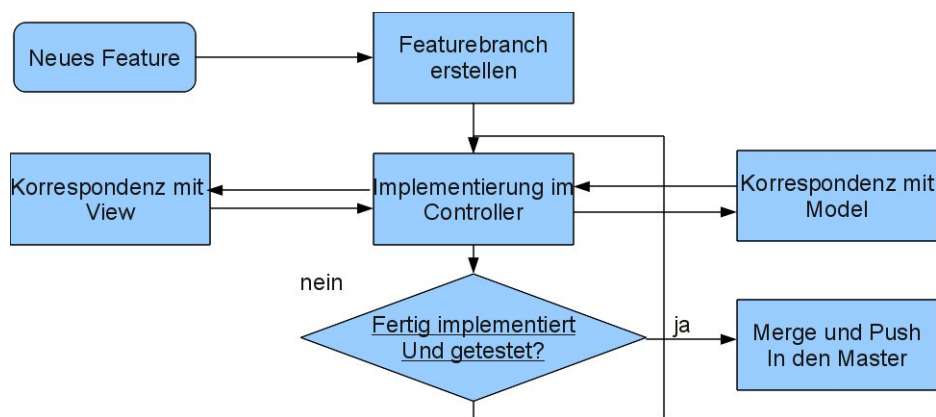
- index
  - Übersicht über alle Datensätze
- show
  - Ein Dateneintrag in der Detailansicht
- new
  - Ein leeres Objekt wird generiert und der View zur Verfügung gestellt
  - Die „new.html“ des Objektes wird aufgerufen
- edit
  - Anhand einer id, die von der view als Parameter zur Verfügung gestellt wird, wird eine Entität aus der Datenbank geholt und der View mit dem entsprechenden Template zur Verfügung gestellt
- create
  - Nach Absenden des Formulars in der „new.html“ wird hier das zuvor noch leere Objekt mit den Parametern befüllt und in die Datenbank gespeichert
- update
  - Ähnlich wie bei Create wird hier allerdings eine bereits vorhandene Entität mit den Parametern aus dem edit Template verändert und in die Datenbank zurückgespeichert
- destroy
  - Anhand einer id, die von der View mittels eines Parameters übergeben wird, suchen wir eine Entität in der Datenbank und löschen ebendieses.

Diese „rohen“ Methoden müssen dann während der Projektphase in Koordination mit den Model -und View Gruppen erweitert und verändert werden, damit die gewünschten Funktionalitäten gewährleistet werden können.

Ein weiterer wichtiger Punkt, den der Controller sicherstellen muss, ist die Rechte -und Rollenverteilung, so soll es einem gewöhnlichen Nutzer nicht möglich sein, beliebige Entitäten zu löschen, indem er einfach die richtige destroy Methode, mit entsprechender id aufruft.



## 2.1.2 Workflow



Zu Beginn eines Arbeitsschrittes stand die Korrespondenz mit der View-Gruppe, die ihrerseits Wünsche bezüglich einer genaueren Umsetzung eines bestimmten Features äußerte. Dann wurde ein Branch erstellt, auf der wir die Anpassungen am Controller realisieren konnten, während wir weiter in Kontakt mit der Model und View Gruppe standen. Die Model-Gruppe stellte uns Methoden zur Verfügung, die uns die von uns gewünschten Informationen aus der Datenbank holten, welche wir dann mit dem Aufruf des Templates weitergeben konnten. Nach dem erfolgreichen Testen der Funktionalität konnte der Branch in den Master gemerget und gepusht werden und stand so auch in der Hauptanwendung zur Verfügung.

## 2.1.3 Zielsetzung

Von Beginn an waren wir uns unserer Aufgabe bewusst, die Schnittstelle zwischen Anwender und Applikation zu sein. Wir wollten versuchen uns auf unsere drei Kernbereiche zu konzentrieren, das Routing, die Rollen -und Rechteverwaltung und die explizite Umsetzung der einzelnen Controller. Für die Rollen -und Rechteverwaltung wollten wir die Plugins Devise und CanCan benutzen, die die von uns gewünschten Funktionalitäten einfach in unsere Anwendung integrieren konnten. Bei der Umsetzung der einzelnen Controller wollten wir, und mussten wir uns in enger Zusammenarbeit mit den anderen Gruppen abstimmen. Wir überlegten uns auf welche Benutzereingaben der Controller reagieren wird müssen und wie wir die entsprechenden Verwendbarkeit umsetzen wollen.

## 2.1.4 Umsetzung

### 2.1.4.1 Konkrete Umsetzung

Jede Controllerklasse ist in ihrem Grundgerüst gleich. Sie erbt vom ApplicationController und ist mit den, bereits oben beschriebenen CRUD Operationen ausgestattet. Damit wir nur angemeldeten Nutzern den Zugang zu unserer Webanwendung erlauben, haben wir in jede Klasse einen `before_filter` integriert. Außerdem verfügen alle Klassen, außer dem RatingsController, über ein `load_and_authorize_resource` Statement, um die Berechtigungen des einzelnen Nutzers für die unterschiedlichen Methoden zu überprüfen, dazu aber in IV.b. mehr. Im Kopf des Controllers finden wir außerdem zwei bis drei Exception Handler, die u.a. die CanCan Exceptions oder die Standardexceptions fangen und die Fehlermeldung in eine für den Anwender leserliche Form wiedergeben.

## Messages

Der erste Teilbereich auf den ich jetzt hier eingehen möchte, ist unser Nachrichtensystem, bzw. den dafür zuständigen MessagesController. Nach Konsultation mit der View Gruppe, waren wir uns über die Features des Nachrichtensystems einig. Folgendes sollte umgesetzt werden:

- Anzahl der noch ungelesenen Nachrichten
- Übersicht über geschriebene Nachrichten
- Eine „Antworten-Funktion“
- Nachricht löschen
- Nachricht einen Trip betreffend

Nachdem wir uns einig waren, wie unser Nachrichtensystem funktionieren sollte, besprachen wir uns mit dem Model, damit die für uns notwendigen Datenfelder und Funktionen integriert werden konnten. In der `index` Methode des Controllers stellten wir der View dann eine Liste mit den Nachrichten des aktuellen Nutzers, und die Anzahl der noch ungelesenen Nachrichten zur Verfügung, außerdem ein Zeitstempel, der angibt, wann der Nutzer zuletzt sein Postfach abgerufen hat. So konnten im Template diejenigen Nachrichten, die einen Zeitstempel haben, der vor dem des letzten Aufrufs des Postfaches liegt, als ungelesen markiert werden. Um den Zeitstempel aktuell zu halten wurde dieser anschließend noch auf die aktuelle Zeit gesetzt. Die `new` Methode passten wir so an, dass sie die gewünschten Funktionalitäten erfüllt, indem wir drei mögliche Parameter von der View berücksichtigen, die wir per GET Request bekommen. Je nachdem, welchen Parameter wir bekommen, bearbeiten wir das Message-Objekt, indem wir den Empfänger setzen oder den Betreff automatisch mit einem Link zu der Fahrt befüllen, den die Nachricht betrifft. Das Löschen einer Nachricht gestaltete sich schwieriger als bei den anderen Objekten, da wir nicht einfach die `destroy` Methode aufrufen konnten. Da unser Nachrichten Objekt in der Datenbank nur einmal vorhanden ist, aber sowohl vom Autor als auch vom Empfänger gelesen wird, können wir, wenn nur einer von beiden die Nachricht löscht, sie nicht einfach entfernen, da sie sonst auch für die Gegenseite nicht mehr sichtbar ist. Um dieses Problem zu lösen, wurde ein weiteres Datenfeld eingefügt, welches die Sichtbarkeit der Nachricht für den Autor, bzw. den Empfänger regelt. Der „Löschvorgang“ wurde dann eher als eine Art Update implementiert, indem das Sichtbarkeitsfeld der entsprechenden Nachricht geändert wurde. Wird hierbei festgestellt, dass sowohl Autor als auch Empfänger die Nachricht gelöscht haben, wird sie entgültig aus der Datenbank entfernt. Beispielhaft ist hier der Codeausschnitt der `update`-Methode zu sehen, die den „Löschvorgang“ realisiert.

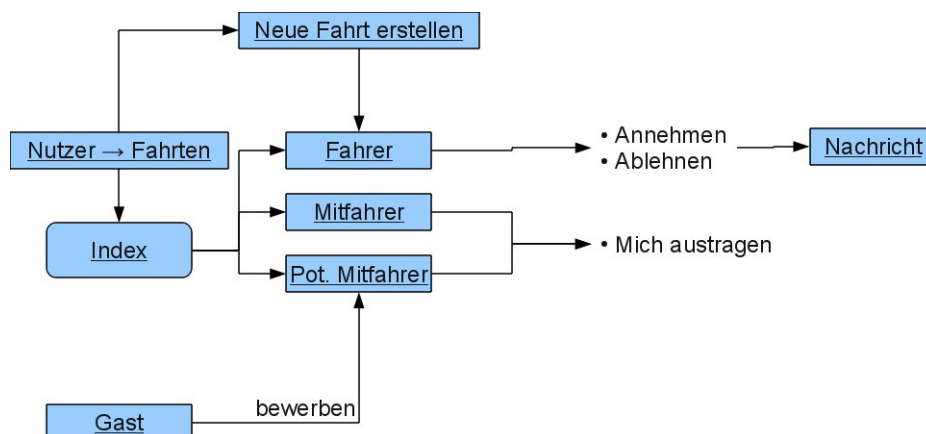
```
def update
  @message = Message.find(params[:id])
  # "Loescht" der Empfaenger eine Nachricht, wird sie fuer ihn invisible
  if params[:who] == "receiver"
    @message.delete_receiver = true
  # "Loescht" der Schreiber eine Nachricht, wird sie fuer ihn invisible
  elsif params[:who] == "writer"
    @message.delete_writer = true
  end
  @message.save
  # Haben Autor und Empfaenger einer Nachricht sie "geloescht" wird sie auch richtig aus der DB
  entfernt
  if @message.delete_receiver and @message.delete_writer
    @message.destroy
  end
end
```

## Trips

Da der Kernpunkt unserer Anwendung aber ja die Fahrten sind, die angeboten werden, haben wir die meiste Zeit in diesen Bereich investiert. Folgende Ideen wurde dazu entwickelt:

- Übersicht über meine Fahrten (Kommende und absolvierte)
- Detailansicht einer Fahrt
- Je nach Status unterschiedliche Berechtigungen
  - Bewerbungen, Annehmen, Ablehnen, Nachricht schicken, Rating abgeben, wenn Fahrt vorbei
- Beim Fahrt erstellen Start und Zieleingaben anhand eines Eingabefeldes
- Fahrt kann abgesagt werden, wenn sie noch nicht absolviert wurde

