Luca Scherer
INF17H

Quicknote AB151-05-06

ERD – Instagram Datenbank

Das ERD zeigt, das ein User mehrere Posts erfassen kann, zu welchen mehrere Fotos gehören. Der Rest ist aktuell noch nicht relevant.

Post und Photo Modell

In diesem Schritt habe ich das Post und Photo Model erstellt. Das war mit 2 Zeilen im Terminal getan. Um zu sagen, dass eine Beziehung zwischen den einzelnen Models besteht, brauchte ich «:references». Damit wurde automatisch ein Fremdschlüssen erstellt. In der Photo-Tabelle heisst dieser «post_id» und in der Post-Tabelle «user_id».

Danach habe ich noch die Beziehung in den Models definiert:

```ruby
class Post < ApplicationRecord
  belongs_to :user
  has_many :photos, dependent: :destroy
end
```

```ruby
class Photo < ApplicationRecord
  belongs_to :post
end
```

```ruby
class User < ApplicationRecord
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable, :trackable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :validatable
  validates :name, presence: true
  validates_length_of *attr_names :name, maximum: 50

  has_many :posts, dependent: :destroy
end
```

Upload image mit «gem CarrierWave»

In diesem Schritt installierte ich zuerst das gem «Carrierwave», Danach erstellte ich einen Uploader mit «rails generate uploader Photo». Carrierwave ist ein Gem welches einem hilft Daten von Ruby-Apps hochzuladen. Danach registrierte ich mich auf Cloudinary.com und installierte das gem «cloudinary». Cloudinary ist ein Framework welches ermöglicht Bilder und Videos für Websites hochzuladen, zu speichern, zu verwalten, zu manipulieren und bereitzustellen. Um auf die «Cloud» zu zugreifen braucht man dessen Namen, einen API-Key und ein API-Secret. Diese drei Werte schrieb ich in einen neuerstellten Initializer namens cloudinary.rb.

Damit CarrierWave und Cloudinary zusammen funktionieren, habe ich folgenden Code aus der Anleitung in den zuvor erstellten Uploader geschrieben und angepasst:

```ruby
class PhotoUploader < CarrierWave::Uploader::Base
  # Include RMagick or MiniMagick support:
  # include CarrierWave::RMagick
  # include CarrierWave::MiniMagick

  # Choose what kind of storage to use for this uploader:
  #storage :file
  # storage :fog

  # Override the directory where uploaded files will be stored.
  # This is a sensible default for uploaders that are meant to be mounte
  def store_dir
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}/#{model.id}"
  end

  include Cloudinary::CarrierWave

  process :convert => 'png'
  process :tags => ['post_picture']

  version name :standard do
    process :resize_to_fill => [1080, 1080, :center]
  end

  version name :thumbnail do
    resize_to_fit( width 100, height 100)
  end
```

«gem Figaro» encrypt, decrypt

In diesem Schritt habe ich das gem «Figaro» installiert und den Befehl «bundle exec figaro install» ausgeführt. Dieser erstellte im Config-Ordner ein File namens application.yml. Dort habe ich dann meine Cloud-Daten eingetragen. In meinem Cloudinary.rb-File konnte ich die Daten nun wie folgt ansprechen:

```ruby
Cloudinary.config do |config|
  config.cloud_name = ENV["cloudinary_cloud_name"]
  config.api_key = ENV["cloudinary_api_key"]
  config.api_secret = ENV["cloudinary_api_secret"]
  config.cdn_subdomain = true
end
```

Post controller

In diesem Schritt habe ich zuerst das Löschverhalten in den Modellen definiert.

```ruby
has_many :photos, dependent: :destroy
```

Mit dependent: :destroy sagt man, dass wenn z.B ein Post gelöscht wird, auch alle seine Bilder gelöscht werden.

