smartlearn
virtual learning. real success.

MODUL 133

TEIL 3: MVC – MODEL VIEW CONTROLLER: MODEL (DATENBANK) ERSTELLEN UND IN VIEW ANZEIGEN

Ralph Maurer

Inhaltsverzeichnis AB133-03

Modul 133: Webapplikationen mit Session-Handling realisieren

Teil 3: MVC – Model View Controller: Model (Datenbank) erstellen und in View anzeigen

MVC – Model View Controller in Rails

MVC ist ein Software-Pattern (zu Deutsch *Entwurfsmuster*), das aus den drei Komponenten *Model* (Datenmodell), *View* (Ansicht, GUI oder Präsentation) und *Controller* (Benutzerinteraktion und Programmsteuerung) besteht. Die Komponenten sind strikte voneinander getrennt, die Trennung erhöht die Flexibilität, Modularität und Wiederverwendbarkeit.

In Übereinstimmung mit dem MVC Software-Pattern ist das Rails-Framework hauptsächlich aus den drei Programmbibliotheken ActiveRecord, ActionView und ActionController aufgebaut, die ggf. auch unabhängig voneinander eingesetzt werden können.

ActiveRecord (Model)

Das Modul ActiveRecord entspricht dem Model im MVC-Paradigma. Seine Objekte repräsentieren die Anwendungsobjekte. Das Model liefert die darzustellenden Daten unabhängig vom Erscheinungsbild. Woher die Daten kommen, ob zum Beispiel aus einer relationalen Datenbank oder einem Web-API, spielt keine Rolle. Das Model kennt weder die View noch den Controller. Das Model weiss also nicht, ob, wann, wie und wie oft es dargestellt und verändert wird.

Die Hauptaufgabe vom Modul ActiveRecord besteht darin, das Mapping vom objektorientierten Klassenmodell zum relationalen Datenbankmodell und umgekehrt herzustellen.

ActionView (View)

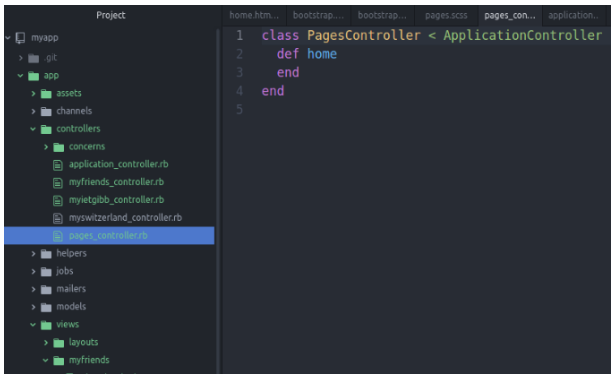
Das Modul ActionView dient zur Erstellung von Templates für die Darstellung. Die Daten, die der Controller liefert, werden bei der Ausgabe als dynamische Inhalte in die Templates eingebunden. Die View kann die Daten in unterschiedlichen Formaten ausgeben (HTML, XML, usw.).

ActionController (Controller)

Das Modul ActionController koordiniert die Interaktion zwischen Anwendung und Benutzer, indem es den Datenfluss zwischen Model und View steuert und kontrolliert. Es stellt zum Beispiel Möglichkeiten zur Verfügung, Daten des Models zu manipulieren und diese zur Darstellung an die View weiterzuleiten. Im Controller werden eintreffende Anfragen an die Actions der Controller-Klassen übergeben und der Controller übernimmt das Session-Management und das Caching.

ActionPack (View und Controller)