Die Übersicht über meine Fahrten wurde wieder in der index Methode realisiert, indem wir aus der Datenbank, mithilfe der Methoden des Models, die entsprechenden Datensätze holen und sie an die View übergeben. Hier wird noch unterschieden, ob ich Admin Berechtigungen haben oder nicht. Sollte dies der Fall sein, werden mir nicht nur meine, sondern alle Fahrten zur Verfügung gestellt. Die Detailansicht eine Fahrt gestaltet sich schwieriger, weil mir mit den bisherigen Rollen des Member und des Admins nicht unterscheiden konnte, ob ich Fahrer einer Fahrt bin oder ich mich auf ebendiese bewerben kann. Gelöst haben wir dieses Problem indem wir spezielle „Konstanten“ eingeführt haben, die meine jeweilige Rolle beschreiben und mir dementsprechende Rechte zuweisen. Rufe ich die Detailansicht einer Fahrt auf, wird zuerst mein Status überprüft, welches mithilfe einer ausgelagerten Methode geschieht. Neben weiteren Variablen für die freien und bereits besetzten Plätze der Fahrt überprüfen wir ob der Nutzer uns weitere Parameter mitsendet, die wir dann entsprechend behandeln mussten. So können wir eine neue Bewerbung für die Fahrt, oder eine Annahme, bzw. eine Ablehnung oder Austragung realisieren. Sollte dieser Parameter erkannt werden, wird noch überprüft, ob der Nutzer dazu überhaupt berechtigt ist, zB. Sollte ein Mitfahrer nicht dazu in der Lage sein, andere Bewerbungen anzunehmen oder abzulehnen, dies ist nur dem Fahrer vorbehalten. Als Besonderheit wird bei allen Anfragen eines Nutzers, sei es eine Bewerbung oder eine Annahme durch den Fahrer automatisch eine Nachricht verschickt. So wird der Bewerber auf eine Fahrt automatisch über das Nachrichtensystem benachrichtigt, wenn er angenommen, bzw. abgelehnt wurde. Bei der Erstellung einer Fahrt wird, wird zur Start -und Zieleingabe nur jeweils ein Eingabefeld angeboten, um es dem Nutzer einfacher zu machen, seine Wünsche einzutragen. Hier kommt es allerdings zu einer Diskrepanz zwischen Nutzereingabe und dem, was die Datenbank erwartet. Innerhalb der Datenbank ist es aus datentechnischen Gründen notwendig, die Start -und Zieleingabe in jeweils drei Datenfeldern zu speichern. Um dieses Problem zu bewältigen wird bei der Erstellung einer Fahrt zuerst anhand der Benutzereingabe mittels des Geocoders jeweils ein Paar von Koordinaten für den Start -und Zielpunkt ermittelt, diese werden dann in das Objekt eingetragen. Um die Daten in die notwendigen drei Teile aufzusplitten, nutzen wir hier eine vom Model zur Verfügung gestellte Methode, die am aktuellen Fahrten-Objekt aufgerufen wird und es passend modifiziert. Ein weiteres Problem, das von uns gelöst werden musste, ergab sich bei der Eingabe der gewünschten Abfahrtszeit. Von der View bekommen wir 5 Parameter übergeben, die in der Datenbank allerdings in einem Datenfeld gespeichert werden. Wir konnten diesen Konflikt leicht beseitigen indem wir die Eingaben einfach in der syntaktisch richtigen Form aneinanderfügen.



## Requests

Der Gegenpart zur Fahrt ist in unserer Anwendung die „Anfrage“, oder auch der „Request“. Anstatt einer Suche, in der mir alle passenden Fahrten ausgegeben werden, erstellen wir Anfragen, die persistent in unserer

Datenbank gespeichert werden. Im Vergleich zu den Fahrten war die Umsetzung unserer Ideen bezüglich der Anfragen recht einfach. Es sollte möglich sein, uns alle unsere Anfragen anzuzeigen, und davon eine in Detailansicht anzuschauen, in der dann wiederum passende Fahrten auf eben diese Anfrage aufgelistet werden. Die Umsetzung ebendieser Übersichts- und Detailansicht erfolgte analog zu denen der Fahrten. Die passenden Trips zu unserer Anfrage beschaffen wir uns mithilfe einer vom Model zur Verfügung gestellten Methode. Auch die Erstellung von Anfragen passiert fast analog zur Fahrtenerstellung, indem wie aus einer Nutzereingabe die Koordinaten bestimmen und sie vom Model in drei Teile splitten lassen.

## Ratings

Ein weiterer wichtiger Punkt, ist das Bewertungssystem. Es sollte möglich sein, für jede Fahrt an der ich teilgenommen habe, den Fahrer und meine Mitfahrer zu bewerten. Wie bei den anderen Controllern wurde wieder eine Übersicht über meine Ratings integriert, und wir geben eine Variable an das Template weiter, welches meine, noch ausstehenden Ratings enthält. Innerhalb der show Methode lassen sich, anhand einer übergeben id, die Ratings des, dem der id zugehörigen Nutzers anzeigen. Dies ist standardmäßig der aktuell eingeloggte Nutzer, jedoch kann es ebenso von Interesse sein, die Bewertungen des Fahrers zu überprüfen an dessen Fahrt ich teilnehmen möchte. Die show erlaubt in diesem speziellen Fall keine Detailansicht eines einzelnen Ratings, sondern erfüllt eine ähnliche Funktion wie die index Methode. Ähnlich wie beim Message Controller prüfen wir auf neue Ratings und setzen gegebenenfalls den Zeitstempel auf die aktuelle Zeit.

### 2.1.4.2 Rollenkonzept

Ein Rollenkonzept wird in der Informatik genutzt, um zwischen den verschiedenen Arten von Nutzern einer Anwendung zu unterscheiden und ihnen allgemein verschiedene Rechte, Sichten etc. zuzuteilen.

## Implementierungsmöglichkeiten der Rollen

Bei der Implementierung eines solchen Konzeptes müssen mehrere Design-Ansätze gegeneinander abgewogen werden:

Wie viele Rollen soll ein Benutzer einnehmen können? Zuerst muss entschieden werden, an welchen Stellen verschiedene Rollen benötigt werden und wie differenziert diese sein sollen. Also ob zum Beispiel ein Moderator in einem Forum nur in einem Bereich erweiterte Rechte besitzt und zu diesen noch weitere Bereiche hinzugefügt werden können, was in einer N : M-Beziehung zwischen den Moderatoren und den Forenbereichen besteht oder ob er nur einen Bereich bzw. das gesamte Forum moderieren kann, was in einer N : 1-Beziehung zwischen den Moderatoren und den Bereichen resultieren würde.

Wie wird das Rollenkonzept in der Datenbank repräsentiert? Dieses ist auf zwei Arten möglich. Zum einen kann man für jede Rolle eine eigenständige Entitäts-Klasse implementieren, was den Vorteil hat, dass man große Unterschiede zwischen Eigenschaften der einzelnen Rolleninhaber machen kann. Jedoch verkompliziert dies eine mögliche spätere Veränderung der Rolle eines Nutzers. Außerdem ist der Anpassungsaufwand der restlichen Applikation für Anwendungen, bei denen der Unterschied zwischen den einzelnen Rollen nicht so wichtig ist, größer als der daraus resultierende Gewinn. Die andere Möglichkeit Rollen zu implementieren, ist es dem User ein zusätzliches Attribut hinzuzufügen, welches dann bei einer Unterscheidung der verschiedenen Rollen abgefragt werden kann; dies ist eine einfache Implementierung, welche außerdem die Möglichkeit bietet, dass man dem Administrator die Fähigkeit gibt weitere Moderatoren etc. einzusetzen, ohne dass jeweils eine Entität zerstört und neu aufgebaut werden muss.

## Unser Rollenkonzept

Wir haben für eine relativ einfache Lösung entschieden, da wir keinerlei komplexe Zusammenhänge zwischen verschiedenen Rollen implementieren mussten, sondern nur eine Abgrenzung zwischen den Rechten von Mitgliedern und Administratoren erreichen wollten. Wir vergeben zwar auch bei den Aufrufen eines Trips verschiedene Rollen, allerdings sind diese nur für diesen eine Fahrt gültig und werden auf anderen Wegen in



der Datenbank festgelegt. (s.o.) Infolge der geringen Komplexität der Rollenverhältnisse haben wir uns dafür entschieden, dass jeder Benutzer nur eine Rolle einnehmen kann und dass diese mittels eines Attributes innerhalb des User-Modells festgelegt wird, da uns der Aufwand der Anpassung unserer Anwendung an eine neue Nutzer-Entität Administrator bei den geringen Unterschieden als nicht lohnenswert erschien.

## Rechte der verschiedenen Rollen

Nicht angemeldete Nutzer erhalten keinerlei Zugriff auf Informationen oder andere Inhalte, sondern können sich ausschließlich anmelden bzw. neu registrieren. Ein Administrator hat komplette Zugriffs- und Editierrechte für die gesamte Anwendung. Ausgenommen sind hiervon allerdings die privaten Nachrichten zwischen Mitgliedern. Ein Nutzer kann jegliche Inhalte der Anwendung erstellen. Ein Nutzer hat das Recht die von ihm selbst erstellten Inhalte zu lesen, zu editieren und zu löschen. Eine Ausnahme stellt hier die Löschung von bereits versendeten Nachrichten dar. Außerdem kann er die nicht versteckten Informationen und die Ratings über den Fahrer und das genutzte Auto von angebotenen Fahrten einsehen. Die Profile der Mitfahrer einer Fahrt kann ein Benutzer einsehen, wenn er ein bestätigter Teilnehmer dieser Fahrt ist. Ein Nutzer kann selbstverständlich alle Nachrichten, die er geschrieben bzw. erhalten hat einsehen, diese sind allerdings von ihm selbst nicht global löscherbar, sondern nur in seiner persönlichen Anzeige entfernbar.

Die Rollen werden vornehmlich mittels der Rechteverwaltung implementiert, allerdings erhält ein Administrator auch stärkeren direkten Zugriff auf Dateien, was sich vor allem in der View äußern sollte. Dies konnte allerdings in den drei Wochen des Praktikums aufgrund von Zeitmangel nicht mehr verwirklicht werden.

### 2.1.4.3 Rechteverwaltung

Die Rechteverwaltung von angemeldeten Nutzern haben wir mithilfe des Ruby on Rails Gems CanCan implementiert. Die Unterscheidung zwischen nicht angemeldeten und angemeldeten Nutzern übernimmt bei uns hingegen das Devise Authentifizierungs-Gem, wobei wir allerdings auf dieses nicht weiter eingehen, da nach der Installation nur sehr geringfügige Anpassungen notwendig waren.

