AIW Final Report
Fall 2014

Group 12

Class: **2C-11**

Members:

Ta Van Phuong

Do Thanh Tung

Nguyen Nhu Binh

Tran Manh Cuong

Nguyen Manh Cong

Table of Contents

- Introduction
- Overview
- Installation
  - Rails Installation on Windows
- Quick start
  - Create New Project
  - Modify Gemfile
  - Run bundle install
  - Configure database
- Getting Up and Running
  - Rails Controller - ActionController
    - Implementing the Listing method
    - Implementing the New method
    - Implementing the Create method
    - Implementing the Update method
    - Implementing the Delete method
    - Defining strong parameters
  - Configuring routes
  - The first form
  - Rails Active Records - Models
    - Creating Active Record files
    - Run bundle install
- Associating Models and Application Administration
  - Associating Models
  - Active Admin
    - Configuring Active Admin
- Integrating CKEditor
- Uploading image using Carrierwave
- Adding some validation
  - Numeric validity
  - Validate presence and uniqueness
- More tiny customizations
  - Web title
  - HTTP Basic Authentication
  - Pagination
- Web content
- Screenshots
- Conclusion

I. Introduction

Ruby is a scripting language designed by Yukihiro Matsumoto, also known as Matz. It runs on a variety of platforms, such as Windows, Mac OS, and the various versions of UNIX.

Ruby on Rails is an extremely productive web application framework written in Ruby by David Heinemeier Hansson. This is a open source Ruby framework for developing database-backed web applications.

II. Overview

Our final project's task is to finish a real web that delivers the content to users, lets them use the web services and will be put online for demonstration. Our topic is closely related to Arts - **Famous Paints Profile** - containing a collection of the most famous paints in the world with precise information represented in Vietnamese language.

This website covers the following features:

- Written in Ruby on Rails and MySQL
- Designs using Bootstrap framework and a little customized style.css
- Contains a simple backend to manage data - ActiveAdmin gem as the administration site
- Uses MySQL as DBMS

III. Installation

To develop a web application using Ruby on Rails Framework, install the following software:

- Ruby
- The Rails framework
- A Web Server
- A Database System

We assume that you already have installed a Web Server and Database System on your computer. You can always use the WEBrick Web Server, which comes with Ruby. Most sites, however, use Apache or lightTPD in production.

Rails works with many database systems, including MySQL, PostgreSQL, SQLite, Oracle, DB2 and SQL Server. Please refer to a corresponding Database System Setup manual to setup your database.

As you know, Ruby on Rails is cross-platform, meaning that you can run it on both Linux, Mac OS or Windows. Personally, I prefer Linux/Unix-based environments such as Linux distros (Ubuntu, Linux Mint...) or Mac OS. However, there are two reasons for writing this guideline assuming that you're using **Windows**:

- There is a larger number of students who are using Windows (7, 8, 8.1, even 10 DP)
- Installing Ruby on Rails in Mac OS and Linux are much easier, you can find many tutorials online

Let's look at the sample installation instructions for Rails on Windows.

Rails Installation on Windows

1. First, let's check to see if you already have Ruby installed. Bring up a command prompt and type ruby -v. If Ruby responds, and if it shows a version number at or above 2.1.3 then type gem --version. If you don't get an error, skip to step 3. Otherwise, we'll install a fresh Ruby.
2. If Ruby is not installed, then download an installation package from RailsFTW page. Follow the **download** link, and run the resulting installer. This is an exe like **rails-ftw-v0.18-2.1.5-4.1.8.exe** and will be installed in a single click. You may as well install everything. It's a very small package, and you'll get RubyGems as well along with this package.
3. Ruby Development Kit is required by some bundles, so we need to install it. First, you can download the package from: DevKit download page. Please note that we will get the ".exe" package, something like "DevKit-tdm-32-4.5.2-20111229-1559-sfx.exe — DevKit-4.5.2 self-extracting archive".

   Extract DevKit to a folder (like C:\DevKit")

   Open Command prompt

   Go to DevKit folder, such as C:\DevKit by typing:

   ```
   > cd C:\DevKit
   ```

   Initialize DevKit:

```
> ruby dk.rb init
```

Install DevKit:

```
> ruby dk.rb install
```

And voila! You're done!

Do not remove DevKit folder

Congratulations! You are now on Rails over Windows.

4. Assuming you installed Rails using RubyGems, keeping up-to-date is relatively easy. Issue the following command:

```
> gem update --system
```

This will automatically update your Rails installation. The next time you restart, your application it will pick up this latest version of Rails. While giving this command, make sure you are connected to the Internet.