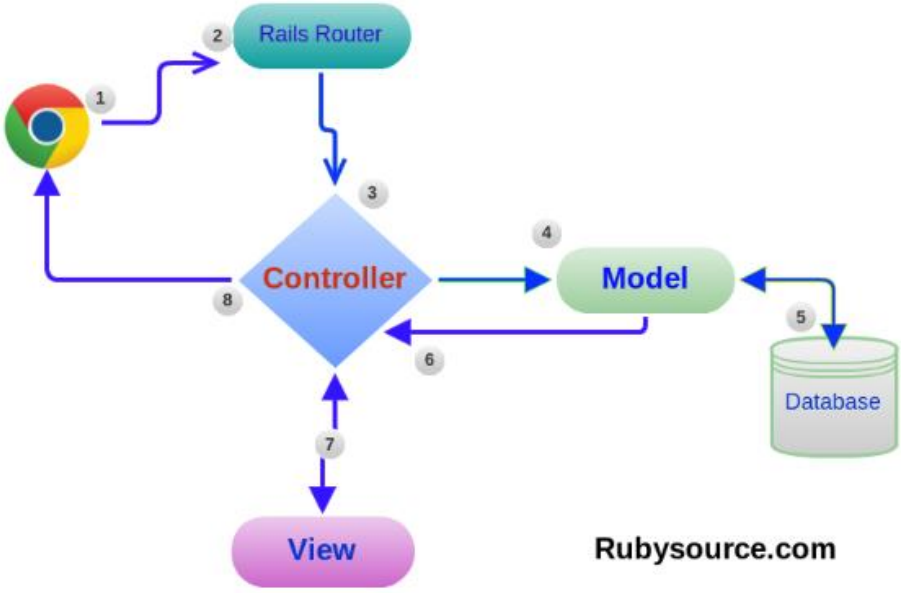
Die beiden Module ActionView und ActionController werden im Modul ActionPack zusammengefasst. Das bedeutet jedoch nicht, dass innerhalb von Rails der Controller und die View miteinander vermischt werden, sondern diese Gruppierung verdeutlicht nur, dass Controller und View eng ineinandergreifen und deshalb typischerweise gemeinsam genutzt werden.

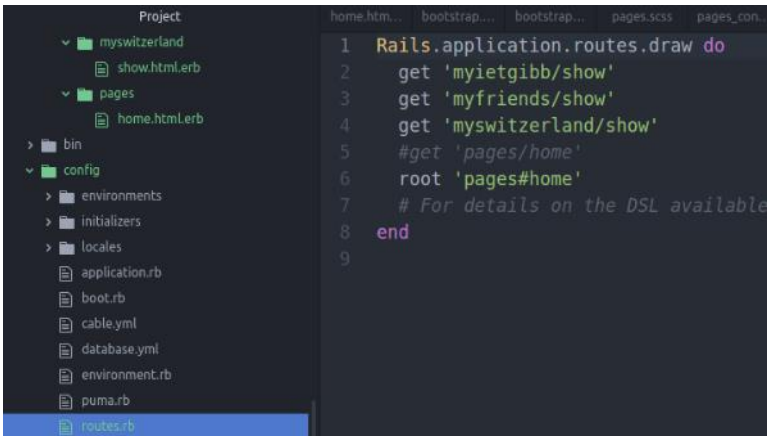


Wir haben bereits in AB133-02 Views und Controller angewendet:

1. View
   In unserer App haben wir im Ordner `myapp/app/views/pages` Seiten mit HTML Quelltext erstellt.
2. Controller
   Im Ordner `myapp/app/controllers` wurde bei der Erstellung statischer Seite jeweils eine Klasse generiert, die eine Methode mit dem Namen der View beinhaltet (z.B. *home* für unsere Startseite).

Betrachten wir eine Grafik aus http://Rubysource.com





(1) Der Benutzer macht eine Browseranfrage (z.B. Aufruf einer bestimmten URL bzw. Seite).

(2) Der Rails-Router, den wir auch schon in AB133-02 erwähnt haben, liegt im Ordner `myapp/config/routes.rb`

Sobald eine Route angefragt wird, ruft `Rails.application.routes.draw` die Methode `get` auf, welche mit dem entsprechenden Parameter die Anfrage weiterleitet (z.B. `myietgibb/show`).

(3) Die Anfrage geht weiter an den dazugehörigen Controller, welcher entweder eine View direkt anzeigt (7) oder über das Model (4) Daten aus einer Datenbank (5) abfragt oder Daten in eine Datenbank eingibt.

(6) Via Model leitet der Controller leitet die Daten an eine dazugehörige, eigenständige View (7).

Da wir nun einiges über den Controller wissen, wenden wir uns jetzt dem Model und einer dazugehörigen View zu.