## CanCan

CanCan ist ein Ruby on Rails Autorisierungs-Gem, welches vornehmlich mit dem Schlüsselwort „can“ und einer „ability“ genannten Klasse arbeitet. Mit dem Schlüsselwort „can“ kann man die von dem jeweiligen User nutzbaren Controller festlegen, was wie folgt in der Implementierung aussieht:

```
can :index, :all
```

Der vorliegende Befehl bedeutet, dass man alle Index-Controller-Methoden nutzen kann. Man gibt also nach dem Schlüsselwort „can“ zuerst die Controller-Methode an und dann für welche Controller die vorangegangene Autorisierung gelten soll. Alternativ zur expliziten Angabe aller nutzbaren Controller-Methoden können vordefinierte Zusammenfassungen von mehreren Standard-Controllern genutzt werden: read: fasst die Index- und die Show-Methoden zusammen create: fasst die New- und die Create-Methoden zusammen manage: fasst alle Methoden zusammen

Der „cannot“-Befehl überschreibt ein bewilligtes „can“ wieder, sodass Ausnahmen von „manage“ eingefügt werden können.

Mithilfe von Schleifen oder durch im Modell definierte Bedingungen, welche Untermengen einer Entität identifizieren, kann man zusätzlich spezielle Bedingungen für die Controller-Zugriffe stellen:

```
can [:show, :update, :destroy], Car do |car|
  car && car.user == user
end
```

```
can :show, user.get_visible_cars
```

Diese Zeilen sind die Festlegung der Zugriffsberechtigungen für die Autos in unserer Anwendung. Zum einen können Nutzer auf ihre eigenen Autos zugreifen und zum anderen können sie die Autos sehen, welche in angebotenen Fahrten verwendet werden. Spezielle Bedingungen können auch mithilfe von Hashes hinzugefügt werden. So könnte man zum Beispiel einem Objekt verschiedene Prioritäten zuweisen. Je nach Priorität wird allen Nutzern, den Moderatoren oder nur dem Administrator Zugriff gewährt:

```
can :read, Project, :priority => 1..3
```

Dieses Beispiel bedeutet, dass der entsprechende User Zugriff auf die Controller-Methoden „index“ und „show“ von allen Projekten mit der Priorität 1-3 möglich ist. Dies hätten wir in unserem Projekt vielleicht anstelle der Konstanten-Lösung bei der Unterscheidung der verschiedenen Trip-Rollen nutzen können, indem wir den verschiedenen Positionen jeweils eine Zahl zugeordnet hätten, welche dann von der View spezifisch hätte abgefragt werden können. Allerdings empfanden wir diese Lösung zu diesem Zeitpunkt als wenig einleuchtend und haben uns deswegen für die schon beschriebene Lösung entschieden. Vom Vorgehen bezüglich der Rollen- und Rechteverwaltung her wäre dies in sich wohl konsistenter gewesen.

Nutzung und Abfrage der definierten Rechte: Dies kann zum einen mithilfe einer spezifischen Abfrage mit dem Schlüsselwort „can?“ geschehen, was vor allem in der View und im Controller in Form von if-Abfragen erfolgt. Dabei wird zum Beispiel entschieden, ob ein bestimmter Button nur für jemanden erscheint, der das Recht hat eine bestimmte Entität zu aktualisieren. Es kann außerdem abgefragt werden, ob demjenigen der eine Entität aktualisieren kann, mehr Informationen übergeben werden sollen oder ob unterschiedlich auf die Eingaben von Nutzern mit verschiedenen Berechtigungen reagiert werden soll. Wenn es darum geht einen gesamten Controller für Unberechtigte zu sperren nutzt man hingegen:

```
load_and_authorize_resource
```

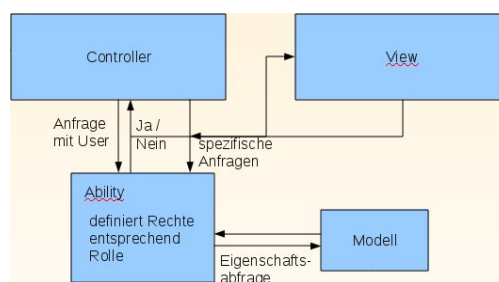
oder bei einer einzelnen Controller-Methode z.B.:

```
@rating = Rating.find(params[:id])
authorize! :update, @rating
```

um abzufragen, ob das momentan ausgewählte Rating aktualisiert werden darf.

Wenn nun ein Zugriff auf eine Controller-Methode verweigert wird, wird eine `CanCan::AccessDenied-Exception` geworfen, welche man dann wie jede andere Exception fangen und weiter bearbeiten kann.

## Beispiel eines Controllerzugriffs



Hier wird noch einmal beispielhaft der Zugriff auf einen Controller erläutert:

1. Jemand fragt von seinem Browser aus eine Adresse an und wird zum entsprechenden Controller geroutet.
2. Wenn der Nutzer nicht angemeldet ist, wird er direkt vom Devise gestoppt und zur Anmeldung weitergeleitet.
3. Wenn er angemeldet ist, fragt der Controller bei der Ability-Klasse nach, ob der Nutzer die Berechtigung hat die Seite zu nutzen.
4. Die Ability nutzt dann je nach Rolle verschiedene Sets von Berechtigungen.
5. In diesen Sets sind Abgleiche mit dem Modell, ob ein Nutzer bestimmte Eigenschaften besitzt, enthalten.
6. Je nachdem ob der Nutzer die gesuchten Eigenschaften erfüllt, gibt die Ability an den Controller eine true/false-Antwort zurück.

7. Bei true erhält der Nutzer Zugriff auf den Controller und damit auf das verbundene View-Template, falls die Antwort false ist wird eine Exception geworfen, welche den Nutzer mit einem entsprechenden Hinweis auf eine vordefinierte andere Seite umleitet.

Spezifische Anfragen eines Controllers oder eines View-Templates mit „can?“ werden im Prinzip genauso behandelt; der einzige Unterschied ist, dass die Reaktion auf die Rückgabe andere Konsequenzen hat.

## 2.1.5 Fazit

Zusammenfassend kann gesagt werden, dass wir, trotz kurzer, anfänglicher Schwierigkeiten, unsere Aufgabe erfolgreich bewältigen konnten. In der Rolle als Vermittler zwischen der Model -und der View Gruppe haben wir als Team gut funktioniert und auch die Umsetzung der Rollen -und Rechteverwaltung konnte ohne größere Probleme umgesetzt werden.

## 3. Frontend

Im Rahmen des Datenbankpraktikums 2011 hat sich unsere Gruppe mit der Umsetzung und Gestaltung der Web-Oberfläche für die Mitfahrzentrale beschäftigt. Unsere Aufgabe bestand darin Views (Ruby-Templates) zu erstellen mit deren Hilfe alle Funktionen für Nutzer der Applikation zur Verfügung stehen. Im wesentlichen haben wir in Ruby, Haml, Scss und Javascript programmiert. Neben dem Erlernen der einzelnen Programmiersprachen stand vor allem die Kommunikation mit den anderen Gruppen im Vordergrund. In der folgenden Ausarbeitung möchten wir das Layout vorstellen und erläutern, wie die einzelnen Views mit den jeweiligen Funktionalitäten aufgebaut sind.

### Gruppenmitglieder

- Ralph Buß
- Sascha Ebelt
- Karina Meyer
- Patrick Schnetger

### 3.1 Umsetzung

Dieser Abschnitt beschäftigt sich mit dem Konzept und den zum Einsatz gekommenen Skriptsprachen, beim Erstellen des Frontends.

- Layout
- SCSS
- HAML
- Javascript
- Paperclip
- Photoshop
- Googlemaps

#### 3.1.1 Layout

Zur Formatierung des gesamten Webseitenlayouts nutzen wir, wie in RoR üblich `~.html.haml`-Dateien. Diese werden per `'= %yield'`-Tag an gewünschter Stelle in der `application.html.haml` eingebunden. Kontextabhängig bekommen wir dadurch den richtigen Inhalt an richtiger Stelle.

Im 'Application'-Layout haben wir neben den üblichen imports der `application.css/scss` die Bereiche der Homepage definiert (Abb.: 1).

#### Header:

Als Header haben wir einen Photoshop-Banner eingefügt.

#### Navigation (Nav):

Je nachdem ob wir eingeloggt sind oder nicht bekommen wir eine unterschiedliche Navigationsleiste, die uns auch auf neue Nachrichten oder Bewertungen aufmerksam macht. In Abb.: 1 sehen wir die Navigationsleiste für Nutzer im eingeloggten Zustand. Wir können nach Fahrten suchen, eigene Fahrten erstellen, Nachrichten und Bewertungen einsehen/schreiben, die Fahrzeuge/das Profil anschauen oder uns ausloggen.

## Main:

In unserem Main-Bereich werden die jeweiligen Seiten neu geladen.



Abb. 1: Homepage Layout

### 3.1.2 CSS

## CSS

Cascading Style Sheets (CSS) werden verwendet, um Inhalt und Layout einer Website zu trennen. Im HTML-Dokument steht ausschließlich der strukturierte Inhalt. Dieser wird mittels CSS in einer externen Datei formatiert. Durch diese Trennung wird eine einheitliche Seitengestaltung vereinfacht: Identische Elemente brauchen nur einmal in CSS formatiert werden, anstatt bei jeder Deklaration und das Layout kann für mehrere Seiten verwendet werden und einfach ausgetauscht werden.

In CSS werden HTML-Elemente mit ihrem Namen oder über Klassen (class) oder Instanz (id) Bezeichner identifiziert. Um geschachtelte HTML-Elemente auszuwählen, werden diese Bezeichner entsprechend kaskadiert. Formatierungen für ein Element gelten auch für alle dessen Kinder, sofern diese auch entsprechend formatiert werden können und nicht explizit anders formatiert wurden.

## CSS Version 3

CSS Version 3 bietet eine Reihe neuer Formatierungsmöglichkeiten, von denen die folgenden für die Gestaltung der Mitfahrzentrale verwendet wurden:

Durch Text- und Elementschatten wirken Elemente plastischer und Kanten wirken weniger hart.

```
box-shadow: 10px 10px 5px #888;
text-shadow: 2px 2px 2px #666;
```

Beispieltext mit Schatten in einer Schattenbox.

