Extracted from:

Advanced Rails Recipes

Upload Images with Thumbnails

By Rick Olson (http://entp.com)
Rick has been an active contributor to the Rails community for more than three years. He has released numerous open source plug-ins and applications such as Mephisto and Altered Beast. He currently spearheads Rails R&D at entp.com, driving the innovation behind the Lighthouse and Warehouse applications.

Problem

You want to let users upload images (or any media file) as an "attachment" to one of your models. In the case of images, you also want to generate a variety of thumbnails for use around your site.

Ingredients

- The attachment_fu plug-in:

  ```
  $ script/plugin install ↩
      http://svn.techno-weenie.net/projects/plugins/attachment_fu/
  ```

- One of the following image-processing libraries and any libraries on which they depend:
  - *ImageScience*:[21] A lightweight inline-Ruby library that resizes only images. It wraps the FreeImage library, which you'll also need.
  - *RMagick*:[22] The granddaddy, both in terms of advanced image-processing features and memory usage. It wraps the ImageMagick library, which you'll also need.
  - *minimagick*:[23] It's much easier on memory than RMagick because it runs the ImageMagick command in a shell. You'll also need ImageMagick installed.

Image processing is best handled by native code. Regardless of the image processor you choose, you'll end up either building a native

21. http://seattlerb.rubyforge.org/ImageScience.html
22. http://rmagick.rubyforge.org/
23. http://rubyforge.org/projects/mini-magick/

library or downloading a prebuilt library specific to your operating system. Then you generally install a Ruby library (gem) that wraps the image-processing library with a Ruby API. If you already have one of these installed, go with it!

Solution

Suppose we're building an online jukebox and we need to upload cover images for the albums. While we're at it, we'd like to generate a few cover image thumbnails of varying sizes to sprinkle around the site. Here's where the attachment_fu plug-in really shines. Rather than groveling around at the API level of whatever Ruby image library we have installed, we can simply declare how we want files to get processed and let attachment_fu work out the details.

Let's start with what we need in the database. Now, we could try to cram all the album and cover information into one database table. But that gets messy, so instead we'll split them up into two tables. First, we need a database table to store information about the cover image: its size, where it lives, which album it belongs to, and so on. We won't actually store the cover image itself in the database, just its metadata. Here's the migration for the Cover model:

Download UploadImages/db/migrate/002_create_covers.rb

```ruby
class CreateCovers < ActiveRecord::Migration
  def self.up
    create_table :covers do |t|
      t.integer :album_id, :parent_id, :size, :width, :height
      t.string  :content_type, :filename, :thumbnail
      t.timestamps
    end
  end

  def self.down
    drop_table :covers
  end
end
```

The attachment_fu plug-in requires all these columns, with the exception of the album_id column, which is specific to our application. In particular, note that the parent_id column is not a foreign key to an album. Rather, it's a foreign key used by thumbnails to point to their parent cover images in the same covers table. Again, the covers table just stores the information about the cover, not the actual cover image. When an image is uploaded, its location will be stored in the covers table, and the actual file data will be stored somewhere else (we'll get to *where* in a minute).

Next we need a Cover model (no, not that kind!). Here's what it looks like:

Download UploadImages/app/models/cover.rb

```
class Cover < ActiveRecord::Base
  belongs_to :album

  has_attachment :content_type => :image,
                 :storage      => :file_system,
                 :max_size     => 500.kilobytes,
                 :resize_to    => '384x256>',
                 :thumbnails   => {
                   :large =>  '96x96>',
                   :medium => '64x64>',
                   :small =>  '48x48>'
                 }

  validates_as_attachment
end
```

There's a lot of magic happening here. In the has_attachment method, we tell attachment_fu what to do with the uploaded image via a number of options:

- :content_type specifies the content types we allow. In this case, using :image allows all standard image types.
- :storage sets where the actual cover image data is stored. So, in fact, we could have stored the covers in the database (:db_file), but the filesystem is easier to manage.
- :max_size is, not surprisingly, the maximum size allowed. It's always good to set a limit on just how much data you want your app to ingest (the default is 1 megabyte).
- :resize_to is either an array of width/height values (for example, :resize_to => [384, 286]) or a geometry string for resizing the image. Geometry strings are more flexible but not supported by all image processors. In this case, by using the > symbol at the end, we're saying that the image should be resized to 384 by 286 only if the width or height exceeds those dimensions. Otherwise, the image is not resized.
- :processor sets the image processor to use: ImageScience, Rmagick, or MiniMagick. As we haven't specified one, attachment_fu will use

whichever library we have installed.

- :thumbnails is a hash of thumbnail names and resizing options. Thumbnails won't be generated if you leave off this option, and you can generate as many thumbnails as you like simply by adding arbitrary names and sizes to the hash.