Model

Man kann unterschiedliche Datenbanken an eine Rails-App anbinden: MySQL, Postgres, MongoDB, usw. Die aktuelle Standard-Datenbank ist sqlite3, diese wird benutzt, wenn nichts Anderes definiert wird und kein zusätzlicher Installationsaufwand betrieben werden soll. Wir belassen es dabei, die verwendete Datenbank ist im ActiveRecord.

Rails bietet eine sehr einfache Art und Weise, mit einer Datenbank zu arbeiten. Wir erinnern uns an das Kommando zur Erstellung einer neuen Seite:

```
$ rails generate controller <verzeichnis> <seite1> <seite2> …
  └──┘ └──────┘ └────────┘ └─────────────┘ └──────────────────┘
   Rails  Parameter    Typ      Verzeichnis mit     Liste der Seiten
 Kommando «erzeugen» «controller» statischen Seiten   im Verzeichnis
```

Ähnlich wie bei einem Controller können Tabellen erstellt werden:

```
$ rails generate model <Tabelle> <Attr1>:Datentyp <Attr2>:Datentyp …
  └──┘ └──────┘ └───┘ └─────────┘ └──────────────────────────────────┘
   Rails  Parameter  Typ  Tabellenname      Attribute (Spalten) mit Datentyp
 Kommando «erzeugen» «model»
```

Datentypen in Rails mit sqlite3

Identifikation	Binär	Zahlen	Zeichen	Datum/Zeit

`primary_key`
(Rails generiert automatisch einen Primärschlüssel)
	`boolean`
	`binary`
		`integer`
		`decimal`
		`float`
			`string`
			`text`
				`date`
				`time`
				`datetime`
				`timestamp`

Model "User" erstellen

Öffnen Sie ein Terminal und gehen Sie ins Verzeichnis `~/workspace/myapp`.

Wir wollen folgende Tabelle erstellen:
`user(name: string, email:string, age:integer)`. Im Rails Model sind alle Attribute automatisch vom Datentypen `string`. Deshalb muss beim Erstellen `string` nicht explizit angegeben werden. Wir brauchen auch keinen Primärschlüssel, dieser wird automatisch erstellt und verwaltet.

Für die Tabelle `user(name: string, email:string, age:integer)` lautet der Befehl zur Erstellung wie folgt:

```
$ rails generate model user name email age:integer
```

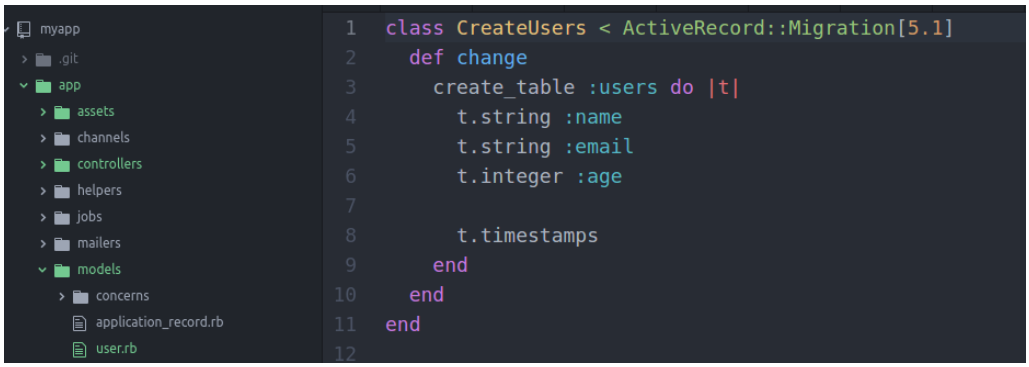
Alternativ können die Datentypen aller Attribute angegeben werden:

```
$ rails generate model user name:string email:string age:integer
```

Das Kommando speichert die Tabelle nicht direkt in die Datenbank, sondern erstellt die benötigten Vorbereitungen in Ruby-Skripts in Form von Ruby-Code:

1. Skript: `myapp/db/migrate/20180427125551_create_users.rb`
   (der 1. Teil des Sktiptnamens stellt Datum und Zeit der Erstellung dar)

   In diesem Skript ist das Erstellen der Tabelle mit Ruby-Code festgehalten:



   Es wird eine Klasse *CreateUsers* definiert, die eine Methode *change* beinhaltet. Die Methode *change* wird unsere Tabelle *user* anlegen.