Durch Pseudoelemente können HTML-Elemente nach speziellen Regeln ausgewählt werden, um zum Beispiel den Hintergrund von Tabellenzeilen abwechselnd unterschiedlich einzufärben.

```
tr:nth-child(2n) {
    background-color: #7C82BA;
}
tr:nth-child(2n+1) {
```



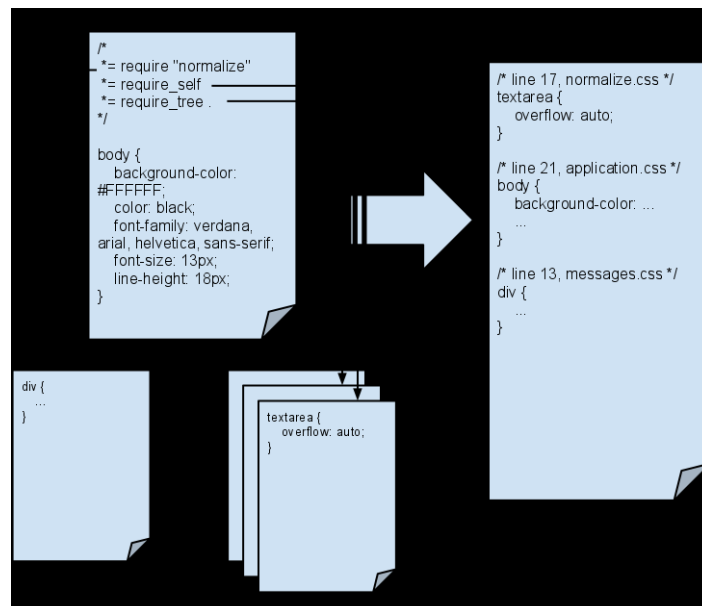
```

#main {
  position: relative;
  z-index: 0;
}
#main .other-site {
  position: absolute;
  width: 70%;
}
#main .other-site a:last-child {
  color: green;
}
#main .form_table {
  border: solid 2px;
}
#main .form_table td {
  color: green;
}
#main .form_table tr:first-child td {
  padding-top: 0.3em;
}

```

## Sprockets

Der Aufbau einer Website in Ruby on Rails besteht aus einer Hauptseite, in die jeweils entsprechende Teilseiten eingebettet werden. Um diesem Aufbau auch in CSS zu folgen, wird Sprockets verwendet. Dabei handelt es sich um eine Erweiterung, die aus vielen Dateien Zeilenweise eine große Datei zusammensetzt. Dabei muss in der Hauptdatei mithilfe von Regeln festgelegt werden, in welcher Reihenfolge die anderen Dateien hinzugefügt werden sollen. Das Prinzip zeigt folgende Grafik:



### 3.1.3 Formulare

Ein Formular in HaML zu erstellen ist relativ simpel. Allerdings ist es um einiges komplexer, die Werte, die in das Formular vom Benutzer eingegeben werden auch so bereitzustellen, dass der Controller sie auffangen und weiterverarbeiten kann. Hierzu dient in RoR das Modul FormHelper, das ein Untermodul des Moduls ActionViews::Helpers ist.

In HaML sieht der Kopf unserer Formulare deshalb folgendermaßen aus:

```

= form_for(resource, :as => resource_name, :url => registration_path(resource_name)) do |f|

```

```
= devise_error_messages!
```

Hierbei erzeugen wir mit `|f|` eine Instanz von der Klasse FormBuilder mit Namen `f`. Diese Klasse FormBuilder enthält nun eine Reihe von Methoden, mit deren Hilfe man nun Struktur und Funktionalität des Formulars, das man erzeugen möchte, festlegen kann.

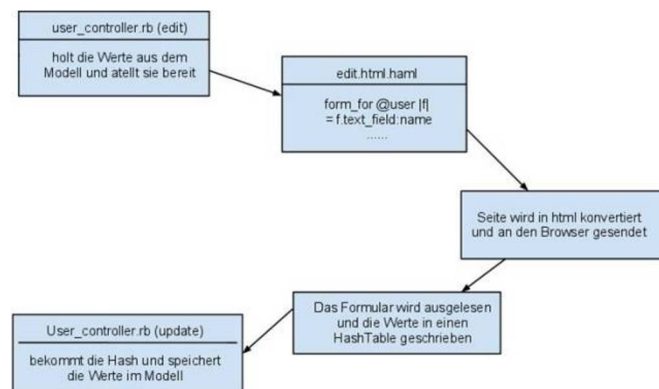
Eines der einfachsten Elemente ist das `text_field`:

```
= f.text_field:address, :required=>"required"
```

Neben dem `text_field` haben wir für unsere Formulare natürlich noch eine ganze Reihe weiterer Elemente verwendet. Wie:

- `text_field` (eine Textzeile)
- `text_area` (Textfeld für freitext)
- `email_field` (dies erkennt automatisch ob das eingegebene von der Form `abc@cde.ff` ist)
- `password_field` (automatisch das eingegebene in der Form `***` anzeigt)
- `checkbox` (generiert ein Feld zum Ankreuzen, zum füllen von boolean- Werten)
- `select` (generiert ein Drop-down Menü mit dem angegeben Inhalt)
- `date_select` (generiert drei Drop-down Menü für Tag, Monat, Jahr)
- `submit` (erzeugt einen link, der beim klicken das Formular an den Controller sendet)

Doch was passiert mit den Werten die in das Formular eingegeben werden nun genau?



**Abb. 2:** Vom Formular zum Controller

Abb1 zeigt, am Beispiel des Formulars für die Editierung eines Profils, wie die Kommunikation mit den Controllern funktioniert. Hierbei werden erst die Werte, die, durch das anlegen des Profils, bereits im Modell stehen von der `edit`- Methode des `User_Controller`s aus dem Modell geholt und an die View weitergegeben, sodass sie im Formular zur Änderung stehen. Drückt der Anwender nun nach dem Ändern der Daten auf `submit`, so wird die Seite zuerst in HTML konvertiert und anschließend an den Browser gesendet. Ist dies erfolgreich, so werden die Werte des Formulars in einen HashTable ausgelesen. Diese kann dann anschließend von der `update`- Methode des `User_Controller`s ausgelesen und ins Modell geschrieben werden.

### 3.1.4 Javascript

Dieser Abschnitt der Dokumentation gibt einen Überblick über eigens entworfene JavaScripts, die der Optischen Aufbereitung und Benutzerfreundlichkeit dienen. Es wird hierbei der jeweilige JavaScript Code dargelegt und erläutert. Die Implementierung der Google-Maps Karte in den Detailansichten der Fahrten, wird als gesondertes eigenes Thema behandelt (siehe hierzu Kapitel 2.7 Google Maps).



## Sitzplätze neuer Fahrten je nach Fahrzeug

Hierbei handelt es sich um eine JavaScript Methode, die auf der Formularseite *Neue Fahrt erstellen* dafür sorgt, dass nur maximal so viele Sitzplätze für diese Fahrt gewählt werden können, wie das ausgewählte Auto besitzt. Die folgende Abbildung zeigt als ausgewähltes Fahrzeug "Kleinbus (10 Sitzplätze)". Möchte der Fahrer nun die für diese Fahrt verfügbaren Plätze verändern, so kann er zwischen eins und neun wählen. Standardmäßig ist *keine Änderung* ausgewählt, was der Maximalzahl an Plätzen entspricht - in diesem Beispiel zehn.

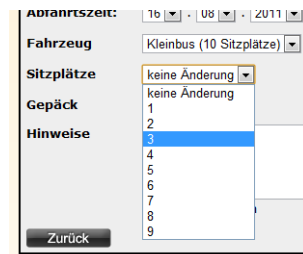


Abb. 3: Sitzplätze Dropdown

Ermöglicht wird dies, indem zuerst ein leeres JavaScript Array erzeugt wird. Dieses wird daraufhin mit Hilfe von Ruby gefüllt, indem über alle Fahrzeuge des Benutzers iteriert und die Anzahl an Sitzplätzen in das Array geschrieben wird. Für das Füllen des Arrays wird bei jedem Iterationsschritt eine entsprechende JavaScript Codezeile generiert.

```
:javascript
var autos = new Array();

- @fahrzeuge.each do |i|
  :javascript
  autos.push(#{i.seats});
```

Auf diese Weise kann mit Hilfe einer JavaScript Methode abgefragt werden, welches Auto gerade gewählt wurde, um iterativ die Auswahlmöglichkeiten in die Sitzplätze-Dropdown-Liste einzufügen. Hierbei wird zuerst die Dropdown-Liste mit der Auswahl an Sitzen (trägt die HTML-ID *freeseats*) geleert und anschließend der Eintrag *keine Änderung* hinzugefügt. Nun wird ausgelesen, welcher Element-Index der Dropdown-Liste mit den Fahrzeugen ausgewählt ist. Mit diesem Index wird auf das, wie oben beschriebene, Array mit den Sitzplätzen zugegriffen und per For-Schleife eine Reihe von Einträgen zur Sitzplätze-Dropdown-Liste hinzugefügt.

```
function carDropDown(){
  document.getElementById("freeseats").options.length = 0;
  document.getElementById("freeseats").options[0] = new Option("keine Änderung", "");
  for (i = 0; i < autos[document.getElementById("car").selectedIndex]-1; i++){
    document.getElementById("freeseats").options[document.getElementById("freeseats").length]
    = new Option((i+1),(i+1),false,false);
  }
}
```

Diese Methode wird immer dann aufgerufen, wenn ein anderes Auto ausgewählt wird. Diese Überwachung geschieht per JavaScript-Event (onchange). Zuletzt wird die Methode einmalig direkt aufgerufen. Dadurch wird die Sitzplätze-Dropdown-Liste initialisiert.

## Verbleibende Zeichen

Auf diversen Unterseiten des Projektes wurde ein JavaScript eingesetzt, das dem Benutzer anzeigt, wie viele Zeichen er in einem Textfeld noch eingeben darf, wie die nachfolgende Abbildung zeigt. So hat der Benutzer einen Überblick darüber, wieviel er noch schreiben kann.

Abb. 4: Textfeld mit verbleibenden Zeichen

Die JavaScript Methode liest hierfür die für das Textfeld festgelegte Maximal Zahl an Zeichen aus und subtrahiert hiervon die Zahl bereits eingegebener Zeichen. Das Ergebnis wird zusammen mit dem Text, wie er im obigen Bild zu sehen ist, in ein HTML-Div-Element geschrieben. Das Div mit dem Text trägt hierbei die ID *cmthinweis* und das Textfeld *comment*.

```
function chars_left(){
    document.getElementById("cmthinweis").firstChild.nodeValue
    = "Noch " + (document.getElementById("comment").maxLength
    - document.getElementById("comment").value.length)
    + " Zeichen";
}
```

Der Aufruf dieser Methode wird von JavaScript-Events gesteuert. Hierbei werden drei Ereignisse abgefangen:

- onkeydown  
wird aufgerufen, wenn der Benutzer im Textfeld eine Taste drückt
- onkeyup  
wird aufgerufen, wenn der Benutzer im Textfeld eine Taste wieder loslässt
- onkeypress  
wird aufgerufen, wenn der Benutzer im Textfeld eine Taste drückt und gedrückt hält

Dadurch, dass diese drei Events zum Aufruf der besagten Methode verwendet werden, entsteht ein flüssiges Gesamtbild beim Anzeigen der verbleibenden Zeichen.

