Quicknotes AB151-04 – 05

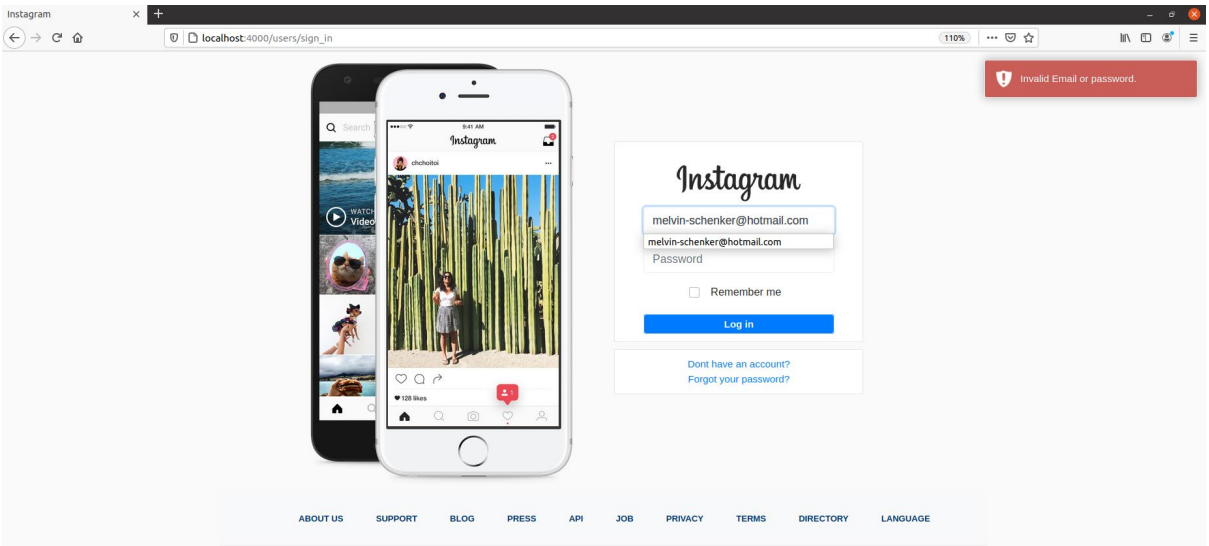
Flash in Rails

Flash ist ein ähnliches Konzept wie Sessions (Variablen, welche über die ganze Applikation hinweg verfügbar sind). Das heißt es ist ebenfalls ein Array mit Variablen, jedoch gibt es ein Unterschied: Die Werte, welche ins Flash Array gespeichert werden, sind nur in der nächsten View verfügbar. Konkret heißt das, dass wenn man eine Login Aktion durchführt, kann auf der nächsten Seite die Nachricht «Erfolgreich eingeloggt» stehen. Sobald man jedoch z.B. mit einem redirect eine neue View öffnet, ist die Login-Nachricht nicht mehr verfügbar. Flash Einträge sind immer Key-Value Pairs. Mögliche Keys sind success, alert, notice oder error. Flash eignet sich sehr gut um Nachrichten an den User zu einer kürzlich getätigten Aktion zu senden.

JavaScript Toast-Meldungen mit gem toastr

toastr ist ein Ruby-Gem, mit dem man sehr einfach Notifikationen in seinem App einrichten kann. Alles was man tun muss, ist die entsprechende JavaScript Methode dieses Gems mit einem Text aufzurufen (z.B. toastr.success(„Wow, a success“)). Dieses Gem baut auf den Flash in Rails auf und gibt dessen Nachrichten aus, sobald sie im Flash-Array erscheinen. Dabei wird die Nachricht immer oben recht im Bildschirm angezeigt und verschwindet nach kurzer Zeit wieder. Je nach Key des Eintrages im Flash sind die Icons und die Hintergrundfarbe der Nachricht unterschiedlich (z.B. der Key error wird rot mit einem Ausrufezeichen angezeigt). Es ist sehr einfach implementierbar, hat ein gutes Design, aber besitzt nur vier davon.

