Ruby on Rails 101 — Chapter 5

written by Thomas Mak.

In this chapter, we will build a photo gallery with management.

1. Setting up the project
2. Building the photo model
3. Building the photo controller and view
4. Using Partial View
5. Dynamic title with content_for
6. Adding album resource
7. Associate photo to album
8. Introducing asset pipeline

Setting up photo gallery project

We are going to use a new project. Create a new project with the following commands

```
$ rvm use 1.9.3
$ rails _3.2.8_ new photo_gallery
$ cd photo_gallery
$ bundle install
```

Note: when creating new project, use `-d mysql` to use mysql as default database engine.

Building the photo model

Generating model

We always define the model first. As a gallery, our photo model will contian the attachment and a title describing it. The attachment will be done by the paperclip gem. Let's create the photo model with titie only by the following commands.

```
$ rails generate model photo title:string
    invoke  active_record
    create
db/migrate/20131003143245_create_photos.rb
    create    app/models/photo.rb
    invoke    test_unit
    create      test/unit/photo_test.rb
    create      test/fixtures/photos.yml

$ rake db:migrate
==  CreatePhotos: migrating =======================
-- create_table(:photos)
   -> 0.0056s
==  CreatePhotos: migrated (0.0059s) ==============
```

Installing paperclip

Now it is time to attach a file to our photo model.

Let's add the paperclip gem to the `Gemfile`.

```
gem 'paperclip'
```

Then bundle install it

```
$ bundle install
```

If the imagemagick isn't installed yet, use homebrew to install it with the following command.

```
$ brew install imagemagick
```

Adding attachment

Now it is time to generate a paperclip attchment to our photo model. Run the following paperclip generation command.

```
$ rails generate paperclip photo file
      create
db/migrate/20131003150131_add_attachment_file_to_photos.
```

The generator creates a database migration, that we need to push it to the database.

```
$ rake db:migrate
==  AddAttachmentFileToPhotos: migrating
============================
-- change_table(:photos)
   -> 0.0033s
==  AddAttachmentFileToPhotos: migrated (0.0035s)
===================
```

And now we can map the database in the model. Make the `photo.rb` file matches the following snippet.

```
class Photo < ActiveRecord::Base
  attr_accessible :title, :file
  has_attached_file :file, styles: { medium:
"300x300>", thumb: "100x100>" }, default_url:
"/images/:style/missing.png"
end
```

Resizing options

Note: We can create different resize options in the styles option hash. Take the following option as example, it defined 4 resize options so there are total 5 sizing including the original dimension.

```
has_attached_file :file,
  styles: {
    large: "1000x1000",
    medium: "800x800",
    thumb: "300x300",
    square: "300x100#" }, default_url:
"/images/:style/missing.png"
```

Note2: The style dimension option is a string which follow exactly the ImageMagick germetry format:

> means resizing to make width and/or height matching the given dimension.

# means cropping to fit the exact dimension.

Building the photo controller and views

Routing

We want RESTful URLs for our photo resource. Here is the `routes.rb` file.

```
PhotoGallery::Application.routes.draw do
  resources :photos
end
```

Generating controller

It is time to create the contorller.

```
$ rails generate controller photos
```

Creating photo

The controller

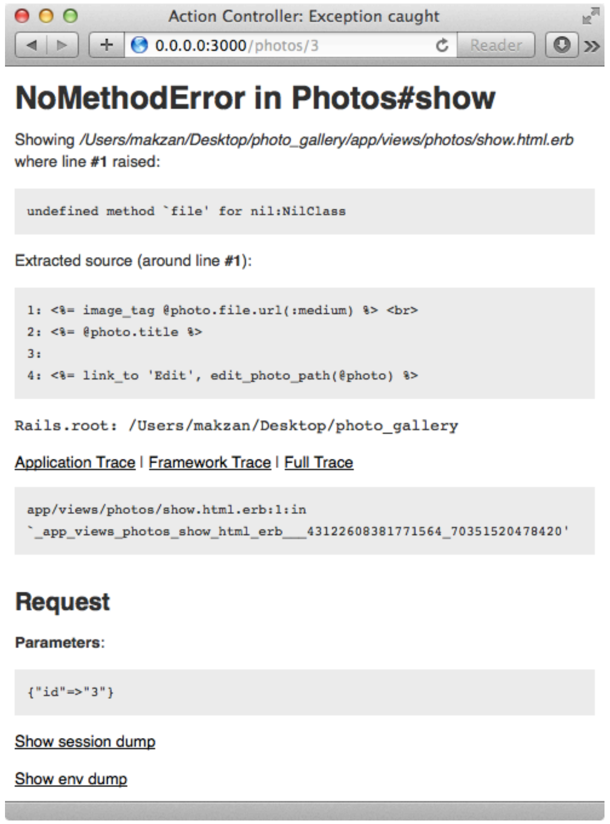
```
class PhotosController < ApplicationController
  def new
    @photo = Photo.new
  end

  def create
    @photo = Photo.new params[:photo]
    if @photo.save
      redirect_to @photo
    else
      render :new
    end
  end
end
```

views/photos/new.html.erb

```
<%- form_for @photo, html: { :multipart => true } do
|f| %>
  <p>
    <%= f.label :title %> <br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :file %> <br>
    <%= f.file_field :file %>
  </p>

  <p>
    <%= f.submit 'Upload Photo' %>
  </p>
<%- end %>
```

Let’s try the function in browser with the following steps.

1. In terminal, in the current project, run `rails server`.
2. Open `http://0.0.0.0:3000/photos/new` in web browser.
3. Select an image file and put in the title.
4. Click the ‘Upload Photo’ button.
5. Now we should see an error of `NoMethodError`.
6. If we check the URL, it is redirected to the photo showing URL with the newly created photo ID.
7. This is normal because we haven’t implemented the `show` method in controller yet.



NoMethodError in Photos#show

Showing /Users/makzan/Desktop/photo_gallery/app/views/photos/show.html.erb where line #1 raised:

```
undefined method `file' for nil:NilClass
```

Extracted source (around line #1):

```
1: <%= image_tag @photo.file.url(:medium) %> <br>
2: <%= @photo.title %>
3:
4: <%= link_to 'Edit', edit_photo_path(@photo) %>
```

```
Rails.root: /Users/makzan/Desktop/photo_gallery
```

Application Trace | Framework Trace | Full Trace

```
app/views/photos/show.html.erb:1:in
`_app_views_photos_show_html_erb___431226083817715 64_70351520478420'
```

Request

Parameters:

```
{"id"=>"3"}
```

Show session dump

Show env dump

Showing photo Then we add the show feature

The photos_controller.rb

```
def show
  @photo = Photo.find params[:id]
end
```

And its related view: views/photos/show.html.erb

```
<%= image_tag @photo.file.url(:medium) %> <br>
<%= @photo.title %>
```

Editing photo Since the edit form shares the same code from the create form, we will extract the form into a common file.

Move the entire `form_for` block to a new file: `views/photos/_form.html.erb`

Now the `views/photos/new.html.erb` becomes

```
<h1>Creating New Photo </h1>
<%= render 'form' %>
```

And the `views/photos/edit.html.erb` file

```
<h1>Edit Photo </h1>
Existing image: <br>
<%= image_tag @photo.file.url(:medium) %>

<%= render 'form' %>
```

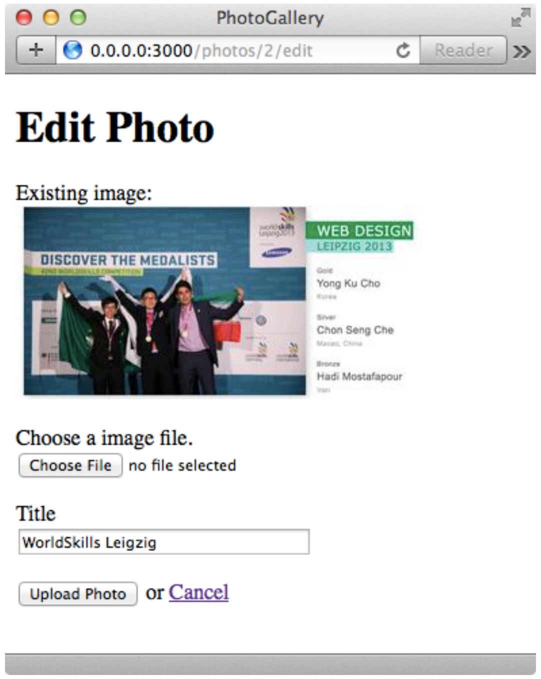
Optionally we may want to let site admin edit the photo from the user interface, we can do that by adding a link to the edit path in the `views/photos/show.html.erb` file.

```
<%= link_to 'Edit', edit_photo_path(@photo) %>
```

And the `edit` and `update` controller method in the `photos_controller.rb` file.

```
def edit
  @photo = Photo.find params[:id]
end

def update
  @photo = Photo.find params[:id]

  if @photo.update_attributes params[:photo]
    redirect_to @photo
  else
    render :edit
  end
end
```

Let’s try the function in browser and we should be able to create and edit photos. Here is a screenshot of the editing screen:


PhotoGallery

0.0.0.0:3000/photos/2/edit

Edit Photo

Existing image:

Choose a image file.

Choose File no file selected

Title

WorldSkills Leigzig

Upload Photo or Cancel

Using partial file

View filename that starts with underscore are called **partial**:

_form.html.erb

_nav.html.erb

When using `render` function, we specify the partial name without the underscore:

```
<%= render 'nav' %>
```

If the partial file is not in the same folder, say `views/layouts/_nav.html.erb`, we can include the path.

```
<%= render 'layouts/nav' %>
```

Sometimes we need to pass variable into partial. We can do that by specifing the locals.

```
<%= render partial: 'layouts/nav', locals: {key:
'value'} %>
```

Dynamic title with content_for

Now all the our web pages share the same title, which is it a good practice for UX and SEO reason.

So we want dynamic title on each page. In the way, the title is a variable that varies depends on the current showing page. Take photo showing as example, the title may be the photo title.

We learnt to define @variable in controller that passes to the view. But title is responded by view instead of controller. Let's use the view approach – `content_for`

Time for actions

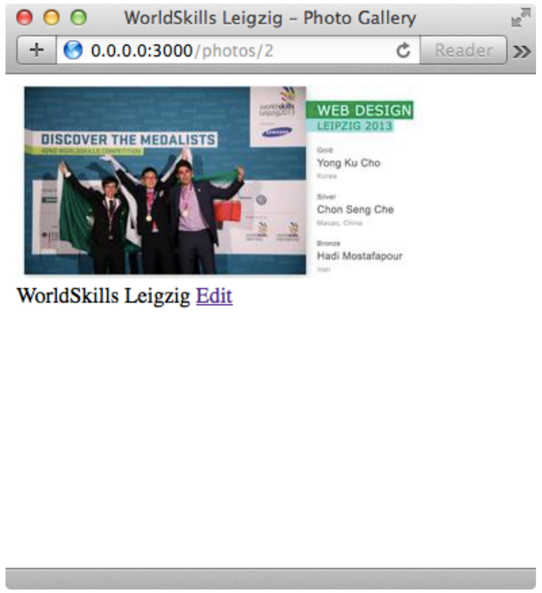
In the `views/layouts/application.html.erb` file

```
<title>
  <%- if content_for? :page_title -%>
    <%= yield :page_title %> — Photo Gallery
  <%- else -%>
    Photo Gallery
  <%- end -%>
</title>
```

in the `views/photos/show.html.erb` file, prepend the following code snippet at the beginning, before any existing content.

```
<% content_for :page_title do
  @photo.title
end %>
```

And the result screenshot. Note how the title reflects the photo title.



WorldSkills Leigzig Edit

Explaining content_for

`content_for` is defined in view. It is used for defining specific content in that view.

For example, a specific page — let say 'about' — may want to include one special CSS file. Since CSS is linked inside the tag, we can't link it anywhere inside that about view file because the view file is included into the tag. That's where we need the `content_for`.

In the `application.html.erb` layout file

```
<!DOCTYPE html>
<html>
<head>
  <%- if content_for? :special_css_files -%>
    <%= yield :special_css_files %>
  <%- end -%>
</head>
<body>

<%= yield %> <!-- this is where the view file
included. -->

</body>
</html>
```

Now assume our about view is `about.html.erb`, we can define a `special_css_files` section that will be included (**yield**) in the layout inside the section.

```
<% contentfor :specialcss_files do %>
  <link rel='stylesheet' type='text/css' media='all'
href='special.css'>
<% end %>
<!-- The rest of about content goes here -->
```

Adding album resource

Generating model

It is very similar to creating photo model. Each album has a title and has many photos.

```
$ rails generate model album title:string
    invoke  active_record
    create
db/migrate/20131004132907_create_albums.rb
    create    app/models/album.rb
    invoke    test_unit
    create      test/unit/album_test.rb
    create      test/fixtures/albums.yml
```

And we can do the association at the same time.

```
$ rails generate migration AddAlbumIdToPhoto
album_id:integer
    invoke  active_record
    create
db/migrate/20131004133355_add_album_id_to_photo.rb
```

Then we can migrate two migrations at once.

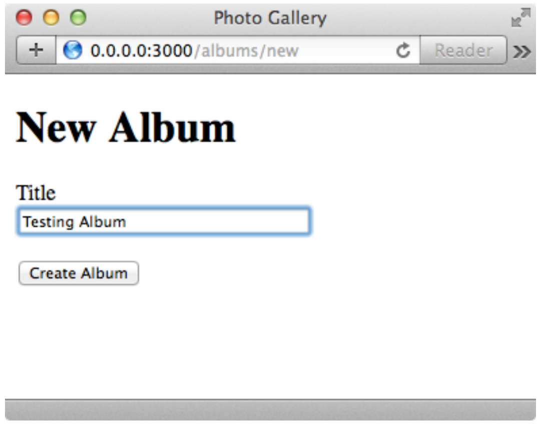
```
$ rake db:migrate
==  CreateAlbums: migrating
=========================================
-- create_table(:albums)
   -> 0.0024s
==  CreateAlbums: migrated (0.0027s)
================================

==  AddAlbumIdToPhoto: migrating
====================================
-- add_column(:photos, :album_id, :integer)
   -> 0.0021s
==  AddAlbumIdToPhoto: migrated (0.0024s)
===========================
```

CRUD controller and views

Generating controller

```
$ rails generate controller albums
```

```
    create  app/controllers/albums_controller.rb
    invoke  erb
    create    app/views/albums
    invoke  test_unit
    create
test/functional/albums_controller_test.rb
    invoke  helper
    create    app/helpers/albums_helper.rb
    invoke    test_unit
    create
test/unit/helpers/albums_helper_test.rb
    invoke  assets
    invoke    coffee
    create
app/assets/javascripts/albums.js.coffee
    invoke    scss
    create      app/assets/stylesheets/albums.css.scss
```



New Album

Title

Testing Album

Create Album

The controller

```
class AlbumsController < ApplicationController
  def new
    @album = Album.new
  end

  def create
    @album = Album.new params[:album]
    if @album.save
      redirect_to @album
    else
      render :new
    end
  end
end
```

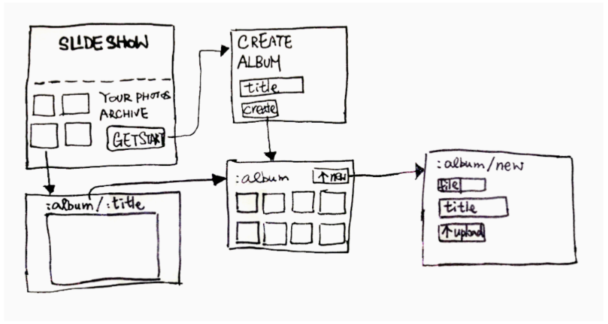
The `views/albums/new.html.erb` file.

```
<h1>New Album</h1>

<%= form_for @album do |f| %>
  <p>
    <%= f.label :title %> <br>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.submit 'Create Album' %>
  </p>
<% end %>
```

Associate photo to album

Routes Planning

Before we associate the album and photo model, it would be better to plan the entry point — routes. By doing that, we can have a clear blueprint on how the app should redirect.



Target URL:

```
/                                    home page
/albums/new                          create album
/albums/:album_id                    show spceific
album with photos
/albums/:album_id/photos/new         create photo
/albums/:album_id/photos/:photo_id   show specific
photo
```

Important: Do not underestimate the power of URL design. It acts as the entry point of your web app. Your users bookmark it; They share it; They hack it; And Google indexes it.

Do not under estimate the power of URL design

routes.rb

We want to update the `routes.rb` file to reflect our URL design.

```
PhotoGallery::Application.routes.draw do
  resources :albums do
    resources :photos
  end
end
```

Model

The association starts in the model.

`album.rb` file.

```
class Album < ActiveRecord::Base
  attr_accessible :title
  has_many :photo
end
```

`photo.rb` file.

```
class Photo < ActiveRecord::Base
  attr_accessible :title, :file, :album_id
  has_attached_file :file, styles: { medium:
"300x300>", thumb: "100x100>" }, default_url:
"/images/:style/missing.png"
  belongs_to :album
end
```

Controller

Now photos is nested inside albums. This means all our photo actions will need the `@album` instance. We can do that by using `before_filter` (or `before_action` in rails 4).

In the `photos_controller.rb` file, add the before_filter line at the beginning and the private methods at the end, before the class `end`.

```
before_filter :set_album

private
def set_album
  @album = Album.find params[:album_id]
end
```

Then we change all the `Photo` reference to `@album.photos` because every photos collection querying is bound by the @album.

before_filter

Note: before_filter means running the given method before running every actions. We can use :only action to run it only in a given list of methods. Or we can use :except for a list of methods that doesn't run the given method.

Example on using before_filter:

```
before_filter :authorize, except: [:index, :show]
before_filter :authorize, only: :delete
```

And here is the full `photos_controller.rb` file dump, in case you failed to complete the changes:

```
class PhotosController < ApplicationController

  before_filter :set_album

  def show
    @photo = @album.photos.find params[:id]
  end

  def new
    @photo = @album.photos.new
  end

  def create
    @photo = @album.photos.new params[:photo]
    if @photo.save
      redirect_to @album
    else
      render :new
    end
  end

  def edit
    @photo = @album.photos.find params[:id]
  end

  def update
    @photo = @album.photos.find params[:id]

    if @photo.update_attributes params[:photo]
      redirect_to @album
    else
      render :edit
    end
  end

  private
  def set_album
    @album = Album.find params[:album_id]
  end

end
```

View And then the views update.

`views/photos/show.html.erb` file.

```
<%= link_to 'Edit', edit_photo_path(@photo) %>

# becomes =>

<%= link_to 'Edit', edit_album_photo_path(@album,
@photo) %>
```

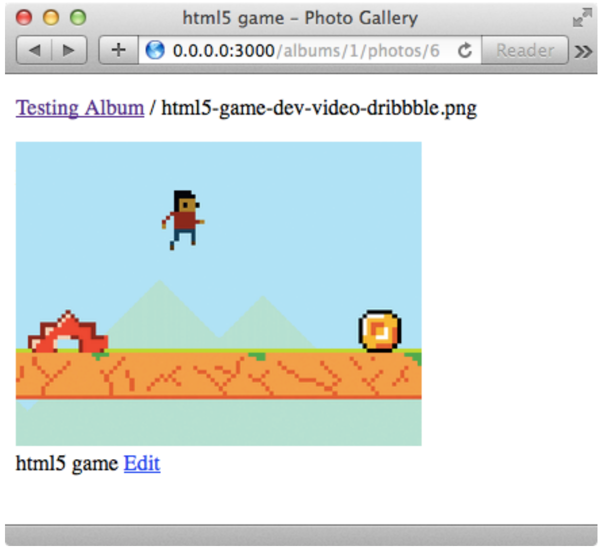
As a bonus, we can also create a breadcrumb when showing the photo:

`views/photos/show.html.erb` file.

```
<p>
  <%= link_to @album.title, @album %>
  /
  <%= @photo.file_file_name %>
</p>
```

Note: As documented in the paperclip gem, we can use `<attachment>_file_name` to refer to the filename string.

This is how the breadcrumb looks like:



Testing Album / html5-game-dev-video-dribbble.png

html5 game Edit

`views/photos/_form.html.erb` file.

```
<%= form_for [@album, @photo], html: { :multipart =>
true } do |f| %>

  ...

  <p>
    <%= f.submit 'Upload Photo' %>
    or
    <%= link_to 'Cancel', [@album, @photo] %>
  </p>

<%- end %>
```

Listing the photos inside album.

First, we get the @album reference in the `albums_controller.rb` file.

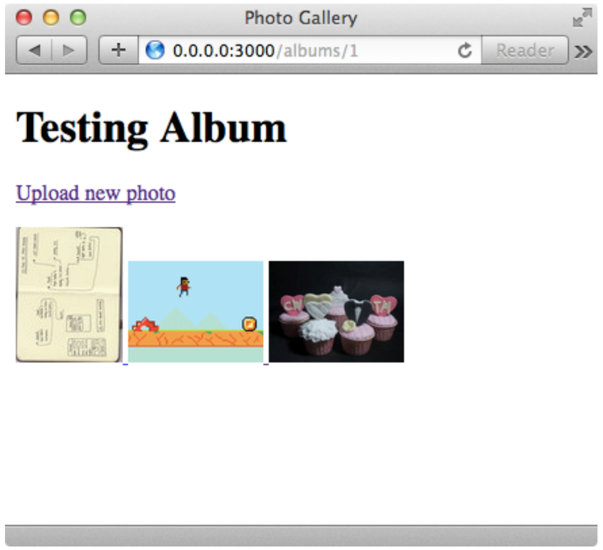
```
class AlbumsController < ApplicationController
  def show
    @album = Album.find params[:id]
  end

  ...
end
```

Now we can list all the photos in the album.

Add the `views/albums/show.html.erb` file with the following HTML/ERB code

```
<h1><%= @album.title %></h1>

<p><%= link_to 'Upload new photo',
new_album_photo_path(@album) %></p>

<% @album.photos.each do |photo| %>
  <%= link_to [@album, photo] do %>
    <%= image_tag photo.file.url(:thumb) %>
  <% end %>
<% end %>
```

When test the code in browser, the albums list the photos and show the photo after clicking on it.



Introducing asset papeline

The styling, scripts, images are called **assets**.

Managing and loading assets has been a big issue for web designers.

Assets issues

There are a lot of inconvenience when managing assets ourselves, here are the major issues.

1. Source file on dev; minification files on production.
2. Separated files for components in dev; one file for HTTP request optimization in production.
3. Writing preprocessing scripts in dev; requiring HTML/CSS/JS in production.

Asset pipeline solves all the above issues.

And one mode thing, finderprint in asset pipeline optimizes the assets caching and invalidation.

asset pipeline optimizes the assets caching and invalidation

For more information on finderprint, check the rails guide.

Assets location

Normally, the assets are located in the `app/assets/` folder. By default, JS and CSS files are automatically loaded when inside the `app/asstes/javascripts/` and `app/assets/stylesheets/` folder.

Image files inside `app/assets/images` folder can be reference with `image_tag`:

```
<%= image_tag  'file_name_here.png' %>
```

Third party assets can be placed in the `app/vendor/assets/` forder. The files here will be correctly loaded.

By default, rails comes with jQuery ready so we do not need to include our own jQuery script.

Note: in older rails version, jQuery is not a default loaded library.

What is preprocessing?

Preprocessing let us write the HTML/CSS/JS code in a higher level. You can think it improves those languane to some points. Leting us writing them easier, faster, less bugs.

For HTML, we have `HAML` and `Slim` choices.

For CSS, we have `Scss`, `Less` and `Stylus`.

For JavaScript, we have `CoffeeScript`.