## Karte skalieren

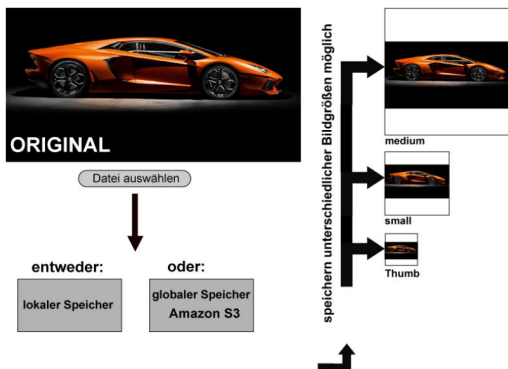
Auf den Detailseiten der Fahrten werden Karten mit der jeweiligen Route angezeigt. Diese Karten werden hierfür in der zweiten Spalte einer Tabelle eingebettet - in der ersten Spalte befinden sich Informationen zur Fahrt (siehe hierzu Kapitel 3.3 Fahrten). Hierdurch entsteht das Problem, dass sich die Karte, je nach Ausmaß der ersten Spalte, in ihrer Höhe verändert. Die Breite der Spalte ist dabei abhängig von der Bildschirmauflösung. Die Karte würde also gegebenenfalls unästhetische und unpraktische Maße annehmen. Um dies zu verhindern, wird ein Javascript eingesetzt, das die Breite der Karte ausliest und anschließend die Höhe der Karte auf eben diesen Wert setzt. Dadurch ist das Seitenverhältnis der Karte immer 1:1, unabhängig vom Inhalt der ersten Spalte.

```
var b = document.getElementById("map_canvas").offsetWidth;
var style = document.getElementById("map_canvas").getAttribute("style");
document.getElementById("map_canvas").setAttribute("style",style+"height:"+b,0);
```

Die Breite des Elements, das die Karte enthält (siehe hierzu Kapitel 2.7 Google Maps) wird ausgelesen und zwischengespeichert. Daraufhin wird dem Style-Attribut des Elements der Eintrag "height:x" hinzugefügt, wobei x der gerade ausgelesenen Breite entspricht.

### 3.1.5 Paperclip

Bei Paperclip handelt es sich um ein Gem welches nachträglich von uns angefordert wurde. Das gem wurde von | ThoughtBot<sup>1</sup> entwickelt und Paperclip dient als einfach zu handhabende Dateianhangsverwaltung für den ActiveRecord<sup>2</sup> von Ruby on Rails. Über den 'Datei auswählen' Button können wir eine Datei von unserer lokalen Festplatte zu der Datenbank hinzufügen. Dadurch, dass das Bild direkt eine eigene Spalte in der Tabelle bekommt ist ein globaler Zugriff darauf möglich. Zusätzlich wird im Model festgelegt in welchen unterschiedlichen Größen das Bild gespeichert wird. Wir haben uns für die Größen Thumbnail (100x100px), Small (200x200px) und Medium (400x400px) entschieden. Diese Größen bieten je nach 'view' einen flexiblen Zugriff auf das Bild und macht die Gestaltung der einzelnen Seiten leichter. Bilder, die nicht dem vorgegebenen Größenverhältnis entsprechen - was häufig der Fall ist - können mit den Operatoren '#' (stretchen) oder '>' (skalieren) in die gewünschte Form gebracht werden. Bei der Art der Speicherung bietet Paperclip unterschiedliche Möglichkeiten. Für gewöhnlich werden die Bilder ohne weitere Angabe automatisch in die Dateisystemstruktur des jeweiligen Hosts eingebunden. Wer mehr Speicher benötigt, was bei größeren kommerziellen Anwendungen häufig der Fall ist, und die Dateien auslagern möchte, kann das Speichersystem Amazon S3<sup>3</sup> verwenden. Diese Funktionalität wird von Paperclip unterstützt. In der nachstehenden Abbildung (Abb.: 1) wird der Speichervorgang vereinfacht dargestellt Darstellung der Verarbeitung eines neuen Bildes in der Datenbank



### 3.1.6 Photoshop

#### Banner



Abb. 6: Banner der Homepage

Die Wahl der Farben passt sich an die Farbwahl der Homepage an. Wir haben uns ausgehend von einem sandigen Farbton für Komplementär- und Hell-/Dunkelkontraste entschieden. Dadurch gewinnt der Banner an Prägnanz. Als Hintergrund wurde der Farbverlauf von einem grau/blau zu dem sehr hellen Blau gewählt. Es wird der Eindruck vermittelt als würde es sich um einen Himmel handeln (Abb.: 1).

Der Schriftzug 'mitfahrZENTRALE' - schwarz/weiß Kontrast - befindet sich vor einem Berg, welcher sich in eine flache Topologie verläuft. Auf den Ausläufern ist ein stilisiertes Auto in schwarz vor hellem Hintergrund gesetzt. Bei der Schriftart haben wir die ATRotisSemiSans verwendet. Die Schrift verfügt über prägnant geformte Einzelbuchstaben ohne Serifen, was sie perfekt für den Einsatz in Überschriften oder Bannern von

1 <http://thoughtbot.com/community/>

2 <http://ar.rubyonrails.org/>

3 <http://aws.amazon.com/de/s3/>

Internetinhalten macht. Vor allem bei großen Schriftgrößen kommt diese Typographie sehr gut zur Geltung. Durch einen leichten Schatten nach innen wird ein plastischer Effekt erzeugt.

Einbettung:

Der Banner kann als 'Header'-Element im Kopf des 'body'-Tags eingebettet werden. Per SCSS kann die Breite angepasst werden.

## Buttons

Anmelden

**Abb. 7:** Button-Desing

Das Design der Buttons (Abb.: 2) wurde an die Gestaltung der Navigationsleiste - mit CSS3 erstellt - angepasst. Ein Rechteck mit abgerundeten Kanten und einem vierstufigen vertikalen Farbgradienten von Schwarz nach Grau bildet das Grundgerüst. Durch das Anwenden des "abgeflachte Kanten und Relief"-Filters mit bestimmten Einstellungen und einer leichten "Kontur" entsteht ein plastischer Button mit leichtem Glanz.

Da auf einem Button kleinere Schriftgrößen verwendet werden wurde als Typographie die serifenlose weboptimierte Schriftart Arial verwendet. Durch den "abgeflachte Kanten und Relief"-Filter kommt ein weiterer plastizitätfördernder Effekt hinzu. Einbettung:

Wir konnten die Photoshop Buttons auf zwei unterschiedliche Weisen verwenden.

Anmelden

**Abb. 8:** alter Registrier-Submit-Button

Speichern

**Abb. 9:** Neuer Speichern-Submit-Button

```
63 = image_submit_tag "/images/button_edit.png"
```

**Abb. 10:** Submit-Button-Code

Für Formulare gibt es in Ruby vorgefertigte 'Submit'-Buttons (Abb.: 3). Mit Hilfe des `image_submit_tags` konnten diese umgestaltet werden (Abb.: 4 und 5).

Gewöhnliche 'link\_to' Anweisungen in Ruby erzeugen einen hyperlink ohne graphische Veränderung. Das Ersetzen des 'link\_to'-namens durch den `image_tag` Befehl erlaubt uns Bilder als links zu verwenden. Sollte das Bild aus irgendwelchen Gründen auch immer nicht zur Verfügung stehen wird ein alternativ Text eingeblendet (Abb.: 6 und 7).

```
61 = link_to image_tag("/images/button_back.png", :alt => "Zurück"), "javascript:history.back();"
```

**Abb. 11:** image\_tag-Code

## 3.1.7 GoogleMaps

### Einleitung

Zur Visualisierung von Fahrten, werden auf deren Detailseiten interaktive Karten mit Markierungen für Start- und Zielpunkt, sowie eingeblendeter Route angezeigt. Im Folgenden werden einige grundlegende Informationen zur Google Maps-API dargelegt und die Implementierung der Karte beschrieben.

Google bietet verschiedene Schnittstellen zum Integrieren von Karten in Webseiten an, die je nach Anwendungsgebiet sinnvoll eingesetzt werden können:

- Google Maps JavaScript-API
- Google Maps-API für Flash
- Static Maps-API
- u.a.<sup>4</sup>

### Integration der JavaScript-API in Webseiten

Bevor eine Karte auf einer Webseite angezeigt werden kann, muss ein externes JavaScript von Google geladen werden. Weiter ist es notwendig, eine Methode zum Initialisieren der Karte einzubinden<sup>5</sup>.

```
var directionDisplay;
var map;

function initialize() {
  directionsDisplay = new google.maps.DirectionsRenderer();
  //var latLng = new google.maps.LatLng(-34.397, 150.644);
  var myOptions = {
    zoom:7,
    mapTypeId: google.maps.MapTypeId.ROADMAP
  }
  map = new google.maps.Map(document.getElementById("map_canvas"), myOptions);
  directionsDisplay.setMap(map);
}
```

Der Beispielcode liefert jedoch noch keine Route mit Markierungen für Start- und Zielposition. Daher wurde dieser Code mit Hilfe weiterführender Informationen<sup>6,7</sup> dahingehend angepasst, dass es möglich ist, Start und Ziel als Koordinaten anzugeben.

```
var start = new google.maps.LatLng("#{trip.starts_at_N}", "#{trip.starts_at_E}", true);
var end = new google.maps.LatLng("#{trip.ends_at_N}", "#{trip.ends_at_E}", true);
var request = {
  origin:start,
  destination:end,
  travelMode: google.maps.DirectionsTravelMode.DRIVING
};

var directionsService = new google.maps.DirectionsService();

directionsService.route(request, function(result, status) {
  if (status == google.maps.DirectionsStatus.OK) {
    directionsDisplay.setDirections(result);
  }
});
```

4 <http://code.google.com/intl/de/apis/maps/>

5 <http://code.google.com/intl/de/apis/maps/documentation/javascript/tutorial.html>

6 <http://code.google.com/intl/de/apis/maps/documentation/javascript/services.html#DirectionsResults>

7 <http://code.google.com/intl/de/apis/maps/documentation/javascript/reference.html>

Hierbei fällt auf, dass ein Teil des eigentlich statischen JavaScript-Codes in den ersten beiden Zeilen mit Hilfe von *Ruby* dynamisch mit Werten versorgt wird (die Stellen, an denen "`#{trip...}`" auftaucht). Dabei werden die Koordinaten von Start- und Zielposition der Fahrt, die gerade aufgerufen wird, in den JavaScript-Code geschleust, um für jede Fahrt die individuelle Route anzeigen zu können.

Weiter muss noch festgelegt werden, an welcher Stelle die Karte erscheinen soll. Hierfür wird ein HTML-Div-Element mit dem Attribut `id="map_canvas"` eingefügt. Die JavaScript-API sucht automatisch nach einem solchen Div-Element, um dort die Karte einzufügen<sup>8</sup>.

Zuletzt muss noch die Initialisierungsmethode aufgerufen werden. Hierbei ist jedoch wichtig, dass die Webseite vollständig geladen wurde, bevor der Methodenaufruf geschieht - andernfalls wird die Karte möglicherweise nicht angezeigt. Dies ist dann der Fall, wenn beim asynchronen Laden der Webseite, die Methode aufgerufen wird, bevor die externe Google-JavaScript-Datei geladen wurde. Um diesen Fall zu vermeiden, muss im Body-Tag der Webseite das JavaScript-Ereignis `onload` die Methode aufrufen<sup>9</sup>. Dies gestaltete sich jedoch bei diesem Projekt schwierig, da alle Views per `%=>yield`-Befehl im gleichen Body integriert werden (siehe hierzu Kapitel 2.1 Layout). Dadurch würde jedoch das Einbinden des `onload`-Ereignisses auf jeder Unterseite erfolgen, obwohl dies nur für die Detailsansicht einer Fahrt gelten soll.