Hier ein Beispiel eines invaliden Login-Versuches (Password nicht eingegeben):



Flash-Meldungen vs. Meldungen des Models

Flash-Meldungen	Meldungen des Models
Sind nur auf dieser Seite verfügbar und verschwinden bei einem Wechsel der Seiten Eignet sich zur Darstellung von Nachrichten auf eine Aktion (z.B. Login Successful) Können Nachrichten von Aktionen der vorherigen Seite aufzeigen	Können Nachrichten der aktuellen Seite aufzeigen Eignet sich zur Darstellung von Nachrichten auf eine Eingabe (z.B. die Validation von Input Feldern).

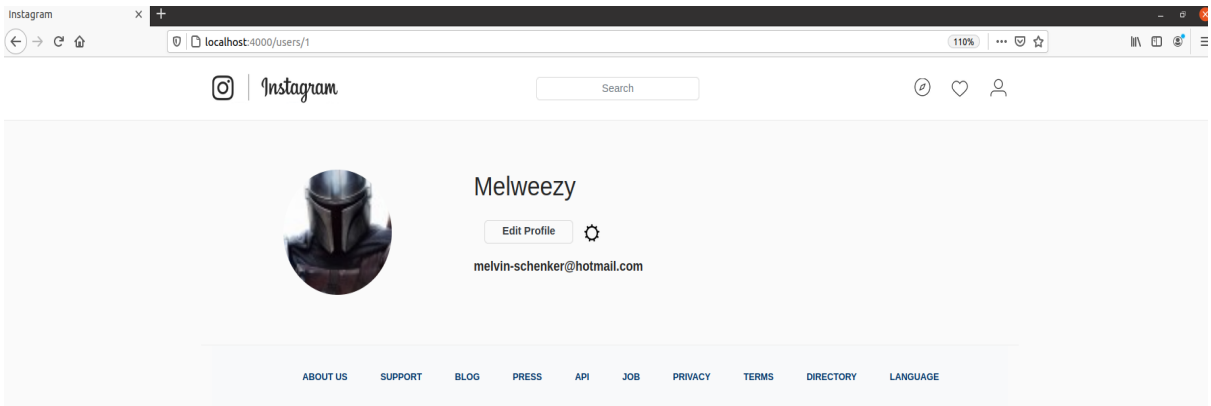
Bootstrap Modal

Modals ist ein Element, welches von Bootstrap zur Verfügung gestellt wird. Es dient dazu ein Fenster anzuzeigen, ohne die Route zu wechseln und eine neue Seite laden zu müssen. Außerdem kann es auch z.B. als Popup-Fenster verwendet werden. In unserem Instagram-Beispiel haben wir die Edit Profile Seite (app/views/users/show.html.erb) mit einem Modal realisiert.

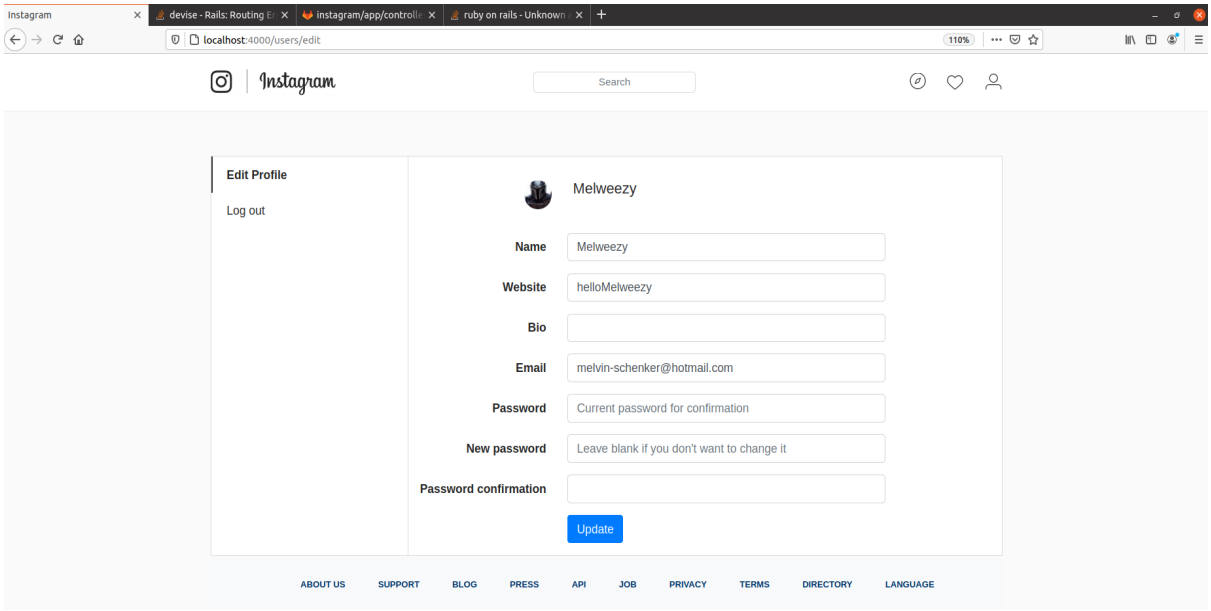
Gravatar für Benutzerbild

Gravatar ist eine Applikation, welche es ermöglicht, einer E-Mail-Adresse ein Profilbild zuzuteilen. Darauf kann dann in anderen Applikationen zugegriffen werden, um diese z.B. in einer Profilseite anzeigen zu lassen. Ein wichtiger Vorteil den Service von Gravatar zu benutzen, ist dass man die Profilbilder nicht selbst auf einer Datenbank speichern muss.

Hier ein Benutzer mit einem auf Gravatar gesetzten Profilbild:



Edit Profile Ansicht:



ERD-Modell

Ein Entity-Relationship-Diagram (ERD) dient dazu, dem Entwickler eine Übersicht über die entsprechende Datenbank zu geben. Wie es der Name schon sagt, sind die Beziehungen zwischen den Tabellen dabei von grosser Bedeutung. Sie zeigen an, wie viele Elemente einer Tabelle in einer anderen vorkommen können. Ein Beispiel dafür wäre: Die Tabellen Post und User haben die Beziehung m:1 (many to 1). Anders ausgedrückt: Ein Post kann nur zu einem User gehören, jedoch kann ein User mehrere Posts haben.

Bilder hochladen mit Gem CarrierWave

Damit wir Bilder von unserer Instagram App hochladen können benutzen wir ein Gem, welches dies ermöglicht: CarrierWave. In Ruby gibt es viele verschiedene Gems, welche die Funktionalität eines Fileuploads bieten, jedoch wird von vielen Entwicklern CarrierWave aufgrund der hohen Flexibilität und Erweiterbarkeit empfohlen. CarrierWave funktioniert mit einem sogenannten Uploader, was einem Controller in Rails sehr ähnlich ist. Darin findet die ganze Logik des Fileuploads statt (Zielordner angeben, Grösse des Bildes ändern etc.).

Fotos in Cloud speichern mit Gem Cloudinary

Die Bilder, welche vom Gem CarrierWave hochgeladen werden, können in einer App wie Instagram nicht nur lokal gespeichert werden, sondern müssen auf der Cloud verfügbar gemacht werden. Hier kommt das Gem Cloudinary ins Spiel. Cloudinary kümmert sich um alles was mit Cloud Storage zu tun hat, das heisst sobald man mit CarrierWave das Bild hochladet, wird Cloudinary dieses selbst auf eine Cloud-Ablage (Amazon S3) ablegen. Das Gem regelt hierbei die Verbindung von der Cloud zur Applikation mit einem Cloud Namen, einem API-Key und einem API-Secret, welche man nach der Registration auf der Webseite erhält. Diese muss man dann nur noch in das automatisch erstellte Config-File von Cloudinary kopieren und schon kann man Bilder in die Cloud laden.