2. Skript: `myapp/app/models/user.rb`

   Es wird die Klasse *User* erzeugt, die später wiederverwendet werden kann.

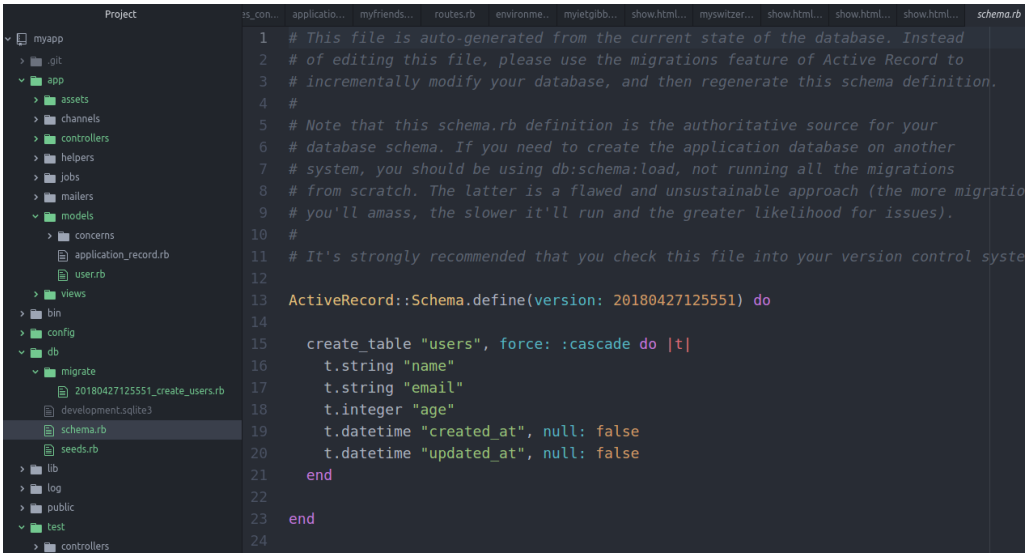
Jetzt ist es an der Zeit, die Tabelle *user* zu erstellen. Mit dem folgenden Befehl werden alle ausstehenden Anpassungen an der Datenstruktur durchgeführt:

```
$ rails db:migrate
```

Bei jeder strukturellen Änderung muss das Kommando wiederholt werden!

3. Skript: `myapp/db/schema.rb`

   Hier werden alle Tabellenstrukturen (Schemas) im ActiveRecord aufgelistet und als `create_table` Ruby-Befehle festgehalten. Das heisst, dass wir später, z.B. beim Wiederherstellen der App, nicht alle Tabellen neu machen müssen. Wir können mit dem Schema alle Tabelle wieder erzeugen.



Mit der Rails-Console Daten vom Model "User" bearbeiten.

Mit dem folgenden Kommando starten Sie die Rails-Console:

```
$ rails console
```

Alternativ können Sie den Shortcut c für den Parameter `console` anwenden:

```
$ rails c
```

Die Rails-Console ist ein interaktiver Ruby-Modus, d.h. man kann Ruby-Kommandos und Programmcodes direkt in der Konsole ausführen. Dies eignet sich besonders gut für die Arbeit mit dem Model. Es empfiehlt sich, die App-Aktivitäten aufzuzeichnen.

Sie können dazu das folgende Kommando in die Rails-Console eingeben:

```
2.6.3 :001 > ActiveRecord::Base.logger = Logger.new STDOUT
```

Damit wir auf unser Model zugreifen können, benötigen wir den Befehl

```
2.6.3 :002 > User.connection
```

Achten Sie darauf, das *U* von *User* grosszuschreiben oder es wird eine Fehlermeldung ausgegeben (Name der Klasse *User* in der Datei `user.rb`).

Sobald wir die Verbindung zum Model *user* etabliert haben, können wir einige einfache Ruby Abfragen ausprobieren.