After describing how the image should be processed, we call the validates_as_attachment method to prevent image sizes out of range from being saved. (They're still uploaded into memory, mind you.) In addition, because we set an image content type, WinZip files won't be welcome, for example.

Of course, we'll also need an Album model to "attach" a Cover object to, but there's not much to it:

Download UploadImages/app/models/album.rb

```
class Album < ActiveRecord::Base
  has_one :cover, :dependent => :destroy
end
```

OK, with our models created, we turn our attention to the form used to upload the cover image file when we create a new Album:

Download UploadImages/app/views/albums/new.html.erb

```
<%= error_messages_for :album, :cover %>

<% form_for(@album, :html => { :multipart => true }) do |f| %>
  <p>
    <%= label :album, :title %>
    <%= f.text_field :title %>
  </p>
  <p>
    <%= label :album, :artist %>
    <%= f.text_field :artist %>
  </p>
  <p>
    <%= label :album, :cover %>
    <%= file_field_tag :cover_file %>
    <span class="hint">
      We accept JPEG, GIF, or PNG files up to 500 KB.
    </span>
  </p>
  <p>
    <%= f.submit "Create" %>
  </p>
<% end %>
```

It's a fairly standard form, but it has three important ingredients. First, to allow the form to accept files as POST data, the form_for includes the :multipart => true option. (If you forget to add this, you're in for a long afternoon of debugging.)

Second, the form uses the file_field_tag helper (instead of f.file_field) to generate a Choose File button on the form. In this case, the name of the file input field will be :cover_file.

Finally, the error_messages_for method handles the @album and @cover objects so that it displays errors related to both objects.

So far, so good. Next, we need to do something with the cover image that gets uploaded. Specifically, we need to use its file data to create a Cover object and attach it to the Album being created. This gets a bit tricky: we're creating two models from one form. So, to keep the create action of our AlbumsController clean, we're going to introduce a new AlbumService class and let it do the grunt work. Here's the create action:

Download UploadImages/app/controllers/albums_controller.rb

```ruby
def create
  @album = Album.new(params[:album])
  @cover = Cover.new(:uploaded_data => params[:cover_file])

  @service = AlbumService.new(@album, @cover)

  respond_to do |format|
    if @service.save
      flash[:notice] = 'Album was successfully created.'
      format.html { redirect_to(@album) }
      format.xml  { render :xml      => @album,
                           :status   => :created,
                           :location => @album }
    else
      format.html { render :action => :new }
      format.xml  { render :xml    => @album.errors,
                           :status => :unprocessable_entity }
    end
  end
end
```

This populates the album-specific fields—name, artist, and so on—into an Album model. Then it assigns the value of the :cover_file parameter (the file data) to the :uploaded_data attribute of the Cover model. This is a virtual attribute that was added to the Cover model when we declared has_attachment. The create action then creates a new AlbumService with the album and cover and attempts to save it.

All the good stuff happens in the AlbumService model. Here's what it looks like:

Download UploadImages/app/models/album_service.rb

```ruby
class AlbumService

  attr_reader :album, :cover

  def initialize(album, cover)
    @album = album
    @cover = cover
  end

  def save
    return false unless valid?
    begin
      Album.transaction do
        if @cover.new_record?
          @album.cover.destroy if @album.cover
          @cover.album = @album
          @cover.save!
        end
        @album.save!
        true
      end
    rescue
      false
    end
  end

  def valid?
    @album.valid? && @cover.valid?
  end
end
```

This class looks like an Active Record model: it has a save method and a valid? method. However, it doesn't subclass ActiveRecord::Base. It's just a plain ol' Ruby class that manages two Active Record models. You can name these methods however you like. I just find it easier to use conventional names.

The save method needs to save both the album and its cover. Now, attachment_fu hooks into the life cycle of the Cover model to do lots of special processing. For example, the thumbnails are automatically generated after the cover has been saved. Things can go wrong when a cover is being saved, in which case attachment_fu will raise an exception. We handle that by wrapping the saving of both the cover and the album in a transaction block. If an exception is raised in the block, all the database operations are rolled back. That way, we don't end up with one model being saved without the other.

Next, we need to deal with updating an album and potentially its cover image. The form for updating an album looks just like the form for creating one. There's nothing new there. However, the update action of the AlbumsController needs to use the AlbumService, too.

Download UploadImages/app/controllers/albums_controller.rb

```
def update
  @album = Album.find(params[:id])
  @cover = @album.cover

  @service = AlbumService.new(@album, @cover)

  respond_to do |format|
    if @service.update_attributes(params[:album], params[:cover_file])
      flash[:notice] = 'Album was successfully updated.'
      format.html { redirect_to @album }
      format.xml  { head :ok }
    else
      @cover = @service.cover
      format.html { render :action => :edit }
      format.xml  { render :xml    => @album.errors,
                           :status => :unprocessable_entity }
    end
  end
end
```

The update action starts by creating an AlbumService for the album we're editing and its current cover. Then it simply throws the album form parameters, including the :cover_file parameter, into the AlbumService#update_attributes method. Here's what that method looks like:

Download UploadImages/app/models/album_service.rb

```
def update_attributes(album_attributes, cover_file)
  @album.attributes = album_attributes
  unless cover_file.blank?
    @cover = Cover.new(:uploaded_data => cover_file)
  end
  save
end
```

When we're editing an album, we may want to keep its existing cover image by not choosing a new file on the edit form. Then, when update_attributes is called, the cover_file parameter will be blank. In that case, the save method simply saves the album and leaves its current cover intact.