Cloudinary ist gut mit dem Gem CarrierWave kompatibel, deshalb wurden diese zwei Gems in unsere Instagram App implementiert. Ausserdem ist Cloudinary für den kleineren Gebrauch gratis.

Per rails-Befehl ist es möglich den Uploader für CarrierWave zu erstellen, hier einen Einblick in den erstellten PhotoUploader in **photo_uploader.rb** mit den nötigen Einträgen für Cloudinary.

```
def store_dir
  "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
end

include Cloudinary::CarrierWave

process :convert => 'jpg'
process :tags => ['post_picture']
```

```
version :standard do
  process :resize_to_fill => [300, 300, :north]
end

version :thumdnail do
  process :resize_to_fit[100, 100]
end
```

Verschlüsselung mit Gem Figaro

Die von der Cloudinary Registration erhaltenen Daten (Cloud-Name. API-Key, API-Secret) sind sensitive Daten, da sie den Zugriff auf die hochgeladenen Bilder ermöglichen. Würde man den Quellcode nun veröffentlichen, wären diese Daten für alle einsehbar. Um dies zu verhindern, wird das Gem Figaro benutzt. Es dient dazu sensitive Konfigurationswerte wie z.B. den API-Key von Cloudinary zu verschlüsseln, um vorhin genannte Situation zu vermeiden. Figaro funktioniert so, dass es die sensitiven Werte in Environment Variablen schreibt, welche nicht im Code selbst definiert werden. Die Environment Variablen werden in einem separaten YAML File definiert, welches vom Programm losgelöst ist und somit auch nicht z.B. auf ein Git-Repository geladen werden kann.