Ausgabe der Tabellenstruktur:

```
2.6.3 :002 > User
=> User(id: integer, name: string, email: string, age: integer,
created_at: datetime, updated_at: datetime)
```

Daten im Model "User" erfassen: Variante 1

Erfassen Sie mindestens drei Personen in der Tabelle *user*:

```
2.6.3 :003 > User.create(name: 'Ralph Maurer', email:
'ralph.maurer@iet-gibb.ch', age: 22)

2.6.3 :004 > User.create(name: 'Michael Abplanalp', email:
'michael.abplanalp@iet-gibb.ch', age: 22)

2.6.3 :005 > User.create(name: 'Christian Schweingruber', email:
'christian.schweingruber@iet-gibb.ch', age: 23)
```

Im Gegensatz zu SQL wird in Rails eine objektorientierte Schreibweise zur Erfassung von Daten verwendet.

Die Methode `create` der Klasse *User* wird aufgerufen, erzeugt ein Objekt und speichert dieses in die Datenbank. Die Methode `create` bekommt drei Parameter, nämlich:

```
name: 'Christian Schweingruber',
email: 'christian.schweingruber@iet-gibb.ch',
age: 23
```

Zuerst steht jeweils das Tabellenattribut, gefolgt vom Doppelpunkt ":" und dem Wert, der dem Attribut zugewiesen wird.

Wir lernen weiter unten eine bessere Variante zum Erfassen von Daten kennen!

Daten aus dem Model "User" abfragen

Probieren wir weitere, coole Ruby-Kommandos wie das Ausgeben des ersten und letzten Datensatzes:

```
2.6.3 :006 > User.first
2.6.3 :007 > User.last
```

Die Methoden `first` und `last` erzeugen jeweils ein Objekt, das zurückgegeben wird.

Datensätze können auch über den Primärschlüssel gefunden werden:

```
2.6.3 :008 > User.find(1)
```

Gute Suchfilter sind Werte von Attributen, hierzu generiert Rails pro Attribut eine eigene Methode:

```
2.6.3 :009 > User.find_by_name('Ralph Maurer')
2.6.3 :010 > User.find_by_email('ralph.maurer@iet-gibb.ch')
2.6.3 :011 > User.find_by_age(22)
```

Die Methoden `find_by` geben nur den ersten Datensatz aus, der auf den Attributwert passt. Zum Beispiel sind Ralph Maurer und Michael Abplanalp beide 22 Jahre alt. Ausgegeben wird nur der ersterfasste Ralph Maurer.

Die Methoden `first`, `last` und `find` geben jeweils keinen oder einen Datensatz zurück.

Es werden jeweils Objekte zurückgegeben, und falls diese einer Objektvariablen zugewiesen werden, können die Attributwerte wiederverwendet werden!
Die Ausgabe ist gekennzeichnet mit *=> #<User id: 1...*

Beispiel:
```
user = User.first
user_name = user.name
```

Alle Datensätze mit der Methode `all` ausgeben:

```
2.6.3 :012 > User.all
```

Nur bestimmte Attribute ausgeben mit der Methode `select`:

```
2.6.3 :013 > User.select(:name,:email)
```

Datensätze mit der Methode `where` (`where` -Klausel) filtern:

```
2.6.3 :014 > User.where(name: 'Ralph Maurer')
```

Datensätze mit der `select`- und `where`-Klausel filtern:

```
2.6.3 :015 > User.select(:name,:email).where(name: 'Ralph Maurer')
```

Datensatz in eine Variable speichern:

```
2.6.3 :016 > u = User.where(name: 'Ralph Maurer')
```

Die Methoden `all`, `select` und `where` geben jeweils keinen, einen oder mehrere Datensätze zurück.

Es werden jeweils Arrays von Objekten zurückgegeben, auch wenn aus der Abfrage nur 1 Datensatz resultiert. Falls die Rückgabe einer Objektvariablen zugewiesen wird, können die Attributwerte wiederverwendet werden! Der Zugriff auf die einzelnen Objekte (Datensätze) erfolgt über den Index.
Die Ausgabe ist gekennzeichnet mit => *#<ActiveRecord::Relation*

Beispiel:
```
user = User.all
user_name = user[0].name
```

Daten im Model "User" erfassen: Variante 2

Jetzt sehen wir erst, wie einfach und cool Rails von Ruby profitiert.

Angenommen, wir wollen weitere Personen erfassen. Dazu instanziieren wir die zwei Objekte *p1* und *p2* der Klasse *User*. D.h. unser Model agiert wie ein Objekt und wir können den Objekteigenschaften `name`, `email` und `age` Werte zuweisen.