IV. Quick start

1. Create New Project

Simply open Command Prompt and type:

```
> rails new fpaintings
```

By default, Rails uses SQLite as database management system and bundle dependencies right after. We can use this command to have more customizable options (it will stop bundle and select MySQL as DBMS instead of default configuration):

```
> rails new fpaintings -d mysql --skip-bundle
```

2. Modify Gemfile

The standard Gemfile should be as below:

```
source 'https://rubygems.org'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '4.1.8'
# Use mysql as the database for Active Record
gem 'mysql2'
# Use SCSS for stylesheets
gem 'sass-rails', '~> 4.0.3'
# Use Uglifier as compressor for JavaScript assets
```

```
gem 'uglifier', '>= 1.3.0'
# Use CoffeeScript for .js.coffee assets and views
gem 'coffee-rails', '~> 4.0.0'
# See https://github.com/sstephenson/execjs#readme for more supported runtimes
# gem 'therubyracer',  platforms: :ruby

# Use jquery as the JavaScript library
gem 'jquery-rails'
# Turbolinks makes following links in your web application faster. Read more:
# https://github.com/rails/turbolinks
gem 'turbolinks'
# Build JSON APIs with ease. Read more: https://github.com/rails/jbuilder
gem 'jbuilder', '~> 2.0'
# bundle exec rake doc:rails generates the API under doc/api.
gem 'sdoc', '~> 0.4.0',          group: :doc

# Use ActiveModel has_secure_password
gem 'bcrypt', '~> 3.1.7'
gem 'activeadmin', github: 'activeadmin'
gem 'devise'
gem 'carrierwave'
gem 'ckeditor', github: 'galetahub/ckeditor'
gem 'paperclip'
gem 'will_paginate'

# Windows does not include zoneinfo files, so bundle the tzinfo-data gem
gem 'tzinfo-data', platforms: [:mingw, :mswin]
```

3. Run bundle install

```
> bundle install
```

4. Configure database

As we are going to use MySQL instead of SQLite, we need to reconfigure this parameter.

Go to **config/database.yml** , replace the content of this file with the following code:

```
default: &default
  adapter: mysql2
  encoding: utf8
  pool: 5
  username: root
  password: [password]
  host: localhost
  port: 3306
```

```
development:
  <<: *default
  database: fpainting

test:
  <<: *default
  database: fpainting

production:
  <<: *default
  database: fpainting
```

After you create the application, switch to its folder:

```
> cd fpaintings
```

The fpaintings directory has a number of auto-generated files and folders that make up the structure of a Rails application. Most of the work in this project will happen in the app folder, but here's a basic rundown on the function of each of the files and folders that Rails created by default:

File/Folder	Purpose
app/	Contains the controllers, models, views, helpers, mailers and assets for your application. You'll focus on this folder for the remainder of this guide.
bin/	Contains the rails script that starts your app and can contain other scripts you use to setup, deploy or run your application.
config/	Configure your application's routes, database, and more.
config.ru	Rack configuration for Rack based servers used to start the application.
db/	Contains your current database schema, as well as the database migrations.
Gemfile Gemfile.lock	These files allow you to specify what gem dependencies are needed for your Rails application. These files are used by the Bundler gem.
lib/	Extended modules for your application.
log/	Application log files.

public/	The only folder seen by the world as-is. Contains static files and compiled assets.
Rakefile	This file locates and loads tasks that can be run from the command line. The task definitions are defined throughout the components of Rails. Rather than changing Rakefile, you should add your own tasks by adding files to the lib/tasks directory of your application.
README.rdoc	This is a brief instruction manual for your application. You should edit this file to tell others what your application does, how to set it up, and so on.
test/	Unit tests, fixtures, and other test apparatus.
tmp/	Temporary files (like cache, pid, and session files).
vendor/	A place for all third-party code. In a typical Rails application this includes vendored gems.

V. Getting Up and Running

1. Rails Controller - ActionController

The Rails controller is the logical center of your application. It coordinates the interaction between the user, the views, and the model. The controller is also a home to a number of important ancillary services.

- It is responsible for routing external requests to internal actions. It handles people-friendly URLs extremely well.
- It manages caching, which can give applications orders-of-magnitude performance boosts.
- It manages helper modules, which extend the capabilities of the view templates without bulking up their code.
- It manages sessions, giving users the impression of an ongoing interaction with our applications.

The process for creating a controller is very easy. We will create 4 controllers here:

```
> rails generate controller categories
```

```
> rails g controller artists
> rails g controller paintings
> rails g controller subpages
```

This command accomplishes several tasks, of which the following are relevant here:

★ It creates a file called **app/controllers/categories_controller.rb**

If you have look at **categories_controller.rb**, you will find it as follows:

```
class CategoriesController < ApplicationController
end
```

Controller classes inherit from *ApplicationController,* which is the other file in the controllers folder: **application.rb**.

