Uploading Images

Optional Exercise

Objective

As the application stands, when you create a new movie you have to use an existing image file that's in the `app/assets/images` directory. So if you want to create a new movie on Heroku's server, you need to add a new image to your local Git repo and then deploy the entire app to the server. That's kinda inconvenient!

Instead, when users create (or edit) a movie we'd like them to be able to select a movie image file on their computer and upload it to the server. This turns out to be fairly easy to do using the Paperclip gem. The only gotcha is Heroku doesn't allow us to put files on their read-only filesystem, so we'll need to store the movie image files on an external server. Thankfully, the Paperclip gem has good support for storing uploaded files on Amazon's Simple Storage Service (Amazon S3).

We'll start by using Paperclip to upload and store images on our local machine during development, and then arrange things so that files are stored on Amazon S3 when the app is running in production.

Specs

To check your work as you go along, you'll need to make the following changes to your specs.

Hide Specs

1. Replace the following line in the `create_movie_spec.rb` file:

```
fill_in "Image file name", with: "movie.png"
```

   with the following:

```
attach_file "Image", "#{Rails.root}/app/assets/images/ironman.jpg"
```

2. In the `list_movies_spec.rb` feature spec file, rather than setting the `image_file_name` attribute for each movie like so:

```
image_file_name: "ironman.jpg"
```

   Instead, set the `image` attribute with the contents of the image file, like so:

```
image: open("#{Rails.root}/app/assets/images/ironman.jpg")
```

   Then replace the following expectation:

```
expect(page).to have_selector("img[src$='#
{movie1.image_file_name}']")
```

   with the following:

```
expect(page).to have_selector("img[src$='#{movie1.image.url}']")
```

3. Similarly, in the `show_movie_spec.rb` file, replace the following expectation:

   ```
   expect(page).to have_selector("img[src$='#
   {movie.image_file_name}']")
   ```

   with the following:

   ```
   expect(page).to have_selector("img[src$='#{movie.image.url}']")
   ```

4. Finally, you'll also need to change the `movie_attributes` method in the `spec/supports/attributes.rb` file. Rather than setting the `image_file_name` attribute like this:

   ```
   image_file_name: "ironman.jpg"
   ```

   Instead, set the `image` attribute with the contents of the image file, like so:

   ```
   image: open("#{Rails.root}/app/assets/images/ironman.jpg")
   ```

1. Install Paperclip

First we need to install the Paperclip gem, as well as the AWS SDK gem for storing the movie image files on the Amazon S3 service.

To do that, add the following lines to your `Gemfile` and save it:

```
gem 'paperclip'
gem 'aws-sdk'
```

Then install the gems using:

```
bundle install
```

2. Update the Movie Model

Next we need to tell our `Movie` model that it can have an attached image and add validations to set limitations on the uploaded file size, content type, and so on. We'll also need to add extra fields to the `movies` database table to store metadata about the uploaded image.

1. Start by adding the following declaration to your `movie.rb` file:

   ```
   has_attached_file :image
   ```

   The `has_attached_file` method is included in the Paperclip library. We give it the name of the attachment, which is `image` in this case. This declaration tells the model that it can have an attachment called `image` with metadata stored in database fields that have an `image_` prefix. We'll add those fields shortly.

2. Then remove the existing format validation for the `image_file_name` attribute, and instead add the following validations:

   ```
   validates_attachment :image,
   ```

```
  :content_type => { :content_type => ['image/jpeg', 'image/png']
},
  :size => { :less_than => 1.megabyte }
```

This ensures that the actual *content type* of the image file, regardless of the file name extension, is a JPEG or PNG file. (We'll leave out GIF files this time.) It's also always a good idea to limit the size of uploaded files to something reasonable, in this case 1 MB.

3. Next we need to add extra fields to the `movies` database table to store metadata about the uploaded image. Start by generating a new migration file:

```
rails g migration AddExtraImageFieldsToMovies
```

4. Since we named our attachment `image`, the corresponding database field names will have an `image_` prefix. Here are the database fields that Paperclip will need to store information about movie images:

name	type
image_file_name	string
image_content_type	string
image_file_size	integer
image_updated_at	datetime

By default, only an `image_file_name` field is required, and conveniently we already have that field! However, since we have validations for the file content type and size, we need fields for both of those attributes. It's also handy to have a field to track when the image was updated.

In the generated migration file, update the `change` method to add the three new fields like so:

```
class AddExtraImageFieldsToMovies < ActiveRecord::Migration
  def change
    add_column :movies, :image_content_type, :string
    add_column :movies, :image_file_size, :integer
    add_column :movies, :image_updated_at, :datetime
  end
end
```

5. Finally, apply the new migration:

```
rake db:migrate
```

3. Update the Form

The `Movie` model can now accept a file attachment, but next we need to change the form so that users can actually upload an image file when creating or editing a movie.

1. In the `app/views/movies/_form.html.erb` partial, start by removing the following lines that generate a text field for entering the `image_file_name`:

```
<li>
  <%= f.label :image_file_name %>
  <%= f.text_field :image_file_name %>
</li>
```

2. Instead, at the bottom of the form right before the submit button, add a label and file field for selecting a file to upload, like so:

```
<li>
  <%= f.label :image %>
  <%= f.file_field :image %>
</li>
```

3. Before you can submit a file to upload, back in your `MoviesController` you'll need to add the `image` attribute to the list of permitted parameters:

```
def movie_params
  params.require(:movie).permit(:title, :description, :rating,
:released_on, :total_gross, :cast, :director, :duration, :image)
end
```

4. Create a New Movie

Now go to the form for creating a new movie and you should see a "Choose File" button near the bottom of the form. Click it and select an image file you want to attach to the movie. For example, you might want to drag the Iron Man 3 poster image to your desktop and select it.

Make sure to fill out the rest of the form with "Iron Man 3" details, and then click "Create Movie". You should get redirected back to the movie's show page with a flash message saying it was successfully created.

There's just one problem: The movie image doesn't show up!

5. Check in the Console

To check that an image was indeed attached to the movie, let's jump into a console and see what's going on:

1. Start by finding the movie you just updated:

```
>> m = Movie.find_by(title: "Iron Man 3")
```

2. Check that the `image_file_name` attribute is the name of the file you uploaded:

```
>> m.image_file_name
=> "Iron_Man_3_theatrical_poster.jpg"
```

3. While you're at it, go ahead and check the other `image_` attributes:

```
>> m.image_content_type
=> "image/jpeg"

>> m.image_file_size
=> 28095

>> m.image_updated_at
=> Fri, 26 Apr 2013 20:49:57 UTC +00:00
```

4. OK, so the database fields contain metadata about the image, but where was the actual image file stored? To answer that, use the `image` attribute to get the path of the image file, like so:

```
>> m.image.path
=>
"/Users/mike/rails_studio/flix/public/system/movies/images/000/000/0
```

Notice that by default Paperclip stores uploaded files beneath the `public` directory. If you dig down into that directory you should find the image file.

5. So how do we get a URL to the uploaded image so that we can display it in the browser? To get the URL, use

```
>> m.image.url
=>
"/system/movies/images/000/000/007/original/Iron_Man_3_theatrical_po
1367243129"
```

This makes sense. Any file in the `public` directory is served directly by the web server. For example, if you take that URL and tack it on to the end of `http://localhost:3000`, then you should see the image in your browser.

Here's the takeaway: Up to this point we've been storing our movie images in the `app/assets/images` directory. However, by default Paperclip stores uploaded images in subdirectories of the `public` directory. So in order to display the uploaded images we have to make some adjustments...

6. Show the Image

If you look in the `show` and `index` templates, you'll recall that we're using our custom `image_for` helper to display the movie image. Here's how that helper is currently implemented:

```
def image_for(movie)
  if movie.image_file_name.blank?
    image_tag('placeholder.png')
  else
    image_tag(movie.image_file_name)
  end
end
```

Notice that it's simply referencing the `image_file_name` attribute and expecting a file with that name to be in the `app/assets/images` directory. We need to change the helper to use `movie.image.url` so that it references the image file that's down in the `public` directory.

1. Change the `image_for` helper like so:

```
def image_for(movie)
  if movie.image.exists?
    image_tag(movie.image.url)
  else
    image_tag('placeholder.png')
  end
end
```

Note that we've reversed the `if` statement from it's original form. We start by checking whether the movie image file exists by using the `exists?` method provided by Paperclip. If the movie has an attached image, the helper returns an image tag based on the image's URL. Otherwise, if the movie doesn't have an attached image, we fall back to displaying the placeholder image that's in the `app/assets/images` directory.

2. Reload the show page for the new "Iron Man 3" movie that has the attached image and you should see the image file displayed!

3. Now try editing that movie. Notice that the form displays "No file chosen" next to the "Choose File" button. Hey, that's confusing. Clearly the movie has an attached image, but the form isn't showing that image. So let's fix that!

In the `app/views/movies/_form.html.erb` partial, use the `image_for` helper to display the movie image (or the placeholder image) right below the file upload field.

```
<li>
  <%= f.label :image %>
  <%= f.file_field :image %>
</li>
<li>
  <%= image_for(@movie) %>
</li>
```

4. Reload the form and you should see the poster image for "Iron Man 3". Now there's no confusion!

5. Finally, the images for the other movies aren't being shown because we haven't uploaded their poster images. Go ahead and do that before moving on.

7. Store Files in Amazon S3 For Heroku

Uploading files to the `public` directory works fine on your development machine, but it won't work on the production Heroku servers. The Heroku filesystem is read-only which means that saving uploaded images to the `public` directory isn't supported on Heroku.

Instead, we'll configure Paperclip to store uploaded files to Amazon's Simple Storage Service (Amazon S3) when the application is running in the production environment.

1. First, sign up for an Amazon S3 account. The free tier offers sufficient storage space for our educational purposes.

2. All files in S3 are stored in *buckets* which act basically like directories. Create a bucket for your uploaded files. You can name the bucket anything you want, but the name must be unique across the entire Amazon S3 system. To avoid naming conflicts, you might want to use the name of the app that Heroku generated as the bucket name. For example, we named our bucket `mighty-thicket`.

3. Next, in order for our application running on Heroku to use S3, we need to give Heroku access to our Amazon S3 account and the name of the bucket to use. Access to S3 is governed by a set of credentials: an access key id and a secret access key. The access key identifies your S3 account and the secret access key should be treated like a password.

   Because of the sensitive nature of the S3 credentials, you never want to store them in a file or add them to your version control system. Instead, on Heroku we can store this sensitive information in application-level configuration variables.

   To set the Amazon S3 configuration variables on your Heroku application, change into your `flix` application directory and use `heroku config:set` to set the following three variables:

   ```
   heroku config:set AWS_BUCKET=your_bucket_name
   heroku config:set AWS_ACCESS_KEY_ID=your_access_key_id
   heroku config:set AWS_SECRET_ACCESS_KEY=your_secret_access_key
   ```

   Your S3 credentials can be found on the Security Credentials section of the AWS "My Account/Console" menu.

   You can review all the Heroku variables you've set using

   ```
   heroku config
   ```

4. Finally, we need to configure Paperclip to use Amazon S3 as the file storage mechanism when the application is running in the production environment. To do that, add the following lines to the `config/environments/production.rb` file *inside* of the block structure:

```
config.paperclip_defaults = {
  :storage => :s3,
  :url => ':s3_domain_url',
  :path => '/:class/:attachment/:id_partition/:style/:filename',
  :s3_credentials => {
    :bucket => ENV['AWS_BUCKET'],
    :access_key_id => ENV['AWS_ACCESS_KEY_ID'],
    :secret_access_key => ENV['AWS_SECRET_ACCESS_KEY']
  }
}
```

See the Paperclip documentation for more information about custom configuration options.

5. Add and commit the changes to your local Git repo:

```
git add .
git commit -m "Configure Paperclip to use S3 in production"
```

6. Then deploy the changes by pushing the code to Heroku just like before:

```
git push heroku master
```

7. You should now be able to upload movie images to your app running on Heroku! The image files will get stored in S3, however all the metadata such as the file's name are still stored in the `movies` database table.

Double Bonus Round!

As the app stands, Paperclip will store a copy of the original file that was uploaded. We have restrictions on the image file size, but what about the image dimensions? Well, Paperclip supports automatic image resizing! The only catch is that Paperclip relies on the external ImageMagick package to do the actual image resizing, and installing ImageMagick can be a bit of a hassle depending on your operating system.

Thankfully, the Heroku servers already have ImageMagick installed. So the only hurdle we need to clear is getting it installed on your development machine.

Install ImageMagick

Mac OS X

The easiest way to install ImageMagick on a Mac is by using the popular Homebrew package manager. Install Homebrew by pasting the following into a Terminal prompt:

```
ruby -e "$(curl -fsSL https://raw.github.com/mxcl/homebrew/go)"
```

Then to install ImageMagick, use:

```
brew install imagemagick
```

Windows

The easiest way to install ImageMagick on Windows is by using the Windows binary release.

Linux

If you're a Linux user, the golden path is to use

```
sudo apt-get install imagemagick
```

Check the Paperclip documentation for more installation options.

Add Resizing Options

Once you have ImageMagick installed, you can define various image *styles*. For example, if you wanted Paperclip to create two resized images in addition to the original upload image, you'd add the following `styles` option to specify the resize dimensions of each file:

```
has_attached_file :image, styles: {
  small: "90x133>",
  thumb: "50x50>"
}
```

Now when you upload an image, Paperclip will automatically create a "small" size where the largest dimension is 90 pixels and a "thumb" size where the largest dimension is 50 pixels. (The "small" size matches the size of the images we used throughout the course.)

By the way, the `>` option tells ImageMagick to proportionally reduce the size of the image. ImageMagick supports a boadload of other resizing options.

Display Resized Images

To get the URL for a specific image size, you simply pass the name of the style as a parameter to the `image.url` method. For example, here's how to display the small image:

```
<%= image_tag @movie.image.url(:small) %>
```

Or if you wanted to display the thumbnail image, use

```
<%= image_tag @movie.image.url(:thumb) %>
```

Since we're using our custom `image_for` helper to display movie images, you'll need to update that method to generate an image tag for the size you prefer:

```
def image_for(movie)
  if movie.image.exists?
    image_tag(movie.image.url(:small))
  else
    image_tag('placeholder.png')
  end
end
```

You can have as many image versions as you like, and they're automatically created and stored when the file is uploaded.

Pretty cool!

Return to Deployment Exercise

Window size: x
Viewport size: x