Erfassen wir die einzelnen Attribute der ersten Person und erstellen dazu mit der Methode new eine neue Person *p1*. Man nennt dies Objektinstanz von der Klasse *User*.

```
2.6.3 :016 > p1 = User.new
2.6.3 :017 > p1.name = 'Reto Glarner'
2.6.3 :018 > p1.email = 'reto.glarner@iet-gibb.ch'
2.6.3 :019 > p1.age = 25
2.6.3 :020 > p1.save
```

Wir können ein Objekt direkt mit Werten bestücken und abspeichern. Hier am Beispiel der Person *p2*:

```
2.6.3 :021 > p2 = User.new(name: 'Thomas Staub', email:
'thomas.staub@iet-gibb.ch', age: 24)
```

Daten im Model "User" ändern

```
2.6.3 :022 > p2.save
```

Wir erfassen zunächst einen Datensatz mit falschen Daten für eine Person *p3*:

```
2.6.3 :023 > p3 = User.new(name: 'Chris Schweingruper', email:
'cs@iet-gib.ch', age: 0)
2.6.3 :024 > p3.save
```

Speichern wir den falsch erfassten Datensatz der Person *p3* in eine Variable `p_change`. Wir finden die Person mit der `where`-Klausel. Die `where` -Klausel gibt ein Array zurück, der Einfachheit halber hängen wir die Methode `first` an, damit wir ein Objekt und nicht ein Array mit Objekten erhalten:

Beispiel der Arrayrückgabe

```
2.6.3 :025 > User.where(name: "Chris Schweingruper")
  User Load (0.2ms)  SELECT  "users".* FROM "users" WHERE
"users"."name" = ? LIMIT ?  [["name", "Chris Schweingruper"],
["LIMIT", 11]]
 => #<ActiveRecord::Relation [#<User id: 7, name: "Chris
Schweingruper", email: "csb@iet-gib.ch", age: 0, created_at: "2018-
04-29 17:25:15", updated_at: "2018-04-29 17:25:15">]>
```

Beispiel der Objektrückgabe

```
2.6.3 :026 > User.where(name: "Chris Schweingruper").first
  User Load (0.2ms)  SELECT  "users".* FROM "users" WHERE
"users"."name" = ? ORDER BY "users"."id" ASC LIMIT ?  [["name",
"Chris Schweingruper"], ["LIMIT", 1]]
 => #<User id: 7, name: "Chris Schweingruper", email: "csb@iet-
gib.ch", age: 0, created_at: "2018-04-29 17:25:15", updated_at:
"2018-04-29 17:25:15">
```

Holen wir nun die Person p3 und weisen das Objekt der Variablen `p_change` zu:

```
2.6.3 :026 > p_change = User.where(name: "Chris
Schweingruper").first
```

Alternativ 😊:

```
2.6.3 :027 > p_change = User.last
```

Wir korrigieren die Fehler im Datensatz der Person *p3*:

```
2.6.3 :028 > p_change.name = 'Christian Schweingruber'
2.6.3 :029 > p_change.email = 'christian.schweingruber@iet-gibb.ch'
2.6.3 :030 > p_change.age = 26
2.6.3 :031 > p_change.save
```

Kontrollieren wir die Aktualisierung

```
2.6.3 :032 > User.all
```

Datensatz aus Model "User" löschen

Einen Benutzer löschen wir einfach mit einer der Methoden *`destroy`* bzw. *`delete`:*

```
2.6.3 :033 > User.last.destroy
oder
2.6.3 :034 > User.last.delete
```

Wenn Sie alle Schritte korrekte durchgeführt haben, haben Sie nun 5 Personen im Model `user`. Falls Sie zu wenig oder zu viele Personen haben, machen Sie entsprechende Korrekturen.