Danach habe ich manuell den Post Controller erstellt und dessen Methoden wie folgt angepasst:

```ruby
class PostsController < ApplicationController
  before_action :authenticate_user!
  before_action :find_post, only: [:show, :destroy]

  def index
    @posts = Post.all.limit(10).includes(:photos, :user).order('created_at desc')
    @post = Post.new
  end

  def create
    @post = current_user.posts.build(post_params)
    if @post.save
      if params[:images]
        params[:images].each do |img|
          @post.photos.create(image: img[1])
        end
      end

      redirect_to posts_path
      flash[:notice] = "Saved ..."
    else
      flash[:alert] = "Something went wrong ..."
      redirect_to posts_path
    end
  end

  def show
    @photos =   @post.photos
  end

  def destroy

  end

  private

  def find_post
    @post = Post.find_by id: params[:id]

    return if @post
    flash[:danger] = "Post doesn't exist!"
    redirect_to root_path
  end

  def post_params
    params.require(:post).permit :content
  end
end
```

Anschliessend habe ich die Datei app/views/posts/index.html.erb erstellt und wie folgt ergänzt:

```
<%= 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%>
```

Diese View enthält ein Formular zum posten und eine Liste aller Posts. Nach einem Server-Neustart konnte ich bereits Posts erstellen und anschauen.

Luca Scherer
INF17H

DropzoneJS-rails

Um eine DragnDrop-Zone zu realisieren benutzen wir das gem «dropzonejs-rails». Dieses habe ich am Anfang dieses Schritts installiert und im application.js und application.scss importiert. Danach habe ich fontawesome in mein Projekt importiert. Das ist ein Framework, welches schöne Icons bereitstellt.

Post-View HTML

In diesem Schritt habe ich die post/index-Seite nochmals angepasst:

```
<div class="d-flex flex-column align-items-center mt-3">
  <div class="col-xl-7 col-lg-8 col-md-10 col-sm-11 post-card">
    <div class="card">
      <div class="card-header">
        Create Post
      </div>
      <div class="card-body">
        <%= form_for @post, :html => {:multipart => true, :class => "dropzone upload-images p-0 border-0"} do |f| %>
          <div class="form-group row mt-2">
            <div class="col-auto pr-0">
              <%= image_tag avatar_url(current_user), class: "post-author-icon" %>
            </div>
            <div class="col pl-0">
              <%= f.text_field :content, class: "form-control border-0",
                placeholder: "Say something about this ..." %>
            </div>
          </div>
          <div class="fallback">
            <%= file_field_tag "images[]", type: :file, multiple: true %>
          </div>
          <div class="dz-message m-0"></div>
          <div class="dropzone-previews mb-3">
            <div class="upload-photos-icon">
              <i class="fa fa-plus fa-2x" aria-hidden="true" style="color:#dddfe2"></i>
            </div>
          </div>
          <%= f.submit "Post", class: "btn btn-primary" %>
        <% end %>
      </div>
    </div>
  </div>
</div>

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

Post view SCSS

In diesem Schritt habe ich das application.scss-File angepasst. Dazu habe ich den ganzen Code vom AB abgeschrieben.

+ Button mit DropzoneJS gestalten

Hier habe ich die Datei upload_post_images.js erstellt und wie folgt ergänzt:

```
$(document).ready(function(){
  $(".upload-images").dropzone({
    addRemoveLinks: true,
    maxFilesize: 1,
    autoProcessQueue: false,
    uploadMultiple: true,
    parallelUploads: 100,
    maxFiles: 100,
    paramName: "images",
    previewsContainer: ".dropzone-previews",
    clickable: ".upload-photos-icon",
    thumbnailWidth: 100,
    thumbnailHeight: 100,

    init: function(){
      var myDropzone = this;

      this.element.querySelector("input[type=submit]").addEventListener("click", function(e){
        e.preventDefault();
        e.stopPropagation();
        myDropzone.processQueue();
      });

      this.on("successmultiple", function(files, response){
        window.location.reload();
      });

      this.on("errormultiple", function(files, response){
        toastr.error(response);
      });
    }
  })
});
```

Wenn die Seite geladen ist werden die Upload-Parameter für das Formular definiert. Danach wird ein Clicklistener definiert.

Wenn man nun auf den submit-Button klickt, werden die Files hochgeladen und eine passende Meldung ausgegeben.

Design für Postliste

Für die Umsetzung der Anforderungen habe ich zuerst die Index-Methode im Post-Controller angepasst. Und wird @posts zusätzlich mit dem User abgefüllt. Ausserdem ist die Reihenfolge nun neu-alt:

```
def index
  @posts = Post.all.limit(10).includes(:photos, :user).order('created_at desc')
  @post = Post.new
end
```

Danach habe ich nochmals die index-View angepasst. Dazu musste ich wieder Code vom AB abschreiben.

Anschliessend habe ich die Datei mit folgendem Code ergänzt:

```
<div class="carousel-inner">
    <div class="carousel-item active">
      <img class="d-block w-100" src=".../800x400?auto=yes&bg=777&fg=555&text=First slide" alt="First slide">
    </div>
    <div class="carousel-item">
      <img class="d-block w-100" src=".../800x400?auto=yes&bg=666&fg=444&text=Second slide" alt="Second slide">
    </div>
    <div class="carousel-item">
      <img class="d-block w-100" src=".../800x400?auto=yes&bg=555&fg=333&text=Third slide" alt="Third slide">
    </div>
  </div>
  <a class="carousel-control-prev" href="#carouselExampleControls" role="button" data-slide="prev">
    <span class="carousel-control-prev-icon" aria-hidden="true"></span>
    <span class="sr-only">Previous</span>
  </a>
  <a class="carousel-control-next" href="#carouselExampleControls" role="button" data-slide="next">
    <span class="carousel-control-next-icon" aria-hidden="true"></span>
    <span class="sr-only">Next</span>
  </a>
```

Dieser erstellt ein Carousel. Damit man alle Fotos eines Posts sehen kann. Danach habe ich die View noch ein wenig formatiert.

Partial-View für Posts list

Damit die Post-View übersichtlicher ist wird die Post-Liste in eine Partial View ausgelagert. Die List wird dann in einem Container mit der id=post gerendert:

```
<div id="post">
  <%= render 'posts_list'%>
</div>
```

Post View Button designen

In diesem Schritt musste ich wieder viel CSS abschreiben:

```
// Customize post views
.post {
  .card-header {
    background-color: white;
  }
  .card-body {
    padding: 5px 16px 5px;
  }
  .actions {
    margin: 12px 0;
  }
  .post-time {
    margin-top: 5px;
    font-size: 10px;
  }
  .comment-input {
    font-size: 14px;
    resize: none;
  }
  .previous-btn {
    background-position: -281px -280px;
    height: 30px;
    opacity: 1;
    width: 30px;
  }
  .next-btn {
    background-position: -353px 0px;
    height: 30px;
    opacity: 1;
    width: 30px;
  }
}

.love {
  background-position: -325px -329px;
  height: 24px;
  width: 24px;
}
.loved {
  @extend .love;
  background-position: -200px -330px;
}
.comment {
  @extend .love;
  background-position: -258px -175px;
  margin-left: 8px;
}
.bookmark {
  @extend .love;
  background-position: -150px -355px;
}
.bookmarked {
  @extend .love;
  background-position: -353px -273px;
}
```

```
.no-text-decoration:hover {
  text-decoration: none;
  color: #262626;
}

.normal-color {
  color: #262626;
}
```

Anschliessend musste ich in der Partial View noch einige Klassen anpassen:

```
<a href="#" class="core-sprite love hide-text"> Love </a>
<a href="#" class="core-sprite comment hide-text"> Comment </a>
<a href="#" class="core-sprite bookmark hide-text ml-auto"> Bookmark </a>
```

Zum Abschluss habe ich noch die resize-Grösse auf 1080x1080 geändert und die Post-Index-View als root definiert.

Selbstreflexion

1. Was habe ich gelernt?

- Wie man in Rails mit verschiedenen Gems Bilder hochlädt, verwaltet und benutzt.
- Wie man Fremdschlüssel mit dem Terminal erstellt und passende Beziehungen definiert.
- Wie man Daten innerhalb einer Rails-App verschlüsselt. (Figaro)
- Wie man das Löschverhalten definiert.
- Wie man DropZone und Fontawesome anwendet.
- Wie man ein Carousel erstellt.

2. Was hat mich behindert?
Behindert wurde ich durch nichts, was mit dem AB zu tun hat. Allerdings habe ich kurz bevor ich mit der QN zu Ende war meine Gibbix verloren. Dabei habe ich den ganzen Projekt-Fortschritt und meine fast fertige QN verloren.
3. Was habe ich nicht verstanden?
Ich habe alle Technologien verstanden.
4. Was kann ich beim Studium besser machen?
Man muss viel abschreiben, wobei man enorm viel Zeit verliert.

Fazit

Die beiden Arbeitsblätter waren sehr gut. Sie zeigen wie man eine App baut, mitwelcher man Files hochladen und verwalten kann. Ausserdem zeigen sie sehr gut, wie man mit Hilfe anderer Frameworks schneller und effizienter vorwärtskommt. Man muss nicht immer alles selber schreiben.