However, we may want to change an album's cover by uploading a new cover image file. In that case, the value of the cover_file parameter will reference the file data when update_attributes is called. Because it's not blank, a new @cover object is created with the file data. Then, when save is called, it'll destroy the album's existing cover (and its thumbnails) and save the new cover (and generate its thumbnails). All this happens within a transaction, just as it does when creating a new album.

OK, now we're off to the races: we select a cover file using the Choose File button on the form, the cover image is uploaded to a file on our server, and the file metadata is stored in the covers database table. We end up with four rows in the covers table: one for the resized original (parent) image and one for each of the three thumbnails. The thumbnails have their parent_id column set to the primary key of the cover from which they were created.

Each image also has a base filename recorded in the covers table. The public_filename method uses this information to give us the public path to the resized original file. Let's inspect our images in the console:

```
$ ruby script/console
>> c = Cover.find :first
=> #<Cover id: 1, album_id: 1, parent_id: nil, size: 72620, width: 201,
    height: 201, content_type: "image/png",
    filename: "foo_fighters.png", thumbnail: nil>
>> c.public_filename
=> "/covers/0000/0001/foo_fighters.png"
```

The public_filename method also takes the name of a thumbnail we used in the :thumbnails hash:

```
>> c.public_filename(:small)
=> "/covers/0000/0001/foo_fighters_small.png"
>> c.public_filename(:medium)
=> "/covers/0000/0001/foo_fighters_medium.png"
>> c.public_filename(:large)
=> "/covers/0000/0001/foo_fighters_large.png"
```

Since we're using the filesystem as storage, our cover image files are stored relative to the RAILS_ROOT/public directory on our server.[24] The thumbnail files have a suffix that corresponds to the name we used in the :thumbnails hash.

Finally, let's write a view helper so we can easily show covers in various sizes (and linked to the full-size image) around our jukebox site:

24. The default path prefix for the filesystem is public/#{table_name}. This can be changed by using the :path_prefix option on the has_attachment method.

Download UploadImages/app/helpers/albums_helper.rb

```ruby
module AlbumsHelper
  def cover_for(album, size = :medium)
    if album.cover
      cover_image = album.cover.public_filename(size)
      link_to image_tag(cover_image), album.cover.public_filename
    else
      image_tag("blank-cover-#{size}.png")
    end
  end
end
```

Then we can use the cover_for helper to list all the albums and their covers:

Download UploadImages/app/views/albums/index.html.erb

```erb
<table>
<% for album in @albums -%>
  <tr>
    <td><%= cover_for(album, :large) %></td>
    <td>
      <strong><%= link_to album.title, album %></strong>
      by <%= h album.artist %>
    </td>
  </tr>
<% end -%>
</table>
```

Now we can create and update an album and its cover image. The creation step was fairly straightforward, but dealing with two models had the added complication of using a transaction. The update step added a bit more degree of difficulty in deleting the old cover images. By introducing an AlbumService class, we were able to encapsulate this complexity in one place and keep the controller clean. If other controllers need to manipulate covers, they can reuse AlbumService to do the heavy lifting.

Discussion

If you want to customize the validations that attachment_fu performs, you can write you own custom validations rather than using the validates_as_attachment convenience method. For example, if you wanted to completely change the error messages, you could remove the call to validates_as_attachment in the Cover model and add the following:

Download UploadImages/app/models/cover.rb

```ruby
validate :attachment_valid?
```

```
def attachment_valid?
  unless self.filename
    errors.add_to_base("No cover image file was selected")
  end

  content_type = attachment_options[:content_type]
  unless content_type.nil? || content_type.include?(self.content_type)
    errors.add_to_base("Cover image content type must an image")
  end

  size = attachment_options[:size]
  unless size.nil? || size.include?(self.size)
    errors.add_to_base("Cover image must be 500-KB or less")
  end
end
```

Also See

Although the attachment_fu plug-in provides support for storing attachments on Amazon's S3 web service, I've found it better to do that in an out-of-band process. See Recipe 42, *Send Lightweight Messages*, on page 212 for how to hook into attachment_fu and upload files to S3 using a queue server.

Snack Recipe 76, *Preserve Files Between Deployments*, on page 361 describes how to keep uploaded images stored on the filesystem from disappearing between deployments.