Kontrolle, ob 5 Personen erfasst sind:

```
2.6.3 :035 > User.all
  User Load (0.2ms)  SELECT  "users".* FROM "users" LIMIT ?
[["LIMIT", 11]]
 => #<ActiveRecord::Relation
[#<User id: 1, name: "Ralph Maurer", email: "ralph.maurer@iet-
gibb.ch", age: 22, created_at: "2018-04-27 13:01:45", updated_at:
"2018-04-27 13:04:14">,
#<User id: 2, name: "Michael Abplanalp", email:
"michael.abplanalp@iet-gibb.ch", age: 22, created_at: "2018-04-27
13:02:21", updated_at: "2018-04-27 13:02:21">,
#<User id: 3, name: "Christian Schweingruber", email:
"christian.schweingruber@iet-gibb.ch", age: 23, created_at: "2018-
04-27 20:16:45", updated_at: "2018-04-27 20:16:45">,
#<User id: 4, name: "Reto Glarner", email: "reto.glarner@iet-
gibb.ch", age: 25, created_at: "2018-04-27 20:55:03", updated_at:
"2018-04-29 16:11:36">,
#<User id: 5, name: "Thomas Staub", email: "thomas.staub@iet-
gibb.ch", age: 24, created_at: "2018-04-29 15:21:02", updated_at:
"2018-04-29 16:23:32">]>
```

Als nächstes wollen wir nun die Personen auf unserer Webseite in einer View anzeigen.

Model "User" mit Rails anzeigen

Das Model können wir ganz einfach als View anzeigen. Es müssen vorerst einige Vorbereitungen gemacht werden.

Wir erstellen zunächst einen neuen Controller `users/show`. Notieren Sie das dafür notwendige Kommando:

Im Menu unserer Website soll ein neuer Menueintrag auf die neue Seite `users/show` zeigen. Ergänzen Sie die Bootstrap-Navbar mit dem neuen Eintrag.

Wie wir bereits im letzten Arbeitsblatt AB133-02 gesehen haben (letzte Seite), gibt es in Rails die Datei `app/views/layouts/application.html.erb` mit einem HTML-Gerüst, das bei allen Seiten zur Anwendung kommt:

Die Datei beinhaltet ein Gerüst, das alle Meta-Tags, CSS-Stylesheet-Verweise und Javascript-Referenzen sammelt und vereint. `yield` ist ein Platzhalter für die View, der Code der entsprechenden Datei wird dort zur Laufzeit eingefügt.

› `<%=yield%>` steht für die Ausgabe der Variablen `yield` (das ist unsere View) .
› `<%...%>` Innerhalb dieses Tags kann Ruby programmiert werden.
› `<%=...%>` Dieses Tag bedeutet eine Anzeige am Bildschirm, also eine Ausgabe.

Unsere View `app/views/users/show.html.erb` braucht kein HTML-Gerüst, da sich dieses in `app/views/layouts/application.html.erb` befindet. Der Inhalt von `show.html.erb` wird beim Rendern der Seite anstelle von `<%= yield %>` eingesetzt.

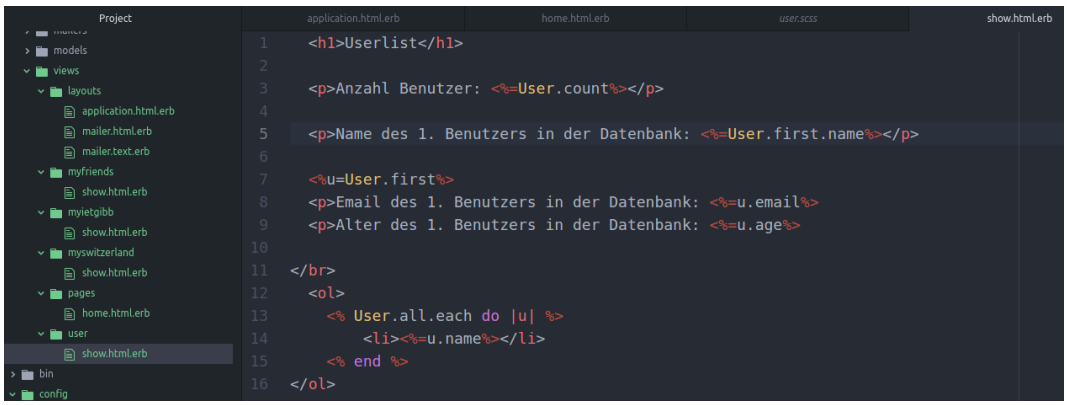
Wenn Sie alle Schritte des letzten Arbeitsblattes durchgeführt haben, dann haben Sie Teile des Headers und die Bootstrap-Navbar bereits in die Datei `app/views/layouts/application.html.erb` verschoben.