The *ApplicationController* contains code that can be run in all your controllers and it inherits from Rails *ActionController::Base* class.

You don't need to worry with the *ApplicationController* as of yet, so just let's define few method stubs in **categories_controller.rb**. Based on your requirement, you could define any number of functions in this file.

Modify the file to look like the following and save your changes. Note that its up to you what name you want to give to these methods, but better to give relevant names.

❖ **Implementing the Listing method**

The index method gives you a printout of all the categories in the database. The *@categories = Category.all* line in the index method tells Rails to search the books table and store each row it finds in the *@categories* instance object.

```
class CategoriesController < ApplicationController

  def index
      @categories = Category.all
  end
```

❖ **Implementing the New method**

The new method lets Rails know that you will create a new object.

```
 def new
      @category = Category.new
  end
```

❖ Implementing the Create method

Once you take user input using HTML form, its time to create a record into the database. To achieve this, edit the create method in the **categories_controller.rb** to match the following:

```
def create
    @category = Category.new(category_params)

    if @category.save
    redirect_to @category
    else
    render 'new'
    end
end
```

The first line creates a new instance variable called *@category* that holds a *Category* object built from the data the user submitted. The data was passed from the **new** method to create using the params object.

The next line is a conditional statement that redirects the user to the **index** method if the object saves correctly to the database. If it doesn't save, the user is sent back to the new method. The ***redirect_to*** method is similar to performing a meta refresh on a web page: it automatically forwards you to your destination without any user interaction.

❖ Implementing the Update method

This method will be called after the edit method when user modifies a data and wants to update the changes into the database. The update method is similar to the create method and will be used to update existing categories in the database.

```
def update
    @category = Category.find(params[:id])

    if @category.update(category_params)
    redirect_to @category
    else
    render 'edit'
    end
end
```

❖ Implementing the Delete method

```
def destroy
```

```
      @category = Category.find(params[:id])
      @category.destroy

      redirect_to categories_path
  end
```

❖ **Defining strong parameters**

Strong parameters require us to tell Rails exactly which parameters are allowed into our controller actions. We have to whitelist our controller parameters to prevent wrongful mass assignment. In this case, we want to both allow and require the *nametype* parameter for valid use of create. The syntax for this introduces require and permit. The change will involve one line in the create action:

```
 private
  def category_params
      params.require(:category).permit(:nametype)
  end
end
```

We do the same as other controllers. Below are contents inside **paintings_controller.rb**

```
class PaintingsController < ApplicationController
  def index
      @paintings = Painting.all
  end
  def new
      @painting = Painting.new
      @categories = Category.all
  end

  def edit
      @painting = Painting.find(params[:id])
      @categories = Category.all
  end

  def show
      @painting = Painting.find(params[:id])
  end

  def create
      @painting = Painting.new(painting_params)

      if @painting.save
      redirect_to @painting
```

```
        else
        @categories = Category.all
        render 'new'
        end
    end

    def update
        @painting = Painting.find(params[:id])
        if @painting.update(painting_params)
        redirect_to @painting
        else
        @categories = Category.all
        render 'edit'
        end
    end
    def destroy
        @painting = Painting.find(params[:id])
        @painting.destroy

        redirect_to paintings_path
    end

    def show_categories
        @category = Category.find(params[:id])
    end

    private
    def painting_params
        params.require(:painting).permit(:title,   :artist_id,   :body,   :year,
:material, :location, :image, :category_id)
    end
end
```

2. Configuring routes

Rails provides a resources method which can be used to declare a standard REST resource. You need to add the *category resource* to the **config/routes.rb** as follows:

```
Rails.application.routes.draw do
  resources :categories
end
```

We set the index page of painting as default or root by adding root '`paintings#index`' to this file. Finally, your **routes.rb** configuration will look like as follows:

```
Rails.application.routes.draw do
```

```
  root 'paintings#index'
  resources :paintings
  resources :artists
  resources :categories
  resources :subpages
end
```

The route, controller, action and view are now working harmoniously! It's time to create the form for a new article.

3. The first form

To create a form within this template, you will use a *form builder*. The primary form builder for Rails is provided by a helper method called *form_for*. But, for convenience, we use this method together with **partial template** by create a new file in **app/views/categories/_form.html.erb** and add this code into this **_form.html.erb**:

```
<%= form_for @category do |f| %>

      <% if @category.errors.any? %>
      <div id="error_explanation">
      <h2>
            <%= pluralize(@category.errors.count, "error") %> prohibited
            this category from being saved:
      </h2>
      <ul>
            <% @category.errors.full_messages.each do |msg| %>
            <li><%= msg %></li>
            <% end %>
      </ul>
      </div>
      <% end %>
      <p>
      <%= f.label :nametype %><br>
      <%= f.text_field :nametype %>
      </p>
      <p>
      <%= f.submit %>
      </p>
<% end %>
```

To render a partial as part of a view, you use the render method within the view:

`<% = render “form” %>` or `<%= render “categories/form” %>`

4. Rails Active Records - Models

Rails Active Record is the Object/Relational Mapping (ORM) layer supplied with Rails. It closely follows the standard ORM model, which is as follows:

- tables map to classes,
- rows map to objects and
- columns map to object attributes

Rails Active Records provides an interface and binding between the tables in a relational database and the Ruby program code that manipulates database records. Ruby method names are automatically generated from the field names of database tables.

Each Active Record object has CRUD (**C**reate, **R**ead, **U**pdate, and **D**elete) methods for database access. This strategy allows simple designs and straightforward mappings between database tables and application objects.

Creating Active Record files

Models in Rails use a singular name, and their corresponding database tables use a plural name. To create the Active Record files for our entities for fpainting application, issue the following command from the top level of the application directory.

```
> rails generate model Category nametype:string
> rails g model Artist name:string born:string died:string nationality:string field:string intro:text
> rails g model Painting title:string artist:references body:text year:string material:string location:string image:string category:references
> rails g model Subpage pagename:string text:text
```

You're telling the generator to create models called Category, Artist, Painting and Subpage to store instances of categories, artists, subpages and paintings. Notice that you are capitalizing Category, Artist, Painting as well as Subpage and using the singular form. This is a Rails paradigm that you should follow each time you create a model.

With the first command, we told Rails that we want a Category model, together with a *nametype* attribute of type string. Those attributes of each model are automatically added to the corresponding tables in the database and mapped to its model.

Run bundle install

```
> bundle install
```

VI. Associating Models and Application Administration

1. Associating Models

When you have more than one model in your rails application, you would need to create connection between those models. You can do this via associations. Active Record supports three types of associations:

- **one-to-one :** A one-to-one relationship exists when one item has exactly one of another item. For example, a person has exactly one birthday or a dog has exactly one owner.
- **one-to-many :** A one-to-many relationship exists when a single object can be a member of many other objects. For instance, one subject can have many books.
- **many-to-many :** A many-to-many relationship exists when the first object is related to one or more of a second object, and the second object is related to one or many of the first object.

You indicate these associations by adding declarations to your models: *has_one*, *has_many*, *belongs_to*, and *has_and_belongs_to_many*.

So now you need to tell Rails what relationships you want to establish within the fpainting data system. To do so, modify **artist.rb**, **category.rb** and **painting.rb** to look like this:

```
class Artist < ActiveRecord::Base
  has_many :paintings
end
class Category < ActiveRecord::Base
  has_many :paintings
end
```

Notice here we have used plural *paintings*, because one *artist* or *category* can have multiple *paintings*.

```
class Painting < ActiveRecord::Base
  belongs_to :artist
  belongs_to :category
end
```

Notice here we have used singular *artist* and *category*, because one *painting* can belong to one *artist* and *category*.

2. Active Admin

Application administration is a common requirement in most web applications and building one from scratch can be a daunting task. There are, however, some options that can save you from starting from nothing when creating your admin interface. We'll be looking at one of the popular options available – Active Admin.

Active Admin is a framework for building administration style interfaces. With little effort, you can create an admin interface that enables you to manage your data and it is highly customizable. We will be looking at how to set up and customize it in a Rails 4 application.

As you can see, Active Admin has already been installed in your application.

```
gem 'activeadmin', github: 'activeadmin'
gem 'devise'
```

Run the generator to install Active Admin. This will create an AdminUser model, an initializer file for configuring Active Admin and an **app/admin** directory that will hold the administration files. It uses Devise for authentication.

```
> rails g active_admin:install
```

You will get further setup instructions on the terminal for some settings you need to do manually.

Next, run the migration.

```
> rake db:migrate
```

Start the server and navigate to http://localhost:3000/admin. You should be able to login using the following:

Username: admin@example.com

Password: password

After logging in, the admin dashboard is presented. At the top is a menu showing the models that have been registered with Active Admin. At this point, only the AdminUser model has been registered. You can view a list of registered admin users, edit their information, and create new ones.

Configuring Active Admin

Register our models

```
> rails generate active_admin:resource Category
> rails g active_admin:resource Artist
> rails g active_admin:resource Painting
> rails g active_admin:resource Subpage
```

Add *permit_params* to **app/admin/"resource"**. For example, Painting will have: `permit_params :title, :artist_id, :body, :year, :material, :location, :image, :category_id`

First, change the columns that are displayed. Active Admin displays columns for all fields that your object has, but in this case we will remove the **Created At**, **Updated At** and **Intro** columns. This is done within the index method in **app/admin/artist.rb**, where the included columns are specified.

```
index do
      column :name
      column :born
      column :died
      column :nationality
      column :field
      actions
  end
```

Active Admin will detect the *belongs_to* relationship that Painting has with Category and Artist, but you will notice that for the Artist column, the name of the entry is displayed as *Artist #1* (depending on the id of the record). Looking at the Artist dropdown menu that is on the Filters sidebar on the right, notice that the Artist object name is what is displayed and not something that is human-readable.

To fix this, define a ***to_s*** method for the Artist model in the **app/models/artist.rb** file.

```
class Artist < ActiveRecord::Base
  has_many :paintings
  def to_s
       "#{name}"
  end
end
```

On refreshing the page, the Artist column will now display the artist's full name.

VII. Integrating CKEditor

CKEditor is a ready-for-use HTML text editor designed to simplify web content creation. It's a **WYSIWYG** editor that brings common word processor features directly to your web pages.

First, we need to add the CKEditor gem to our gemfile.

```
gem 'ckeditor', github: 'galetahub/ckeditor'
```

Now, let's add the CKEditor javascript include to our *application.js*. Modify your *application.js* file so that it looks like the code listed below.

```
// This is a manifest file that'll be compiled into application.js, which will include
// all the files listed below.
//
// Any JavaScript/Coffee file within this directory, lib/assets/javascripts,
// vendor/assets/javascripts, or vendor/assets/javascripts of plugins, if any, can be
// referenced here using a relative path.
//
// It's not advisable to add code directly here, but if you do, it'll appear at the
// bottom of the compiled file.
//
// Read Sprockets README (https://github.com/sstephenson/sprockets#sprockets-directives)
// for details about supported directives.
//
//= require jquery
//= require jquery_ujs
//= require turbolinks
//= require ckeditor/init
//= require bootstrap.min
//= require_tree .
```

Replace `<%= f.text_field %>` by `<%= f.cktext_area %>`. For example, `<%= f.cktext_area :body %>` instead of `<%= f.text_field :body %>` in **app/views/paintings/_form.html.erb**

To integrate CKEditor to Active Admin, we need to add a form to file in **app/admin** folder. Let us take **artist.rb** as an example:

```
form :html => {:multipart => true} do |f|
      f.inputs do
      f.input :name
      f.input :born
      f.input :died
      f.input :nationality
      f.input :field
      f.input :intro, :as => :ckeditor
      end

      f.actions
  end
```

Simply `add :as => :ckeditor` at the end of the line that we want to integrate.

For the views, we also need to add `.html_safe` into the corresponding object.

`<%= @artist.intro.html_safe %>` rather than `<%= @artist.intro %>` as normal.

Note: We've also found it necessary to add to **app/assets/stylesheets/active_admin.css.scss** to fit it to on the form:

```
.cke_chrome {
  width: 79.5% !important;
  overflow: hidden;
}
```

VIII. Uploading image using Carrierwave

```
gem 'carrierwave'
```

Start off by generating an uploader:

```
> rails generate uploader Image
```

this should give you a file in **app/uploaders/image_uploader.rb**

Check out this file for some hints on how you can customize your uploader. It should look something like this:

```
class ImageUploader < CarrierWave::Uploader::Base

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

Open your model file and mount the uploader:

```
mount_uploader :image, ImageUploader
```

Modify *form:html* in **app/admin/*.rb** to ***:as => :file:***

```
f.input :image, :as => :file
```

In form, change `<%= f.text_field :image %>` to `<%= f.file_field :image %>`

In view, modify to image tag for show off `<%= image_tag (@painting.image) %>`

IX. Adding some validation

Rails includes methods to help you validate the data that you send to models. Rails can validate a variety of conditions in a model, including the presence or uniqueness of columns, their format, and the existence of associated objects.

1. Numeric validity

(in artist model)

```
validates :born, :died, numericality: { only_integer: true, greater_than: 1300, less_than_or_equal_to: 2015 }
```

2. Validate presence and uniqueness

(in artist, painting, subpage and category model)

```
validates :title, :artist_id, :body, :image, :category_id, presence: true
validates_uniqueness_of :title
validates_presence_of :nametype
validates :name, presence: true, length: { minimum: 5 }
validates_uniqueness_of :name
validates_presence_of :nationality, :intro
validates_presence_of :pagename, :text
```

X. More tiny customizations

1. Web title

Set up in **application.html.erb**

```
<title><%= yield(:title) %> | Fpainting</title>
```

In **app/views/artists/index.html.erb**

```
<% provide(:title, "Họa sĩ nổi tiếng")%>
<!DOCTYPE html>
<html>
<head>
  <title><%= yield(:title) %> | Fpainting</title>
</head>
```

In **app/views/paintings/show.html.erb**

```
<% provide(:title, @painting.title)%>
<!DOCTYPE html>
<html>
<head>
  <title><%= yield(:title) %> | Fpainting</title>
</head>
```

2. HTTP Basic Authentication

HTTP basic authentication is an authentication scheme that is supported by the majority of browsers and other HTTP clients. As an example, consider an administration section which will only be available by entering a username and a password into the browser's HTTP basic dialog window. Using the built-in authentication is quite easy and only requires you to use one method, *authenticate_or_request_with_http_basic*.

Let us start with our Fpainting project. We do not have much to do to implement authentication. I'm going to add few lines in blue in our **~fpaintings/app/controllers/artists_controller.rb:**

Finally, your **artists_controller.rb** file will look like as follows:

```
class ArtistsController < ApplicationController
  USER_ID, PASSWORD = "admin", "123456"

  # Require authentication for new, edit and delete operation
  before_filter :authenticate, :only => [ :new, :create, :update, :edit, :destroy ]

  def index
      @artists = Artist.order(:born)
  end

  def new
      @artist = Artist.new
  end

  def edit
      @artist = Artist.find(params[:id])
```

```
  end

  def show
      @artist = Artist.find(params[:id])
  end

  def create
      @artist = Artist.new(artist_params)

      if @artist.save
      redirect_to @artist
      else
      render 'new'
      end
  end

  def update
      @artist = Artist.find(params[:id])

      if @artist.update(artist_params)
      redirect_to @artist
      else
      render 'edit'
      end
  end

  def destroy
      @artist = Artist.find(params[:id])
      @artist.destroy

      redirect_to artists_path
  end

  private
  def artist_params
      params.require(:artist).permit(:name,  :born,  :died,  :nationality,  :field,
:intro)
  end
  private
  def authenticate
      authenticate_or_request_with_http_basic do |id, password|
      id == USER_ID && password == PASSWORD
      end
  end
end
```

Let me explain these new lines:

- First line is just to define user ID and password to access various pages.
- Second line, I have put *before_filter* which is used to run the configured method *authenticate* before any action in the controller. A filter may be limited to specific actions by declaring the actions to include or exclude. Both options accept single actions (:only => :index) or arrays of actions (:except => [:foo, :bar]). So here we have put authentication for new, edit and delete operations only.
- Because of second line, whenever you would try to edit or delete an artist record, it will execute private *authenticate* method.
- A private *authenticate* method is calling *authenticate_or_request_with_http_basic* method which comprises of a block and displays a dialogue box to ask for User ID and Password to proceed. If you enter a correct user ID and password then it will proceed otherwise it would display access denied.

Now try to add new, edit or delete any available record, to do so you would have to go through authentication process using credentials:

Username: admin

Password: 123456

Note: Operations using ActiveAdmin are always bypassed and can't be affected by the HTTP Basic Authentication. This method is designed to protect data against random or intentional manipulation, loss, destruction or access by unauthorised persons.

3. Pagination