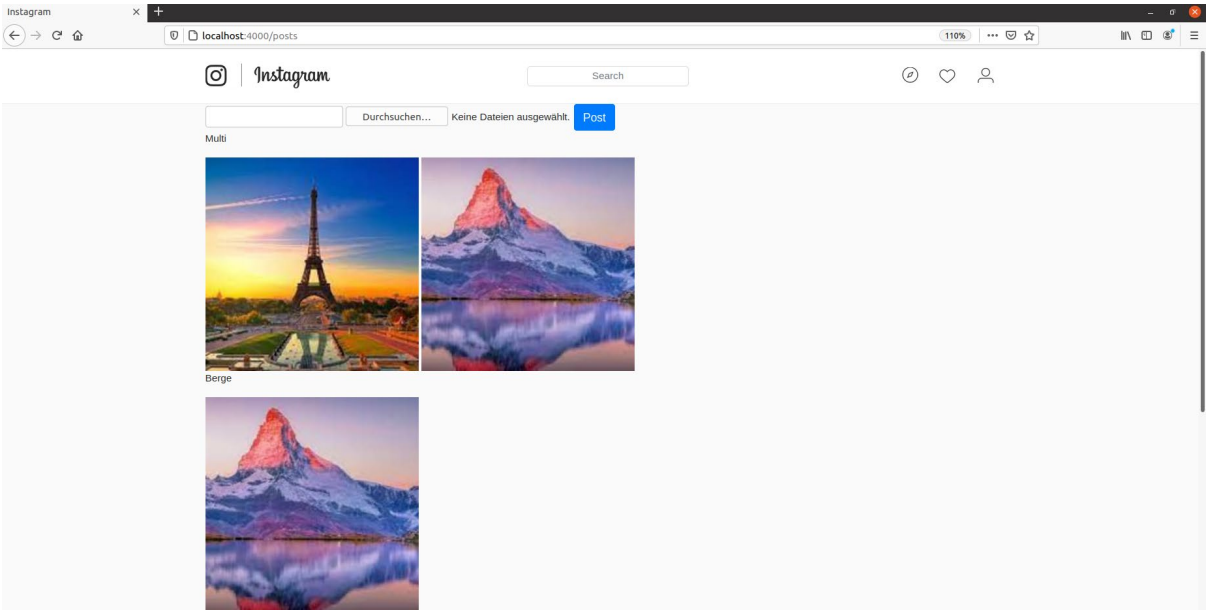
View post#index

Um nun Fotos hochladen zu können und diese schließlich anzuzeigen, erstellten wir nun den neuen Ordner app/views/posts für die index.html.erb Datei. In dieser Datei wird ein Form-Tag mit dem Rails-Helper form_for für den Bilder-Upload erfasst. Und zum Anzeigen der Posts wird eine doppelte Iteration durch die Posts und deren Photos erstellt:

```
<%= form_for @post, html: {multipart: true} do |f| %>
  <%= f.text_field :content, class: "control-label" %>
  <%= file_field_tag "images[]", type: :file, multiple: true %>
  <%= f.submit "Post", class: "btn btn-primary" %>
<% end %>

<%= @posts.each do |post| %>
  <p><%= post.content %></p>
  <% post.photos.each do |photo| %>
    <%= image_tag photo.image.url(:standard) %>
  <%end%>
<%end%>
```

Erste Posts:



Selbstreflexion

Was habe ich gelernt?

Dieses Arbeitsblatt war sehr lehrreich, da viele Technologien vorkamen, welche ich noch nicht kannte. Vor Allem Gravatar ist eine interessante Applikation, welche ich sicher wiederverwenden werde. Auch das Konzept von Flash-Speicher und dessen Implementation in Rails war mir neu.
In diesem Arbeitsblatt hat man drei sehr hilfreiche Gems kennengelernt, welche das Hochladen von Files in die Cloud ermöglichen. Ich kannte zuvor noch keine davon und konnte viel bei der Implementierung lernen. Vor allem das Gem Figaro werde ich in weiteren Rails Projekten benutzen können.

Wie bin ich beim Ausführen des Auftrages vorgegangen?

Ich habe zuerst das Grundgerüst der Dokumentation aufgestellt und überlegt, zu welchen Technologien ich eine Erklärung schreiben werde. Dann habe ich nach jedem Kapitel die Dokumentation weitergeschrieben und zum Schluss noch die Reflexion ergänzt.
Ich habe zuerst das Grundgerüst der Dokumentation aufgestellt und überlegt, zu welchen Technologien / Methoden ich eine Erklärung schreiben werde. Dann habe ich nach jedem Kapitel die Dokumentation weitergeschrieben und zum Schluss noch die Reflexion ergänzt.

Was waren die Schwierigkeiten?

Ich hatte Schwierigkeiten den Cloudinary Account mit der Applikation zu verbinden. Am Schluss hat es jedoch trotzdem funktioniert.

Was habe ich nicht verstanden?

Zum Schluss hatte ich etwas Mühe die Methoden des Post Controllers zu verstehen, jedoch konnte ich mir mit der offiziellen Dokumentation Klarheit schaffen.

Was kann ich das nächste Mal besser machen?

Die Zeit besser einteilen, damit zum Schluss weniger Arbeit zu Hause anliegt.

Abschließende Reflexion über das Gelernte

Zusammenfassung über das Gelernte
Die Technologien und Techniken von Flash-Speicher in Rails, vom Gem toastr, von Bootstrap Modal und von Gravatar waren mir neu. Einzig die Resource Routes waren mir bereits durch das Vorgängermodul bekannt.
Die drei Gems CarrierWave, Cloudinary und Figaro waren mir neu. Das Konzept einer relationalen Datenbank und eines ERD waren mir schon von Vorgängermodulen bekannt.
Das Erstellen des Posts Controllers war mir ebenfalls schon vom Vorgängermodul bekannt.

Lernfortschritt
Ich konnte mein Wissen in Rails mit toastr und Modals erweitern und habe neues über Bootstrap gelernt. Ausserdem habe ich die Applikation Gravatar kennengelernt.
Ich konnte mein Wissen in den drei Gems erweitern und habe dabei neues über das Abspeichern von Daten in der Cloud gelernt. Außerdem weiß ich nun, wie man sensitive Konfigurationswerte in Rails richtig abspeichert.