Dieses Problem wird behoben, indem per Ruby abgefragt wird, welche Unterseite gerade aufgerufen wird.

```
def gmap_onload()
  if params[:controller] == "trips" and params[:action] == "show"
    { :onload => "initialize()" }
  else
    {}
  end
end
```

Wird die gewünschte View aufgerufen, so wird eine Hashmap generiert, die dem `onload`-Ereignis den Aufruf der Methode zum Initialisieren der Karte zuweist. Wird eine andere Seite aufgerufen wird ein leerer Hash generiert. Das Ergebnis dieser Methode wird in das Body-Element eingefügt - somit wird das `onload`-Ereignis des Body nur gesetzt, wenn erwünscht.

## 3.2 Mitfahrzentrale 2.0

Dieser Abschnitt behandelt inhaltlich den Aufbau der einzelnen Views (Webseiten) mit Elementen des Web 2.0.

- Startseite
- Suche
- Fahrten
- Nachrichten
- Bewertungen
- Fahrzeuge
- Profil

### 3.2.1 Startseite

Für neue/unangemeldete Nutzer wird folgende Seite im `#main` erzeugt:

<sup>8</sup> <http://code.google.com/intl/de/apis/maps/documentation/javascript/tutorial.html>

<sup>9</sup> <http://code.google.com/intl/de/apis/maps/documentation/javascript/tutorial.html>

The screenshot shows a login form titled 'Einloggen'. It contains two input fields: 'eMail' and 'Passwort'. Below these is a checkbox labeled 'Merken'. At the bottom of the form are three buttons: 'Anmelden', 'Registrieren', and 'Passwort vergessen?'.

Abb. 12: Startseite

Der Nutzer hat die Möglichkeit sich zu registrieren, anzumelden oder wenn das Passwort abhanden gekommen ist ein Neues anzufordern.

## 3.2.2 Suche

### Übersicht über eigene Anfragen

Über den Button Suche im Hauptmenü kommt man auf die Übersicht über seine eigenen Anfragen. Übersicht

Suche nach Fahrer						Neue Anfrage
Von	Nach	Intervallstart	Intervallende	Gepäck	Anzeigen	
28195 Bremen	20354 Hamburg	20.08.2011 21:10	22.08.2011 22:10	Ja	Anzeigen	
28195 Bremen	20354 Hamburg	20.08.2011 21:10	22.08.2011 22:10	Ja	Anzeigen	

Von dort aus kommt man über den Button "Neue Anfrage" eine neue Suche anlegen und über den Button "anzeigen" auf die Detailansicht zu jeder Anfrage.

### Neue Anfrage stellen oder bestehende Anfrage bearbeiten

The screenshot shows a form titled 'Neue Anfrage'. It has several fields: 'Start\*' (text input), 'Ziel\*' (text input), 'Abfahrtsintervall:' (date and time pickers), 'Gepäck' (checkbox), and 'Anmerkungen' (text area). There are two 'Radius' dropdown menus, both set to '5km'. At the bottom are buttons for 'Zurück' and 'Erstellen'. A note at the bottom says 'Mit \* gekennzeichnete Felder sind Pflichtfelder'.

Abb. 14: Neue Anfrage

Beim anlegen einer neuen Anfrage kann man in die Felder Start- und Zielort einen absolut beliebigen String eintragen. Dieser wird im Controller ausgewertet und auf jeweils drei Datenfelder (PLZ, Ort, Straße) verteilt.

## Detailansicht einer eigenen Anfrage

Meine Anfrage	
<b>Anfrage</b>	
<b>Startadresse:</b>	28195 Bremen Wachtstraße 24 (5km Umkreis)
<b>Zieladresse:</b>	20354 Hamburg Große Bleichen 13 (5km Umkreis)
<b>Startzeit:</b>	20.08.2011 21:10
<b>Endzeit:</b>	22.08.2011 22:10
<b>Fahrzeit*:</b>	01:16
<b>Gepäck:</b>	Ja
<b>Anmerkungen:</b>	

\*Angaben ohne Gewähr

Abb. 15: Detailansicht

Zu jeder Anfrage werden unten die dazu passenden Fahrten angezeigt. Diese sind vom Model in absteigender Reihenfolge von sehr gut bis schlecht passend sortiert worden.

### 3.2.3 Fahrten

#### Vorwort

Der folgende Abschnitt behandelt den Aufbau und die Funktion der Unterseiten der Fahrten. Hierzu gehören die Übersichtsseite über alle Fahrten des jeweiligen Benutzers, die Formularseite zum Anlegen einer neuen Fahrt und die Detailansichten der Fahrten.

Besondere Aufmerksamkeit ruht hierbei auf dem schematischen Aufbau und der bedingten Ausgabe von Inhalten. Auf den besagten Seiten verwendete JavaScript-Funktionalitäten werden in einem gesonderten Abschnitt dieser Dokumentation genauer erläutert (siehe Kapitel 2.4 Javascript).

#### Index

Die Indexseite der Fahrten ist gleichzeitig Übersicht über alle eigenen Fahrten und Startseite nach dem Login in das Benutzerkonto.

Unterhalb eines Buttons, zum Anlegen einer neuen Fahrt, werden die Fahrten des Benutzers in vier untereinander angeordneten Tabellen kategorisiert:

- Tabelle 1:  
Enthält die noch ausstehenden Fahrten, die der Benutzer als Fahrer antritt.
- Tabelle 2:  
Enthält alle beendeten Fahrten, die der Benutzer als Fahrer angetreten hat.
- Tabelle 3:  
Hier werden alle noch ausstehenden Fahrten aufgelistet, die der Benutzer als Mitfahrer antreten wird.
- Tabelle 4:  
Die letzte Tabelle enthält, analog zur zweiten, alle beendeten Fahrten, die der Benutzer als Mitfahrer antrat.

Die Tabellen werden jeweils chronologisch absteigend sortiert vom Controller übergeben. Falls noch keine zur jeweiligen Tabelle passende Fahrt existiert, wird dies mittels einer einzeiligen Tabelle angemerkt. Andernfalls wird je Fahrt eine Zeile mit groben Informationen und einem Link zur Detailansicht der Fahrt angezeigt. Das folgende Schema veranschaulicht den Aufbau der Index-Seite:



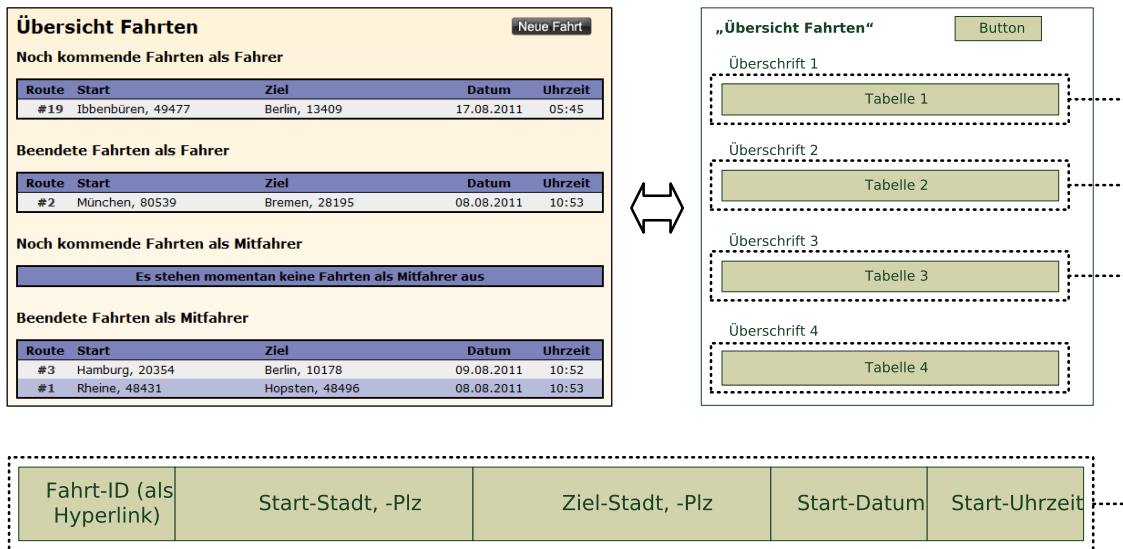


Abb. 16: Schema der Indexseite

## Neue Fahrt erstellen

Wie im vorherigen Abschnitt beschrieben, befindet sich auf der Übersichtsseite der Fahrten außerdem ein Button mit der Aufschrift *Neue Fahrt*. Ein Klick auf diesen Button leitet den Benutzer zu einer Formularseite, mit deren Hilfe eine neue Fahrt als Fahrer erstellt werden kann. Hierbei hat der Benutzer die Möglichkeit, den Start- und Zielort in Form von Freitext einzugeben.

**Neue Fahrt erstellen**

Startadresse\*: Große Straße 11, 49477 Ibbenbüren

Zieladresse\*: 13409 Berlin, Sommerstraße 21

Abfahrtszeit: 17 . 08 . 2011 - 05 : 45

Fahrzeug: Mercedes (2 Sitzplätze)

Sitzplätze: keine Änderung

Gepäck:

Hinweise: Ich fahre keine Umwege, um jemanden abzuholen. Wer zur besagten Zeit nicht am Treffpunkt ist, wird nicht mitgenommen.  
Noch 43 Zeichen

Zurück Erstellen

Mit \* gekennzeichnete Felder sind Pflichtfelder

Abb. 17: Aufbau des Formulars *Neue Fahrt*

Die Zeitangaben für den Start der Fahrt erfolgen mit Hilfe von Dropdown-Listen (*Select-Tag* in HTML). Die voreingestellte Zeit entspricht der Zeit, zu der der Benutzer das Formular aufruft. Die Liste für Minuten ist in fünf Schritten aufgelöst, was eine benutzerfreundliche Verwendung ermöglicht: die Liste wird dadurch kleiner. Außerdem sind wir davon ausgegangen, dass es schwer möglich ist, im Straßenverkehr bis auf die Minute genau zu planen. Das Voreinstellen der aktuellen Zeit geschieht per Ruby, bevor das Formular an den Webbrowser geschickt wird. Hierzu werden die passenden Werte der Dropdown-Listen via HTML als Startwerte markiert. Dies wird beispielhaft an der Dropdown-Liste für die Minuten veranschaulicht:

```
%select{:name => "start_minute", :required => "required"}
- 12.times do |i|
- if i*5 < 10
- if i*5 > Time.now.min and (i-1)*5 <= Time.now.min
%option{:value => "0"+(i*5).to_s, :selected => "selected"}= "0#{i*5}"
- else
%option{:value => "0"+(i*5).to_s}= "0#{i*5}"
```

```

- else
- if i*5 > Time.now.min and (i-1)*5 <= Time.now.min
  %option{:value => (i*5).to_s, :selected => "selected"}= "#{i*5}"
- else
  %option{:value => (i*5).to_s}= "#{i*5}"

```

Die Minuten-Liste wird mit zwölf Werten gefüllt, von 0 bis 55. Bevor ein Wert ins HTML-Formular ausgegeben wird, wird geprüft, ob die aktuelle Uhrzeit (in diesem Fall der Wert der Minuten) kleiner ist, als der auszugebene Wert, und größer ist, als der vorher ausgegebene Wert (also ob sich der aktuelle Wert für die Minuten zwischen zwei auszugebenen Werten liegt). Ist dies der Fall, wird der Wert ins HTML-Dokument geschrieben und als *ausgewählt* markiert. Auf diese Weise wird als voreingestellte Zeit immer der nächste fünf-Minuten-Schritt ausgewählt. Außerdem wird, um die Ausgabe ästhetisch ansehlicher zu gestalten, bei auszugebenen Werten kleiner zehn eine "0" vorangestellt.

Ruby on Rails bietet für einfache Formulare, deren Werte problemlos direkt in ein Datenmodell geschrieben werden können, eine solche Funktionalität an. Jedoch ist es hier nicht möglich, diese Funktionalität zu nutzen, da die Kommunikation mit dem Controller der Fahrten per POST-Parameter erfolgen muss, damit einige der eingegebenen Daten weiterverarbeitet werden können, bevor sie in das passende Datenmodell geschrieben werden.

Als nächstes erfolgt die Wahl des zu verwendenden Fahrzeuges via Dropdown-Liste. Hier sind alle vom Benutzer angegebenen Fahrzeuge in der Form *Modellname (Anzahl Sitzplätze)* aufgelistet. Darunter kann man wiederum per Dropdown-Liste die Anzahl an Sitzplätzen für diese Fahrt variieren. Hintergrund hierbei ist, dass der Fahrer möglicherweise von sich aus schon mögliche Plätze belegt oder nur eine gewisse Anzahl an Mitfahrern wünscht. Die neue Zahl an Sitzplätzen kann nur kleiner oder gleich der für das Auto angegebenen Sitzplätze sein. Dies wird per JavaScript gewährleistet. Wird ein anderes Fahrzeug gewählt, wird die Dropdown-Liste für die Anzahl an Sitzplätzen entsprechend angepasst. Weiter kann der Fahrer angeben, ob die Mitnahme von Gepäck möglich ist. Dies geschieht per Checkbox.

Das letzte Eingabefeld bietet dem Fahrer abschließend die Möglichkeit, einen kleinen Hinweistext zu hinterlassen. Dabei wird beständig angezeigt, wieviele Zeichen hierfür noch verwendet werden dürfen.

## Detailansicht für Fahrten

Dieser Abschnitt der Dokumentation befasst sich letztlich mit der Detailansicht der Fahrten.

The screenshot shows a web interface for trip details. At the top right are buttons for 'Löschen' and 'Zurück'. The main content is divided into two columns. The left column contains a table with trip details and a list of participants. The right column contains a map of the route from Ibberbüren to Berlin.

Fahrtinformationen	
Abfahrt:	Große Straße 11 , 49477 Ibberbüren 17.08.2011 05:45
Ankunft:	Sommerstraße 21 , 13409 Berlin
Fahrer:	Michael Blömer 5,0 (1)
Auto:	Mercedes (Kofferraum verfügbar aber kein Platz)
Gepäck:	Es kann kein Gepäck mitgenommen werden
Hinweise:	Ich fahre keine Umwege, um jemanden abzuholen. Wer zur besagten Zeit nicht am Treffpunkt ist, wird nicht mitgenommen.

Mitfahrer (1/2)	Bewertung		
Sascha Ebelt	3,0 (0)	Nachricht schicken	Austragen

Bewerber (1)	Bewertung		
Benutzername	Rating		
Uwe Fischer	6,0 (1)	Nachricht schicken	annehmen ablehnen

Entfernung: 445km  
ca. Dauer: 3 Stunden 59 Minuten

Abb. 18: Die Detailansicht einer Fahrt aus Sicht des Fahrers

Die Seite besteht aus einer Tabelle mit zwei Spalten. Die erste Spalte beinhaltet Informationen für die Fahrt, Mitfahrer und Bewerber, die jeweils wiederum einer Tabelle entsprechen. Die zweite Spalte enthält eine Karte der Route, Länge der Strecke und geschätzte Fahrzeit. Überhalb der umschließenden Tabelle befindet sich zusätzlich eine Überschrift, ein Button zum Löschen der Fahrt und einer zum Zurückkehren auf die vorherige Seite.

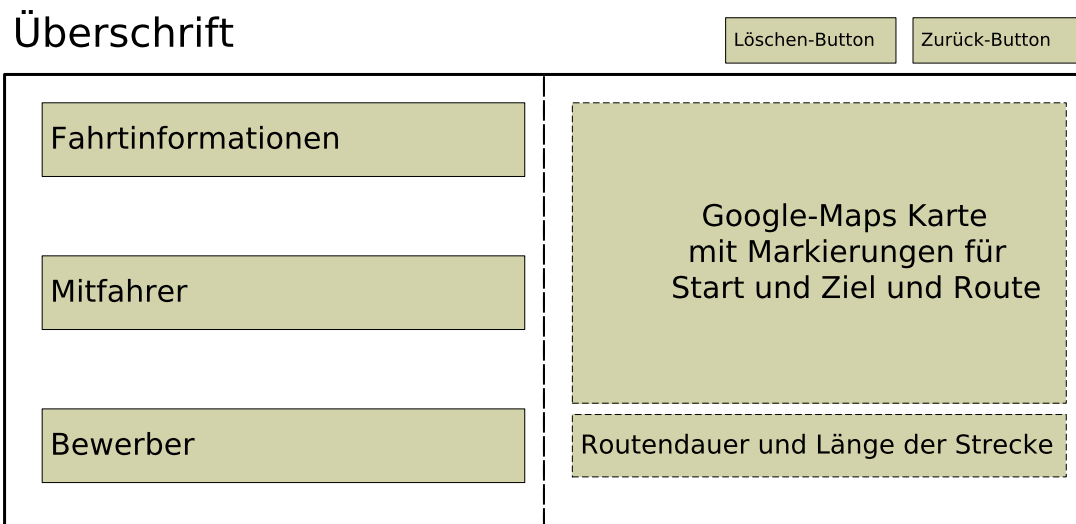


Abb. 19: Schema der Detailansicht von Fahrten

Welche Informationen auf dieser Seite angezeigt werden, hängt vom Status des Benutzers ab. Hierzu liefert der Controller der Trips eine Variable mit Informationen hierüber an die View. Diese Variable wiederum kann mit Status-Variablen verglichen werden, um auf diese Weise eine statusabhängige Ausgabe zu erzeugen. Folgendes Schema stellt dar, wie die Sichtbarkeit organisiert ist:

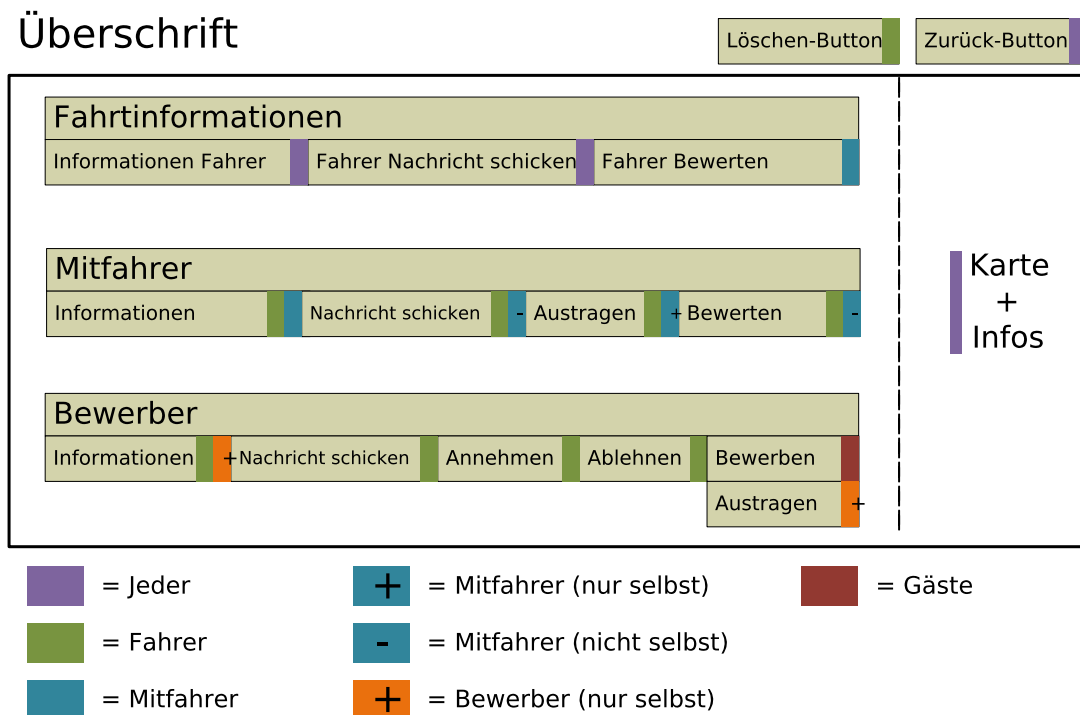


Abb. 20: Statusabhängige Anzeige von Informationen

Jeder Benutzer darf Informationen über den Fahrer anschauen, ihm eine Nachricht schreiben, die Karte sehen und den Zurück-Button benutzen. Das Bewerten des Fahrers ist jedoch ausschließlich den Mitfahrern vorbehalten. Der entsprechende Link wird auch erst nach Ablauf der Fahrt angezeigt. Um herauszufinden, ob die Fahrt bereits vorbei ist, ist im Model eine entsprechende Methode implementiert. Weiter dürfen die Mitfahrer sich gegenseitig Nachrichten schicken und bewerten (ebenfalls nach Fahrtende) und auch vom Fahrer bewertet werden. Aus der Fahrt austragen kann sich jeder Mitfahrer selbst. Ebenso kann der Fahrer jeden Mitfahrer von

der Fahrt ausschließen. Weiter werden Links zu den Profilen der Mitfahrer aus Datenschutzgründen nur für den Fahrer und die Mitfahrer angezeigt. Gäste und Bewerber können daher nicht unmittelbar die Identität der vorhandenen Mitfahrer ermitteln.

Die Bewerbungen in der entsprechenden Tabelle werden nur dem Fahrer und dem jeweiligen Bewerber angezeigt. Der Datensatz eines Bewerbers wird ihm jedoch ebenfalls angezeigt, zusammen mit einem Link, um die Bewerbung zurückzunehmen. Hat sich ein Benutzer noch nicht für eine Fahrt beworben, so wird im ein entsprechender Link zum Bewerben angezeigt. Der Fahrer kann außerdem über entsprechende Links Bewerbungen annehmen, ablehnen und den Bewerbern eine Nachricht schreiben.

### 3.2.4 Nachrichten

Die Darstellung von Nachrichten geschieht getrennt nach empfangenen (index) und gesendeten (outbox) Nachrichten. Die einzelne Nachricht wird durch eine farblich hervorgehobene Tabelle dargestellt. In der Kopfzeile steht der Absender bzw. Empfänger mit Verlinkung zu dessen Profil und das Datum der Nachricht. Darunter befindet sich der hervorgehobene Betreff und darunter der Inhalt der Nachricht. Unter jeder Nachricht befinden sich Buttons für Antworten bzw. neue Nachricht und zum Löschen der Nachricht. Bei neuen Nachrichten wird dies neben dem Absender in der Kopfzeile kenntlich gemacht, außerdem wird die Zahl neuer Nachrichten in der Navigationsleiste angezeigt. Vom System erstellte Nachrichten können interne Links enthalten. Da Ruby on Rails automatisch HTML escaped, werden diese in spezieller Syntax (`[[Link|Linktext]]`) gespeichert und müssen beim Erzeugen der Seite besonders geparkt werden.

Im Formular zum Erstellen von Nachrichten ist der Empfänger bereits festgelegt, wird aber zusätzlich (entsprechend verlinkt) angezeigt. Das Feld für den Betreff ist bei Antworten bereits mit einem entsprechenden Text initialisiert, in ein Textfeld kann die Nachricht eingetragen werden.

Wo sinnvoll, sind entsprechende Links zur vorherigen und/oder zu verwandten Seiten intuitiv platziert.

### 3.2.5 Bewertungen

Die Darstellung von Bewertungen geschieht getrennt nach empfangenen (show) und erstellten (index) Bewertungen. Die einzelne Bewertung wird durch eine farblich hervorgehobene Tabelle dargestellt. In der Kopfzeile steht der Absender bzw. Empfänger mit Verlinkung zu dessen Profil und das Datum der Bewertung. Darunter befindet sich der hervorgehobene Betreff mit Note und darunter ein Kommentar. Bei neuen Bewertungen wird dies neben dem Absender in der Kopfzeile kenntlich gemacht, außerdem wird die Zahl neuer Bewertungen in der Navigationsleiste angezeigt. Der Betreff enthält den internen Link zur entsprechenden Fahrt. Da Ruby on Rails automatisch HTML escaped, werden dieser in spezieller Syntax (`[[Link|Linktext]]`) gespeichert und muss beim Erzeugen der Seite besonders geparkt werden.

Erhaltenen Bewertungen werden nach Bewertungen für Fahrer und Mitfahrer getrennt und es wird jeweils die durchschnittliche Benotung angezeigt. Auf der Seite der erstellten Bewertungen werden, wenn vorhanden, noch offenen Bewertungen angezeigt, jeweils mit Link zum entsprechenden Formular.

Im Formular zum Erstellen von Bewertungen sind Empfänger und Betreff (Fahrt) bereits festgelegt, werden aber zusätzlich (entsprechend verlinkt) angezeigt. Über ein Drop-Down-Menü kann die Note ausgewählt werden und in ein Textfeld ein Kommentar eingetragen werden.

Wo sinnvoll, sind entsprechende Links zur vorherigen und/oder zu verwandten Seiten intuitiv platziert.

### 3.2.6 Fahrzeuge

Über den Link 'Fahrzeuge' auf der Navigationsleiste wird der Nutzer zunächst zu einer groben Übersicht aller bisher angelegten Fahrzeuge weitergeleitet.



Fahrzeugdetails						Hinzufügen
Bild	Modell	Sitzplätze	Kosten pro 100 km	Raucher	Kennzeichen	
	BMW X3	5	8,9	Ja	OS-NA-9999	Löschen
	VW Käfer	3	8,9	Ja	WST-CJ-229	Löschen

Abb. 21: Tabelle aller Fahrzeuge

Der Nutzer bekommt neben dem Miniaturbild die wichtigsten Informationen zusammengefasst (Abb.: 1). In dieser Darstellung kann der Nutzer auf eine Detailansicht des Fahrzeuges gelangen, das Fahrzeug aus der Liste löschen oder ein Neues anlegen.


Fahrzeugdetails		Zurück	Editieren														
<table border="1"> <tr><th colspan="2">Details</th></tr> <tr><td>Modell</td><td>BMW X3</td></tr> <tr><td>Sitzplätze</td><td>5</td></tr> <tr><td>Kosten: € pro 100km</td><td>8,9</td></tr> <tr><td>Raucherwagen</td><td>ja</td></tr> <tr><td>Nummernschild</td><td>OS-NA-9999</td></tr> <tr><td>Beschreibung</td><td>Dieses Auto ist der Hammer</td></tr> </table>		Details		Modell	BMW X3	Sitzplätze	5	Kosten: € pro 100km	8,9	Raucherwagen	ja	Nummernschild	OS-NA-9999	Beschreibung	Dieses Auto ist der Hammer		
Details																	
Modell	BMW X3																
Sitzplätze	5																
Kosten: € pro 100km	8,9																
Raucherwagen	ja																
Nummernschild	OS-NA-9999																
Beschreibung	Dieses Auto ist der Hammer																

Abb. 22: Detailansicht eines Fahrzeuges

In der 'show-view' der Fahrzeuge bekommt der Aufrufer der Seite alle Informationen zu dem Fahrzeug geliefert. Dazu gehört eine angepasste Version (":medium") des Bildes sowie die Beschreibung zu dem Auto. Über den Button 'Editieren' können die Informationen nach belieben verändert werden - natürlich nur solange die Validations eingehalten werden.

Neues Fahrzeug	
Sitzplätze*	<input type="text" value="5"/>
Modell*	<input type="text"/>
Kennzeichen*	<input type="text"/>
Kosten: € pro 100km*	<input type="text" value="8,9"/>
Raucher	<input type="checkbox"/>
Beschreibung	<input type="text" value="Großer Kofferraum, Keine Türen"/>
Noch 160 Zeichen	
Bild hochladen	<input type="button" value="Datei auswählen"/> <input type="button" value="Keine ausgewählt"/>
<input type="button" value="Zurück"/> <input type="button" value="Hinzufügen"/>	

Mit \* gekennzeichnete Felder sind Pflichtfelder

Abb. 23: Neues Fahrzeug erstellen

Wer ein neues Fahrzeug erstellt (leere Formularfelder) oder bearbeitet (vorherige Werte voreingestellt) hat die Möglichkeit über das von uns implementierte gem Paperclip ein Bild von der lokalen Festplatte hinzuzufügen. Über ein Javascript wird die Einhaltung des Textes in dem Feld 'Beschreibung' geregelt und dem Nutzer werden die noch zur Verfügung stehenden Zeichen angezeigt. Mit dem Attribut ':placeholder' werden Platzhalter in leeren Formularfelder eingetragen um dem Nutzer Vorschläge für die Art und Form des Inhaltes zu geben.

## 3.2.7 Profil

### Ein Profil erstellen

Beim erstellen eines Profils kann man als erstes auswählen, ob man ein privater oder ein geschäftlicher Kunde ist. Wenn man die Checkbox nicht anklickt, so bekommt man folgendes Formular zum ausfüllen:

Abb. 24: Registrierung

Klickt man jedoch die Checkbox an, so wird das onclick Ereignis ausgelöst und der folgende JavaScriptCode aufgerufen:

```

: javascript
function formtest(){
  if (document.getElementById("business").checked==true){
    document.getElementById("gebdatumLabel").style.display='none'; //blendet aus
    document.getElementById("gebdatumSelect").style.display='none';
    document.getElementById("geschlechtLabel").style.display='none';
    document.getElementById("geschlechtSelect").style.display='none';
  } else {
    document.getElementById("gebdatumLabel").style.display='block'; //blendet ein
    document.getElementById("gebdatumSelect").style.display='block';
    document.getElementById("geschlechtLabel").style.display='block';
    document.getElementById("geschlechtSelect").style.display='block';
  }
}

```

Dieser blendet bei ankreuzen der Box die Tabellenkästchen für die Auswahl des Geschlechts und des Geburtsdatums, sowie die Labels die in derselben Reihe stehen aus.

## Profilansicht

Wenn man im Hauptmenü auf den Button Profil klickt kommt man auf die Ansicht seines Profils: Ansicht

Profildetails	
Nutzername:	Erik Mustermann
Email:	emuster@uos.de
Geschlecht:	Maennlich
Adresse:	Musterstraße 11
PLZ:	12345
Ort:	Musterhausen
Geburtsdatum:	1997-08-19
InstantMessenger:	
Telefon:	

Erhaltene Bewertungen

Hier wird entweder das ausgewählte Bild angezeigt, oder ein vorgegebenes Standardbild (männlich, weiblich, Firma). Außerdem wird bei der Ansicht eines Firmenprofils wiederum Geschlecht und Geburtsdatum nicht mit angezeigt. Zudem werden Standardmäßig bei der Betrachtung von fremden Profilen alles außer Name, Geschlecht und Geburtsdatum versteckt. Von jedem Profil aus kommt man auf die zu dieser Person gehörigen Bewertungen. Außerdem kann man, wenn man sich auf einem fremden Profil befindet, über das anklicken des

Buttons "ignorieren", festlegen, das jegliche Informationen über diese Person zukünftig nicht mehr angezeigt werden.

## Profil bearbeiten

Wenn man sich auf seinem eigenen Profil befindet kommt man über das anklicken des Buttons "bearbeiten" auf die Seite zum bearbeiten seines Profils

**Profil bearbeiten**

Mit \* gekennzeichnete Felder müssen ausgefüllt sein

Name*	<input type="text" value="Erik Mustermann"/>	
E-Mail*	<input type="text" value="emuste@uos.de"/>	
Neues Passwort	<input type="password"/>	
Neues Passwort wiederholen	<input type="password"/>	
Geburtsdatum*	<input type="text" value="19"/> <input type="text" value="8"/> <input type="text" value="1987"/>	sichtbar <input type="checkbox"/>
Geschlecht	<input type="text" value="Männlich"/>	
Strasse und Hausnummer*	<input type="text" value="Musterstraße 11"/>	sichtbar <input type="checkbox"/>
PLZ*	<input type="text" value="12345"/>	sichtbar <input type="checkbox"/>
Ort*	<input type="text" value="Musterhausen"/>	sichtbar <input type="checkbox"/>
Telefon	<input type="text"/>	sichtbar <input type="checkbox"/>
InstantMessengerDaten	<input type="text"/>	sichtbar <input type="checkbox"/>
Bild hochladen	<input type="button" value="Datei auswählen"/> <input type="button" value="Keine ausgewählt"/>	
Passwort*	<input type="password"/>	

**Abb. 26:** Bearbeiten

Hier kann man festlegen was andere beim ansehen des Profils sehen dürfen und auch eigene Bilder hochladen. Diese kommen dann über Paperclip in die Datenbank. Außerdem kann man hier auch den gesamten Account löschen.