```
gem 'will_paginate'
def index
      @paintings = Painting.paginate(:page => params[:page], :per_page => 6)
  end
## render page links in the view:
<%= will_paginate @paintings %>
```

Note: ActiveAdmin uses Kaminari for pagination and if you use *will_paginate* in your app, you need to configure an initializer for Kaminari to avoid conflicts. Put this in **config/initializers/kaminari.rb**

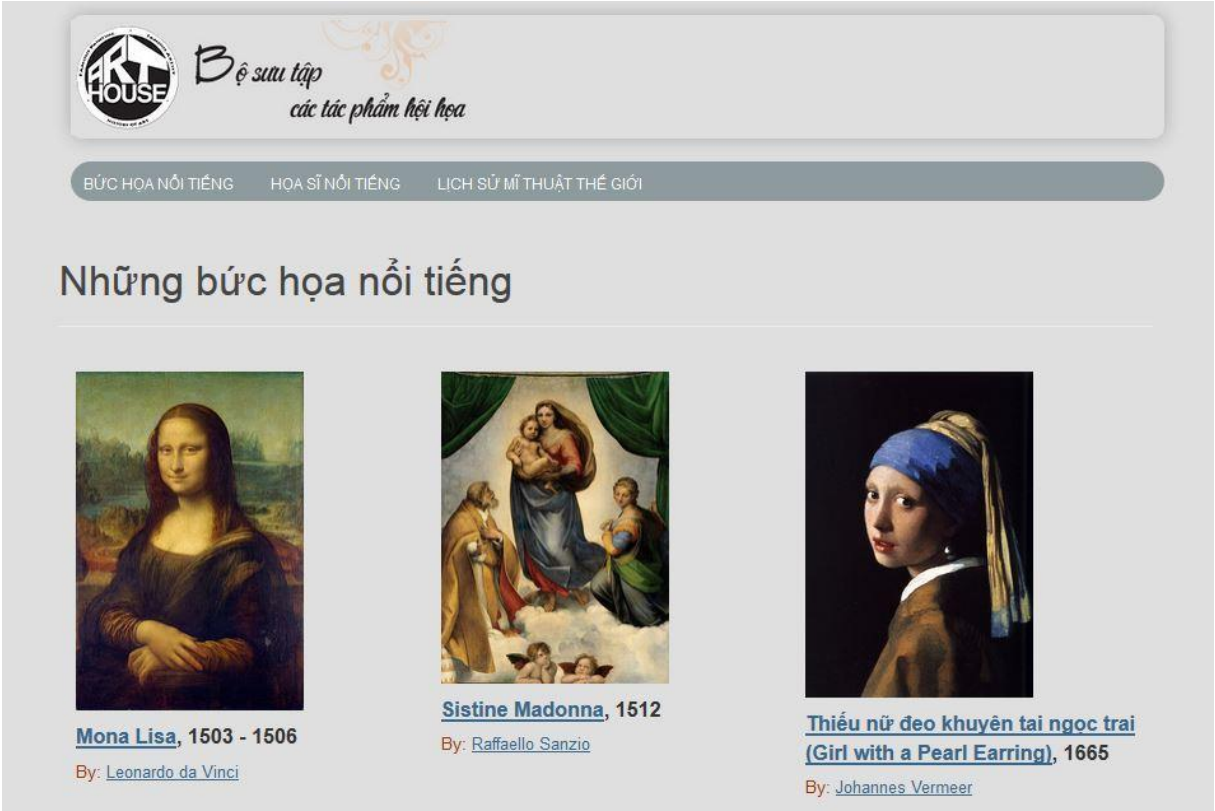
```
Kaminari.configure do |config|
  config.page_method_name = :per_page_kaminari
end
```

XI. Web content

Without original and desirable content, or consideration for the rights and commercial interests of content creators - any media venture is likely to fail through lack of appealing content, regardless of other design factors. Therefore, we concentrate on content creation to produce high-quality articles that contributes to our helpful, information-rich site. All materials are collected from various sources, translated into Vietnamese, posted and structured clearly on the site so that anyone can read and understand thoroughly.

XII. Screenshots

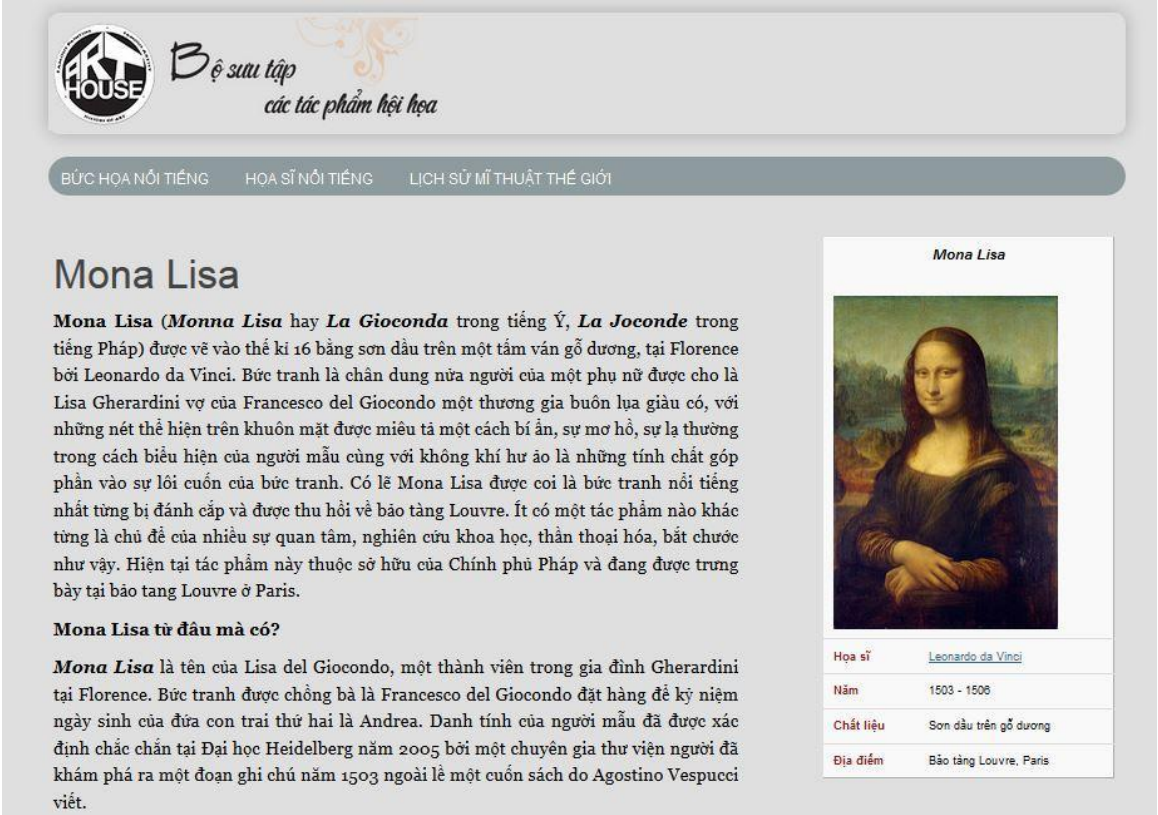
Below are some screenshots taken from our website.



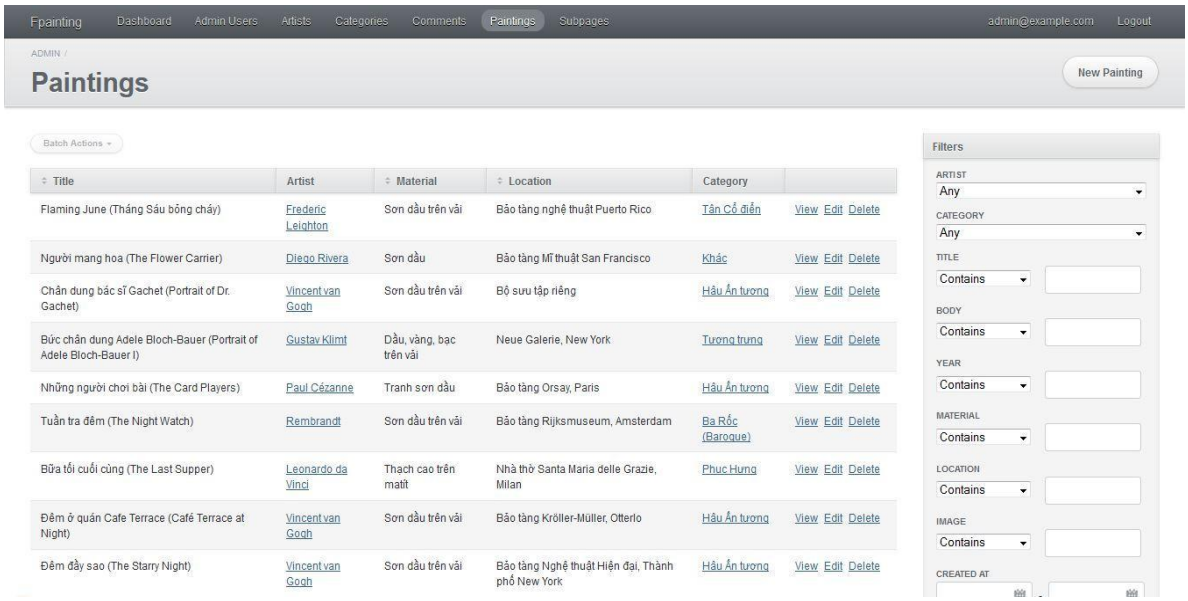
Homepage with simple layout and main-focused contents

Footer and Pagination

World History of Art Page



A Single page: ***Mona Lisa*** *by* ***Leonardo da Vinci***



ActiveAdmin back-end administration site

XIII. Conclusion

After significant initial frustration and pain, we now feel quite productive in Rails. It allows us to develop a lot of functionality in a small amount of time, but, as with all frameworks, when encountering a problem we'll need to acquire a thorough understanding of the framework in order to proceed. This may negate the development speed advantage, but only for the first project.

Rails has a very active community and there are thousands of tutorials, screencasts and blog posts on common tasks. However, sometimes, we're more dependent on Google and hoping that the blog post we found is still current because anything older than about a year is quite likely obsolete.

In summary, this project helps us in developing website as well as learning a new programming language - Ruby and its web application framework - Ruby on Rails. We all hope that our project will be a start-up for now and be deployed in the future.