



# **MODUL 151**

## **TEIL 5: INSTAGRAM POST FEATURE**

Ralph Maurer

# Inhaltsverzeichnis AB151-05

## Modul 133: Instagram mit Rails bauen

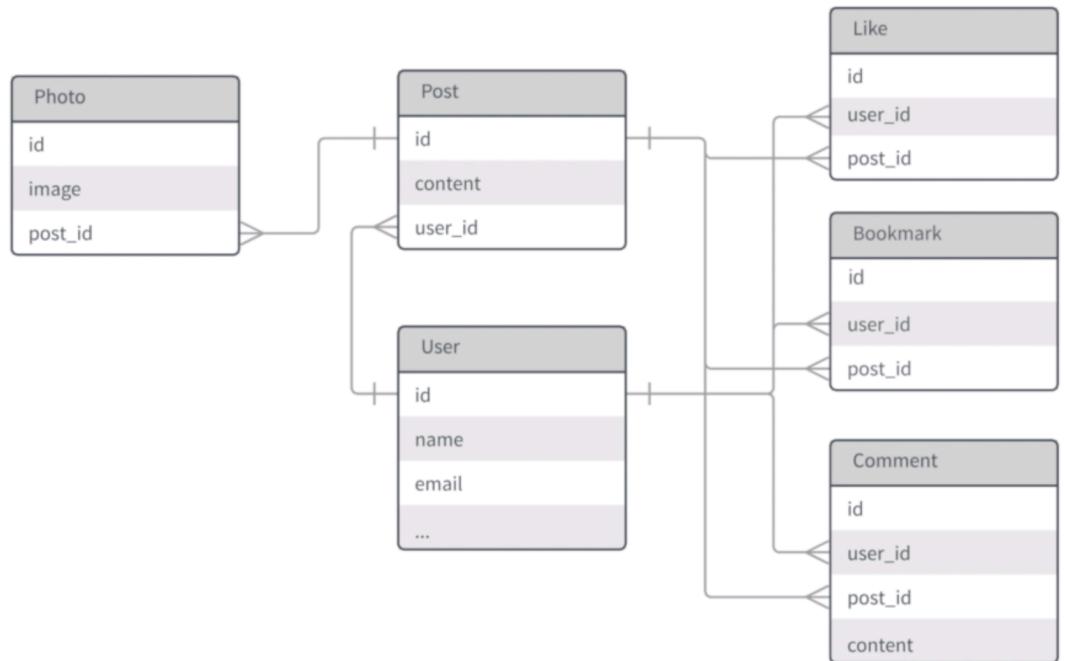
### Teil 5: Instagram Post Feature

Inhaltsverzeichnis AB151-05 .....	1
Modul 133: Instagram mit Rails bauen .....	1
Post modell.....	2
ERD – Instagram Datenbank .....	2
Post und Photo Modell.....	2
Upload image mit «gem CarrierWave» .....	4
«gem Cloudinary»: Photos in free cloud speichern .....	4
API-Key und API-Secret verschlüsseln.....	7
«gem Figaro» encrypt, decrypt .....	7
Post controller .....	9
Methode index .....	11
User access control .....	11
Methode show .....	11
Methode create.....	12
View index .....	12
Auftrag Post Controller: .....	14
Auftrag: Quicknote AB151-05.....	16

# Post modell

## ERD – Instagram Datenbank

Betrachten wir das Entity Relationship-Diagramm (ERD) unserer Instagram Datenbank, die wir nun schrittweise bauen müssen.



Gehen wir von User aus und formulieren zum Verständnis kurz die Beziehungen zwischen User, Post und Photo:

Ein User oder eine Userin kann mehrere Posts erfassen, ein Post wurde von einer bestimmten Userin oder einem bestimmten User erstellt. Ein Post kann mehrere Bilder beinhalten, ein Bild gehört immer zu einem Post.

## Post und Photo Modell

Erstellen wir einfach und schnell die Modelle für Post und Photo:

```
vmadmin@bmLP1:~/workspace/instagram$ rails generate model Post content:string user:references
Running via Spring preloader in process 10237
  invoke  active_record
  create   db/migrate/20181128122902_create_posts.rb
  create   app/models/post.rb
  invoke  test_unit
  create   test/models/post_test.rb
  create   test/fixtures/posts.yml
vmadmin@bmLP1:~/workspace/instagram$ rails generate model Photo image:string post:references
Running via Spring preloader in process 10260
  invoke  active_record
  create   db/migrate/20181128122943_create_photos.rb
  create   app/models/photo.rb
  invoke  test_unit
  create   test/models/photo_test.rb
  create   test/fixtures/photos.yml
vmadmin@bmLP1:~/workspace/instagram$ rails db:migrate
== 20181128122902 CreatePosts: migrating =====
-- create_table(:posts)
-> 0.0082s
== 20181128122902 CreatePosts: migrated (0.0083s) =====
== 20181128122943 CreatePhotos: migrating =====
-- create_table(:photos)
-> 0.0021s
== 20181128122943 CreatePhotos: migrated (0.0022s) =====
vmadmin@bmLP1:~/workspace/instagram$
```

Was bewirkt das Schlüsselwort `references` auf den Attributen `Post.user` und `Photo.post` in der Migration?

A large rectangular area with horizontal lines for handwriting practice. In the top right corner, there is a small icon of a pen.

Wie ist die Namensgebung der Attribute im TabellenSchema von Post und Photo?

A large rectangular area with horizontal lines for handwriting practice. In the top right corner, there is a small icon of a pen.

Ergänzen Sie die Beziehung zwischen Post und Photo. Wir erinnern uns, dass ein Post mehrere photos haben kann ☺.

A large rectangular area with horizontal lines for handwriting practice. In the top right corner, there is a small icon of a pen.

# Upload image mit «gem CarrierWave»

Informationen zur Uploadkomponente «gem CarrierWave» finden Sie unter  
<https://github.com/carrierwaveuploader/carrierwave>

Tragen Sie in das gemfile

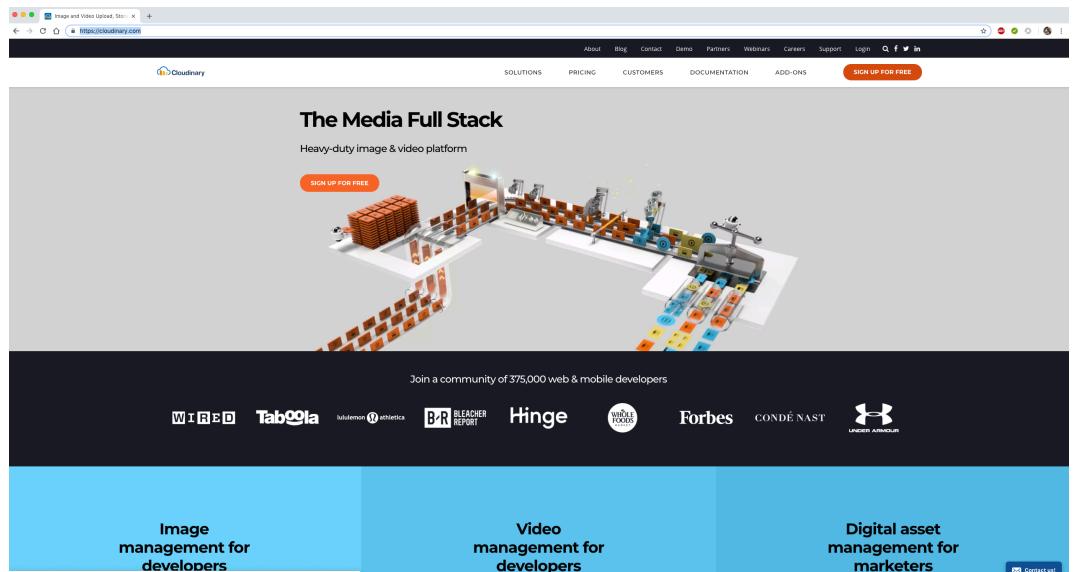
```
gem 'carrierwave', '~> 1.0'  
ein und installieren sie «gem CarrierWave» mit bundle install.
```

Danach generieren Sie in Rails den Uploader:

```
rails generate uploader Photo
```

## «gem Cloudinary»: Photos in free cloud speichern

Gehen Sie auf <https://cloudinary.com/>



Klicken Sie auf Pricing und erstellen Sie einen Account für “Free for as long as you need for developers and businesses”:

The screenshot shows the Cloudinary Pricing page. At the top, it says "Plans that scale with your growth". Below that are four plan options: "Free", "Plus", "Advanced", and "Custom". The "Free" plan is highlighted with a red circle around its row. The "Free" plan details are: "Free for as long as you need for developers and businesses", "\$0 per month", "No credit card required", "25 users", and a "SIGN UP FOR FREE" button. The "Plus" plan details are: "Great step up for growing image and video needs", "\$89 per month", "225 users", and a "BUY NOW" button. The "Advanced" plan details are: "Better choice for teams with rich media requirements", "\$224 per month", "600 users", and a "BUY NOW" button. The "Custom" plan details are: "Media at scale, enhanced security, custom configurations, dedicated support, and more", "Flexible Pricing", "Our pricing scales to any usage volume", and a "CONTACT US" button. The navigation bar at the top of this page is identical to the one on the previous screenshot.

# SIGN UP TO CLOUDINARY

Your name:

E-mail:

Password:

Country:

Phone: (Optional)

Company or site name: (Optional)

Primary interest:

Assigned cloud name: **xordiza**

By clicking Create Account you agree to Cloudinary's  
[Terms of Service](#) and [Privacy Policy](#)

**CREATE ACCOUNT**

Cloudinary is the media management platform for web and mobile developers. An end-to-end solution for all your image and video needs.

 File Upload & Storage

 Cloud Asset Management

 Image and Video Manipulation

 Optimization & Fast Delivery

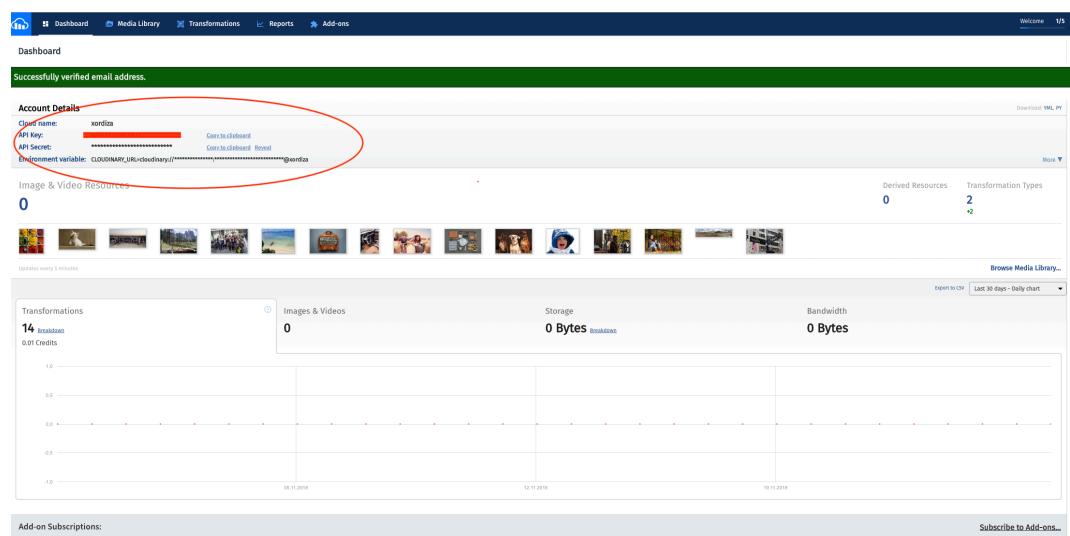
 Presentation

[LEARN MORE](#)

Sie müssen die Registrierung bestätigen. Hierzu sendet Ihnen Cloudinary eine Reply-Email mit entsprechendem Link.

Sollte alles geklappt haben, finden Sie auf dem Dashboard von Cloudinary folgende Angaben, die wir in unserer Rails-App wiederverwenden:

Cloud name: **xordiza**  
API Key: **\*\*\*\*\***  
API Secret: **\*\*\*\*\***  
Environment variable: **CLOUDINARY\_URL=cloudinary://\*\*\*:\*\*\*@xordiza**



The screenshot shows the Cloudinary dashboard with the following details:

- Account Details:** Cloud name: **xordiza**, API Key: **\*\*\*\*\***, API Secret: **\*\*\*\*\***, Environment variable: **CLOUDINARY\_URL=cloudinary://\*\*\*:\*\*\*@xordiza**. A red oval highlights this section.
- Dashboard Statistics:** 0 Image & Video Resources, 0 Derived Resources, 2 Transformation Types.
- Media Library:** 14 images, 0 Images & Videos, 0 Bytes Storage, 0 Bytes Bandwidth.
- Usage Graph:** A line graph showing usage over time from 08.11.2018 to 19.11.2018.
- Add-on Subscriptions:** None listed.

Tragen Sie in das gemfile

```
gem 'cloudinary'  
ein und installieren sie «gem Cloudinary» mit bundle install.
```

Erstellen Sie einen neuen Initializer für den Zugriff auf unsere Cloudinary Bildersammlung:

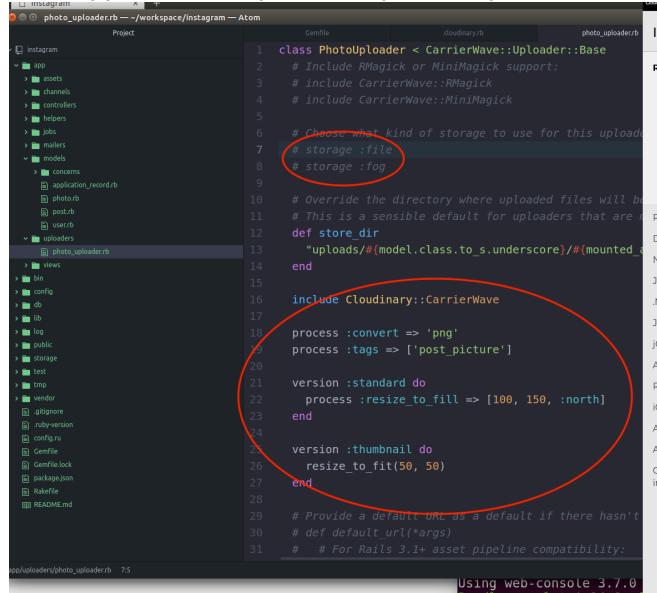
› config/initializers/cloudinary.rb

```
1 # Add configuration values here, as shown below.  
2 #  
3 # pusher_app_id: "2954"  
4 # pusher_key: 7381a978f7dd7f9a1117  
5 # pusher_secret: abdc3b896a0ffb85d373  
6 # stripe_api_key: sk_test_2J0l093x0yW72XUYJHE4Dv2r  
7 # stripe_publishable_key: pk_test_ro9jV5SNwGb1yYlQfzG17LHK  
8 #  
9 # production:  
10 #   stripe_api_key: sk_live_EeHnL644i6zo4Iyq4v1KdV9H  
11 #   stripe_publishable_key: pk_live_9lcLthxpSIHbGwmd094101XVU  
12   cloudinary_cloud_name: "xordiza"  
13   cloudinary_api_key: "██████████"  
14   cloudinary_api_secret: "██████████"
```

Cloudinary hat eine genaue Installationsanleitung für das Zusammenspiel mit der «gem CarrierWave»:

[https://cloudinary.com/documentation/rails\\_carrierwave#content](https://cloudinary.com/documentation/rails_carrierwave#content)

Wir können aus dieser Anleitung einige Code-Fragmente wiederverwenden und zwar in der Datei app/models/uploader/photo\_uploader.rb



```
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  
  # intended to be mounted onto a model.  
  def store_dir  
    "uploads/#{model.class.to_s.underscore}/#{mounted_as}"  
  end  
  
  include Cloudinary::CarrierWave  
  
  process :convert => 'png'  
  process :tags => ['post_picture']  
  
  version :standard do  
    process :resize_to_fill => [100, 150, :north]  
  end  
  
  version :thumbnail do  
    process :resize_to_fit => [50, 50]  
  end  
  
  # Provide a default URL as a default if there hasn't been  
  # any files uploaded yet.  
  # def default_url(*args)  
  #   # For Rails 3.1+ asset pipeline compatibility:  
  #   # "/images/fallback/" + [version_name, "x2x.png"].join('_')  
  # end
```

**Upload examples**  
Below is a short example that demonstrates using Cloudinary with CarrierWave. In this example, we use the Post model entity to support `attach_file` by the `'picture'` attribute (column) of the Post entity. To get started, first define a CarrierWave uploader class [CarrierWave documentation](#). In this example, we'll convert the uploaded image to a `Post` object's `picture` attribute. We define two additional transform 'standard' and 'thumbnail'. A randomly generated unique identifier is added to each image before it is stored in the database. In the following example, we explicitly define a public ID for the uploaded file.

```
class PictureUploader < CarrierWave::Uploader::Base  
  include Cloudinary::CarrierWave  
  process :convert => 'jpg'  
  process :tags => ['post_picture']  
  
  version :standard do  
    process :resize_to_fill => [100, 150, :north]  
  end  
  
  version :thumbnail do  
    process :resize_to_fit => [50, 50]  
  end  
  
  # Provide a default URL as a default if there hasn't been  
  # any files uploaded yet.  
  # def default_url(*args)  
  #   # For Rails 3.1+ asset pipeline compatibility:  
  #   # "/images/fallback/" + [version_name, "x2x.png"].join('_')  
  # end
```

The `'picture'` attribute of `Post` is simply a String (as DB migration). The `'PictureUploader'` class we've just defined:

```
class PictureUploader < CarrierWave::Uploader::Base  
  include Cloudinary::CarrierWave  
  
  ...  
  
  def public_id  
    return model.short_name  
  end  
end
```

Achten Sie darauf, dass Sie `storage: file` auskommentieren.

Die Parameter für die Bildgrößen sind zu klein. Passen Sie diese wie folgt an:

```
include Cloudinary::CarrierWave

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

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

version :thumbnail do
  resize_to_fit(100, 100)
end
```

Initialisieren Sie den Photo-Uploader in der Datei app/models/photo.rb

```
Gemfile           cloudinary.rb          photo_uploader.rb      photo.rb
1  class Photo < ApplicationRecord
2    belongs_to :post
3
4    mount_uploader :image, PhotoUploader
5  end
6
```

## API-Key und API-Secret verschlüsseln

Damit kein Hacker auf die dumme Idee kommt, unseren API-Key und das API-Secret zu klauen, verschlüsseln wir dieses von Beginn an. Es muss nicht einmal ein Hacker sein: Angenommen Sie pushen diese Rails-App auf GitHub, dann sind API-Key und API-Secret von Cloudinary öffentlich publiziert.

## «gem Figaro» encrypt, decrypt

Das «gem Figaro» ist unter <https://github.com/laserlemon/figaro> beschrieben. Grundsätzlich kann alles innerhalb einer Rails-App einfach und sicher verschlüsselt werden. Figaro nutzt hierzu eine 12-Faktoren Methodik.

Tragen Sie in das gemfile

```
gem 'figaro'
```

ein und installieren sie «gem figaro» mit `bundle install`. Damit «gem figaro» lauffähig ist muss noch der Befehl: `bundle exec figaro install` ausgeführt werden. Dieser Befehl erstellt eine unsichtbare Datei `app/config/application.yml`, wo Sie nun Cloud-Namen, API-Key und API-Secret eintragen können:

```
1 # Add configuration values here, as shown below.  
2 #  
3 # pusher_app_id: "2954"  
4 # pusher_key: 7381a978f7dd7f9a1117  
5 # pusher_secret: abdc3b896a0ffb85d373  
6 # stripe_api_key: sk_test_2J0l093x0yW72XUYJHE4Dv2r  
7 # stripe_publishable_key: pk_test_ro9jV5SNwGb1yYlQfzG17LHK  
8 #  
9 # production:  
10 #   stripe_api_key: sk_live_EeHnL644i6zo4Iyq4v1KdV9H  
11 #   stripe_publishable_key: pk_live_9lcLthxpSIHbGwmd094101XVU  
12   cloudinary_cloud_name: "xordiza"  
13   cloudinary_api_key: "  
14   cloudinary_api_secret: "  
15
```

Der Initializer für den Zugriff auf unsere Cloudinary Bildersammlung

config/initializers/cloudinary.rb kann nun einfach angepasst werden:

The screenshot shows a code editor with three tabs: Project, Gemfile, and application.yml. The Project tab shows a directory structure for an 'instagram' project with subfolders like app, bin, config, environments, and initializers. Inside the initializers folder, there is a file named 'cloudinary.rb'. The application.yml file is also visible. The Gemfile tab contains standard Ruby dependencies. The code editor's status bar indicates the current file is 'cloudinary.rb'. The code in 'cloudinary.rb' is as follows:

```
1 Cloudinary.config do |config|  
2   config.cloud_name = ENV["cloudinary_cloud_name"]  
3   config.api_key = ENV["cloudinary_api_key"]  
4   config.api_secret = ENV["cloudinary_api_secret"]  
5   config.cdn_subdomain = true  
6 end  
7
```

Wenn Sie alles richtig gemacht haben, ist nun der Zeitpunkt, den Server neu zu starten und zu testen, ob alle Llamas in Pyjamas sind und alles ordnungsgemäß läuft.

The screenshot shows a terminal window with the command 'rails s' being run. The output shows the server booting up with Puma, starting in development mode, and listening on port 4000. It also mentions the codename 'Llamas in Pajamas'.

```
vmadmin@bmLP1: ~/workspace/instagram$ rails s  
=> Booting Puma  
=> Rails 5.2.1 application starting in development  
=> Run `rails server -h` for more startup options  
Puma starting in single mode...  
* Version 3.12.0 (ruby 2.4.1-p111), codename: Llamas in Pajamas  
* Min threads: 5, max threads: 5  
* Environment: development  
* Listening on tcp://0.0.0.0:4000  
Use Ctrl-C to stop
```

# Post controller

Erinnern wir uns kurz an die HTTP-Verben für eine restful API:

- › GET wird benötigt, um eines oder mehrere Objekte abzufragen.
- › POST wird benötigt, um ein neues Objekt zu erzeugen.
- › PATCH/PUT wird benötigt, um ein bestehendes Objekt zu ändern.
- › DELETE wird benötigt, um ein Objekt zu löschen.

Rails controller action kennt für die Anwendung von restful API sieben Methoden:

- › index wird benötigt, um alle Objekte aufzulisten.
- › show wird benötigt, um ein Objekt anzuzeigen.
- › new wird benötigt, um ein Formular zur Erstellung eines Objekts anzuzeigen.
- › create wird benötigt, um ein neues Objekt zu erzeugen.
- › edit wird benötigt, um ein Formular zur Erstellung eines Objekts anzuzeigen.
- › update wird benötigt, um ein bestehendes Objekt zu ändern.
- › destroy wird benötigt, um ein Objekt zu löschen.

Betrachten wir nun die Beziehungen zwischen den Modellen Post und Photo:

- › Ein Photo gehört zu einem Post.
- › Ein Post kann kein, ein oder mehrere Photos beinhalten.

Solche Beziehungen lassen sich als Routen im Rails Router (routes.rb) abbilden:

```
resources :posts, only: [:index, :show, :create, :destroy] do
  resources :photos, only: [:create]
```

Wir sagen mit resources, dass alle HTTP-Verben von Posts (Plural) auf die Controller-Methoden index, show, create und destroy anwendbar sind und da ein Photo zu einem Post gehört, kann man die Erstellung eines Photo (create) als eingebettete Ressource (nested resources) von Posts deklarieren.

Lesen Sie Kapitel 2.2 und Kapitel 2.7 in <https://guides.rubyonrails.org/routing.html>

Ergänzen Sie app/config/routes.rb:

```
Gemfile application.yml routes.rb user.rb post.rb posts_controller... index.html.erb cloudinary.rb photo
1 Rails.application.routes.draw do
2   root 'pages#home'
3   devise_for :users, controllers: { registrations: 'registrations' }
4   # For details on the DSL available within this file, see http://guides.rubyonrails.org/routing.html
5   resources :users, only: [:index, :show]
6
7   resources :posts, only: [:index, :show, :create, :destroy] do
8     resources :photos, only: [:create]
9   end
10 end
```

Als nächstes müssen wir das Löschverhalten in den Modellen festhalten:

- › Wenn ein Benutzer (User) gelöscht wird, sollen alle seine Posts auch gelöscht werden.  
`User has_many :posts, dependent: :destroy`
- › Wenn ein Post gelöscht wird, sollen alle Photos in dem Post mitgelöscht werden.  
`Post has_many :photos, dependent: :destroy`

Welche Anpassung müssen Sie dazu in app/models/user.rb machen:



Handwriting practice area for notes.

Welche Anpassung müssen Sie dazu in app/models/post.rb machen:



Handwriting practice area for notes.

Als nächstes erstellen wir den Post Controller. Erstellen Sie manuell (Datei neu) im Ordner app/controllers/ die Datei posts\_controller.rb.

Bauen Sie folgendes Grundgerüst in posts\_controller.rb:

```
Gemfile      application.yml      routes.rb      user.rb      post.rb      posts_controller.rb •  
1  class PostsController < ApplicationController  
2  
3    def index  
4  
5    end  
6  
7    def create  
8  
9    end  
10  
11   def show  
12  
13   end  
14  
15   def destroy  
16  
17   end  
18  
19   private  
20  
21   def find_post  
22  
23   end  
24  
25   def post_params  
26  
27   end  
28 end|
```

## Methode index

In der Methode index wollen wir nur die 10 letzten Posts inklusive aller dazugehörigen Photos anzeigen:

```
5  def index
6    @posts = Post.all.limit(10).includes(:photos)
7    @post = Post.new
8  end
```

`Post.new` benötigen wir später, wenn wir einen Post erfassen wollen.

## User access control

Wir müssen sicherstellen, dass nur angemeldete Benutzerinnen und Benutzer auf die Methoden im Post Controller `posts_controller.rb` zugreifen können. Zudem soll geprüft werden, ob ein Post überhaupt existiert und falls nein, soll eine flash-notification entsprechend informieren. Diese Anweisungen kommen zuerst.

```
1  class PostsController < ApplicationController
2    before_action :authenticate_user!
3    before_action :find_post, only: [:show, :destroy]
4
5    ...
6  end
```

Die private Methode `find_post` müssen wir nun programmieren. Hier ein einfacher und leicht lesbarer Vorschlag:

```
37  def find_post
38    @post = Post.find_by id: params[:id]
39
40    return if @post
41    flash[:danger] = "Post doesn't exist!"
42    redirect_to root_path
43  end
```

## Methode show

Die Methode `show` bekommt von `@post` alle Photos. `@post` wird als Instanzvariable in der Methode `find_post` definiert.

```
27  def show
28    @photos = @post.photos
29  end
```

## Methode create

Die Methode `create` sieht wie folgt aus:

```
10  def create
11      @post = current_user.posts.build(post_params)
12      if @post.save
13          if params[:images]
14              params[:images].each do |img|
15                  @post.photos.create(image: img)
16              end
17          end
18
19          redirect_to posts_path
20          flash[:notice] = "Saved ..."
21      else
22          flash[:alert] = "Something went wrong ..."
23          redirect_to posts_path
24      end
25  end
```

Die Methode `create` benötigt die Methode `post_params`

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

## View index

Erstellen Sie nun einen neuen Ordner

app/views/posts

und die Datei

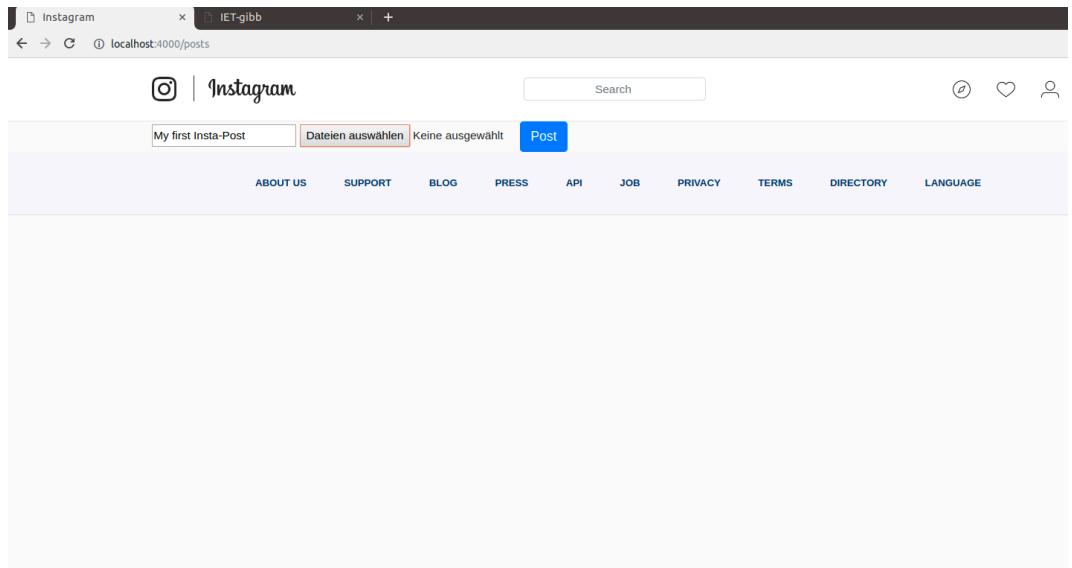
app/views/posts/index.html.erb

Wir erfassen in `app/views/posts/index.html.erb` ein `form_tag` für den Bilderupload und zum Anzeigen eine doppelte Iteration durch die Posts und deren Photos:

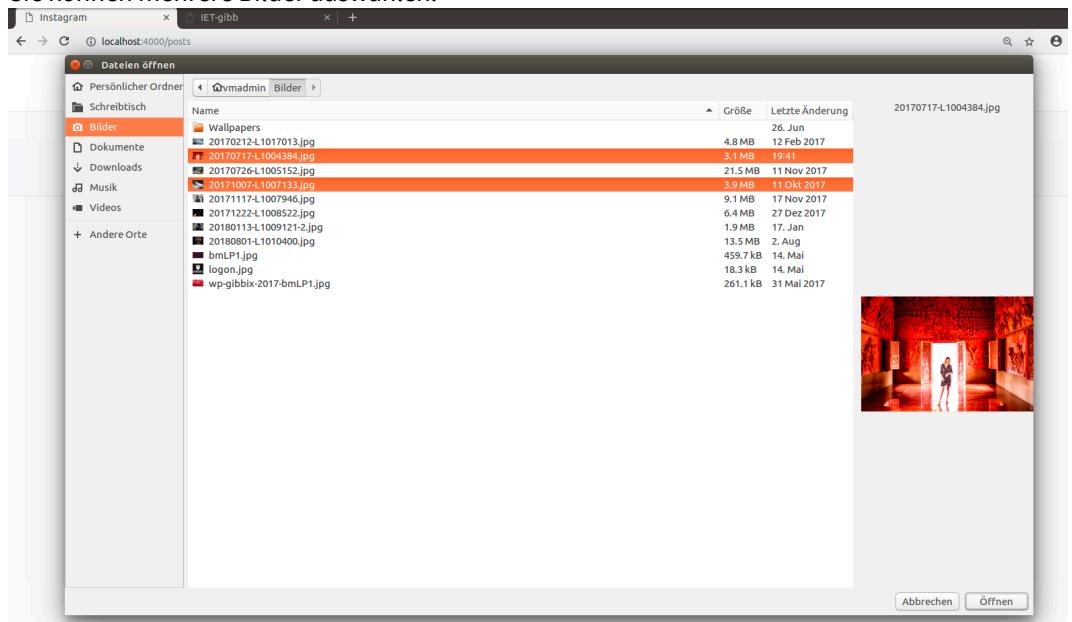
```
Gemfile           application.yml       routes.rb        user.rb        post.rb        posts_controller...   index.html.erb
1  <%= form_for @post, :html => { :multipart => true} do |f| %>
2      <%= f.text_field :content, class: "control-label" %>
3      <%= file_field tag "images[]", type: :file, multiple: true %>
4      <%= f.submit "Post", class: "btn btn-primary" %>
5  <% end %>
6
7  <%@posts.each do |post|%>
8      <p><%=post.content%></p>
9      <%post.photos.each do |photo|%>
10         <%=image_tag photo.image.url(:standard)%>
11     <%end%>
12 <%end%>
13
```

Starten Sie den Server neu und wenn alles geklappt hat, können Sie nun einen ersten «Post with multiple photos» testen:

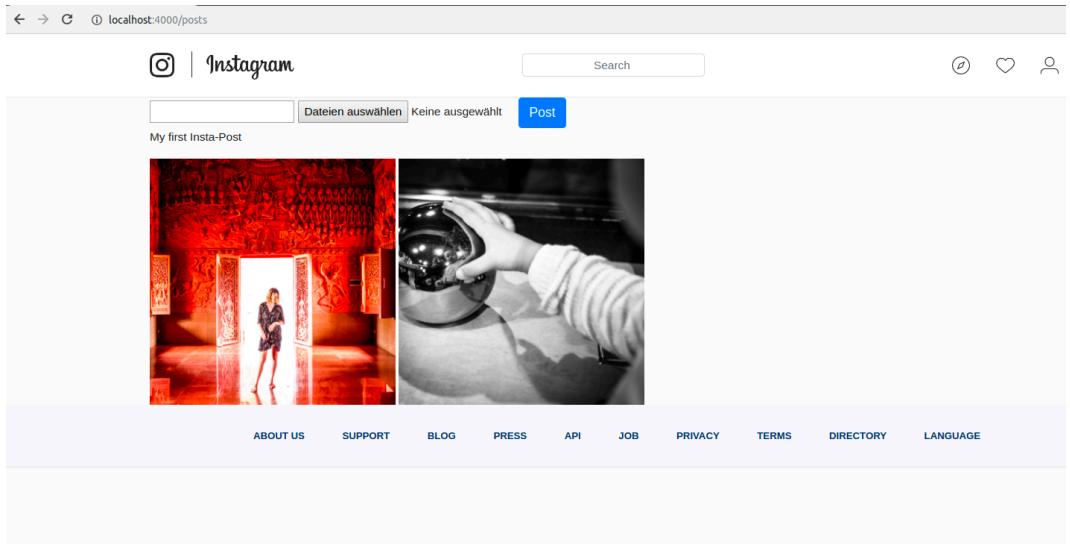
<http://localhost:4000/posts>



Sie können mehrere Bilder auswählen:



...et voilà:



## Auftrag Post Controller:

Erklären Sie die create-Methode im Post Controller:

Beginnen Sie mit dieser Zeile:

```
@post = current_user.posts.build(post_params)
```

Um zu verstehen was passiert, können Sie die Rails Console zur Hilfe nehmen und das Kommando:

```
User.first.posts.build ausführen.
```

Erklären Sie den Rest der create-Methode im Post Controller:

```
10  def create
11    @post = current_user.posts.build(post_params)
12    if @post.save
13      if params[:images]
14        params[:images].each do |img|
15          @post.photos.create(image: img)
16        end
17      end
18
19      redirect_to posts_path
20      flash[:notice] = "Saved . . ."
21    else
22      flash[:alert] = "Something went wrong . . ."
23      redirect_to posts_path
24    end
25  end
```



---

---

---

---

---

Erklären Sie die `index`-Methode im Post Controller:

```
5   def index
6       @posts = Post.all.limit(10).includes(:photos)
7       @post = Post.new
8   end
```

---

---

---

---

---

Erklären Sie folgende Zeile:

```
3   before_action :find_post, only: [:show, :destroy]
```

---

---

---

---

---

# Auftrag: Quicknote AB151-05

Alle Aufträge des Typen Quicknote sind Bestandteil der Bewertung. Erstelle Sie eine Quicknote von max. 4 A4-Seiten und geben Sie diese in PDF Form gemäss Zeitangabe der Lehrperson ab. Angedacht sind max. 4 Lektionen seit Abgabe dieses Dokumentes.

Titel der Quicknote: *Klasse\_Name\_Vorname\_QN\_AB151-05.zip*

Das Archiv beinhaltet:

- › **Quicknote:** Klasse\_Name\_Vorname\_QN\_AB151-05.pdf
- › **Code:** ganze Instagram APP als Archiv

Was beinhaltet die Quicknote AB151-05?

## Zusammenfassung AB151-05

Die Quicknote soll eine kurze prägnante Zusammenfassung des Dokuments AB151-05 beinhalten und einen Überblick über die vorgestellten Techniken, Methoden und Konzepte beinhalten.

Achten Sie darauf, dass alle wesentliche Themen vollständig erwähnt und wenn möglich in Zusammenhang gebracht sind.

Beantworten Sie folgende Fragen zu Anwendungszweck und Selbstreflexion:

### Anwendungszweck:

1. Wie und wo können die vorgestellten Techniken, Methoden und Konzepte in einer Rails-App angewandt werden?
2. Was sind Vorteile und was sind Nachteile?

### Selbstreflexion:

3. Was habe ich gelernt?
4. Was hat mich behindert?
5. Was habe ich nicht verstanden?
6. Was kann ich beim Studium besser machen?

### Abschliessende Reflexion über das Gelernte:

7. Schreiben Sie eine persönliche Schlussfolgerung über den Lerninhalt in 2-3 Sätzen.