Falls Sie dies noch nicht erledigt haben, dann holen Sie es jetzt nach!

Kontrollieren Sie, ob via neuen Menueintrag die Seite `myapp/app/views/users/show.html.erb` erreichbar ist.

`.erb` steht übrigens für Embedded Ruby. Das bedeutet, dass wir Ruby-Code direkt in die View einbinden können. Wir können also die in diesem Arbeitsblatt gelernten Konsolen-Eingaben in der View verwenden.

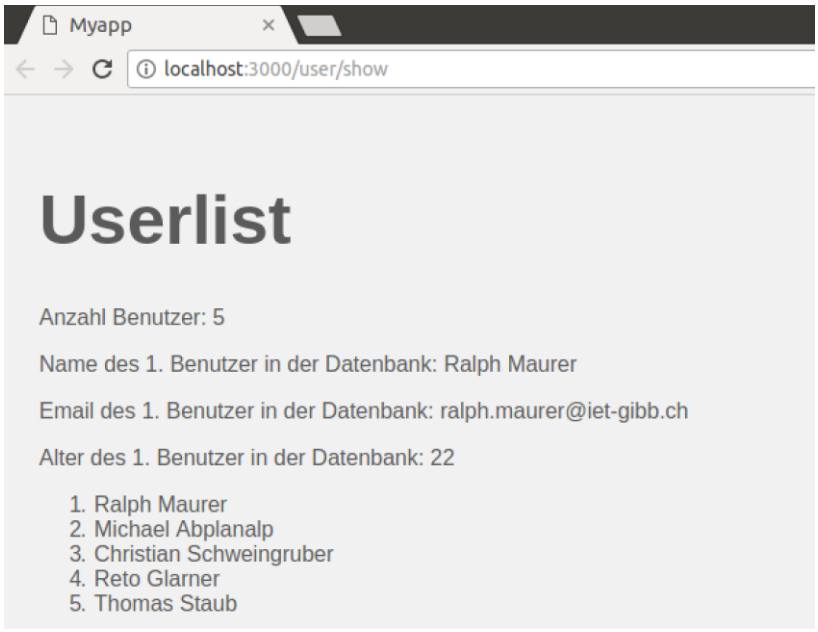
Schreiben Sie folgenden Code in `myapp/app/views/users/show.html.erb`:



Die Seite sieht so aus:

Was passiert in Zeile 3:

```
<p>Anzahl Benutzer: <%= User.count %></p>
```



Was passiert in Zeile 5:

```
<p>Name des 1. Benutzers in der Datenbank: <%= User.first.name %></p>
```

Was passiert in Zeile 7:

```
<% u=User.first %>
```

Was passiert in Zeile 8:

```
<p>Email des 1. Benutzers in der Datenbank: <%= u.email %>
```

Was passiert in Zeile 9:

```
<p>Alter des 1. Benutzers in der Datenbank: <%= u.age %>
```

Was passiert in den Zeilen 12 bis 16:

```
<ol>
  <% User.all.each do |u| %>
    <li><%=u.name%></li>
  <% end %>
</ol>
```

Sämtlicher Code befindet sich in der View (Zugriff auf die Datenbank und Aufbereitung der Daten).

Dies widerspricht dem MVC-Muster und wird hier aus didaktischen Gründen so gemacht, damit wir uns im Moment nicht mit dem Controller befassen müssen!

In Zukunft werden wir die Kontrolle der Applikation und die Datenbankzugriffe ausschliessliche über den Controller abwickeln.

Verbesserte Anzeige des Models

Ändern Sie nun die View `/users/show` folgendermassen ab:

1. Alle Benutzer werden aus der Datenbank geholt und in einer `each`-Schleife als sortierte Liste ausgegeben.
2. Innerhalb der Listenelemente werden die Eigenschaften `name`, `email` und `age` ausgegeben, jeweils auf einer neuen Zeile, aber ohne `<br />`-Tag!
3. Die E-Mail-Adresse soll als Link dargestellt werden (`<a>`-Tag mit `mailto`-Attribut).

Ansicht der fertigen View: