This tutorial was born out of a need to enable multiple uploads of photos for a site I am working on. The site requires that an admin uploads photos that users can then browse through.

In my searching through the web, I became aware of a certain amount of confusion. Using pure javascript seemed impossible but there is a Flash based multiple upload plugin for jQuery called 'uploadify' http://www.uploadify.com.

(Re the pure javascript issue - I'd love this to be confirmed or otherwise, with a reference).

User authentication issues have been completely ignored - that would be the subject of a different tutorial and this one is going to concentrate on getting uploadify to work together with Paperclip, albeit in a basic form, with Rails.

Paperclip is a Rails gem that makes the uploading of files simple. See http://github.com/thoughtbot/paperclip for details. Using Paperclip means that the database will store details of the files in the database and the actual files in a folder. I will be using default values for Paperclip, which means that the files will be stored in public/system.

For reference, I am using Ruby 1.8.7, Rails 2.3.8 and SQLite Database Browser 1.3 to inspect and clean up the database. Start in the usual way:

```
rails uploader
cd uploader
```

Ensure that the nifty-generators gem is installed

`sudo gem install nifty-generators` (where I use sudo, Windows users should run with administrator privilege) and then do the following:

```
script/generate nifty_layout
script/generate nifty_scaffold user name:string
script/generate nifty_scaffold upload user_id:integer
sudo gem install paperclip
sudo gem install mime-types
script/generate paperclip upload photo
```

(I actually used an app template for the above which is included in the code sample. If you haven't used one, take a look at Ryan Bates Railscast #148 http://railscasts.com/episodes/148-app-templates-in-rails-2-3

I then used uploader_templat.rb, to create the first parts of the uploader project:

`rails -m uploader_template.rb uploader`).

edit the config/database.yml file appropriately and run

```
rake db:migrate
```

add the following to config/environment.rb:

```
config.gem 'mime-types', :lib => 'mime/types'
config.gem 'paperclip'
```

add the following to the user model:

```
has_many :uploads
```

add and amend the upload model to the following:

```
attr_accessible :user_id, :photo
belongs_to :user
has_attached_file :photo
```

edit the show action in apps/controllers/users_controller.rb

```
def show
  @user = User.find(params[:id], :include => :uploads)
end
```

edit the apps/views/users/show.html.erb file to the following:

```
<% title h(@user.name) %>

<h3 id="photos_count"><%= pluralize(@user.uploads.size, "Photo")%></h3>

<div id="uploads">
  <%= render :partial => @user.uploads %>
</div>

<h3>Upload a Photo</h3>
<% form_for Upload.new(:user_id => @user.id), :html => {:multipart => true} do |f| %>
<%= f.hidden_field :user_id, "value" => @user.id %>
    <p>
    <%= f.file_field :photo %>
  </p>
  <p><%= f.submit "Upload" %></p>
<% end %>

<p><%= link_to "All users", users_path %></p>
```

Don't omit the `:html => {:multipart => true}` or nothing will happen!

In uploads_controller, edit the `redirect_to @upload` in the create action as follows:

```
redirect_to @upload.user
```

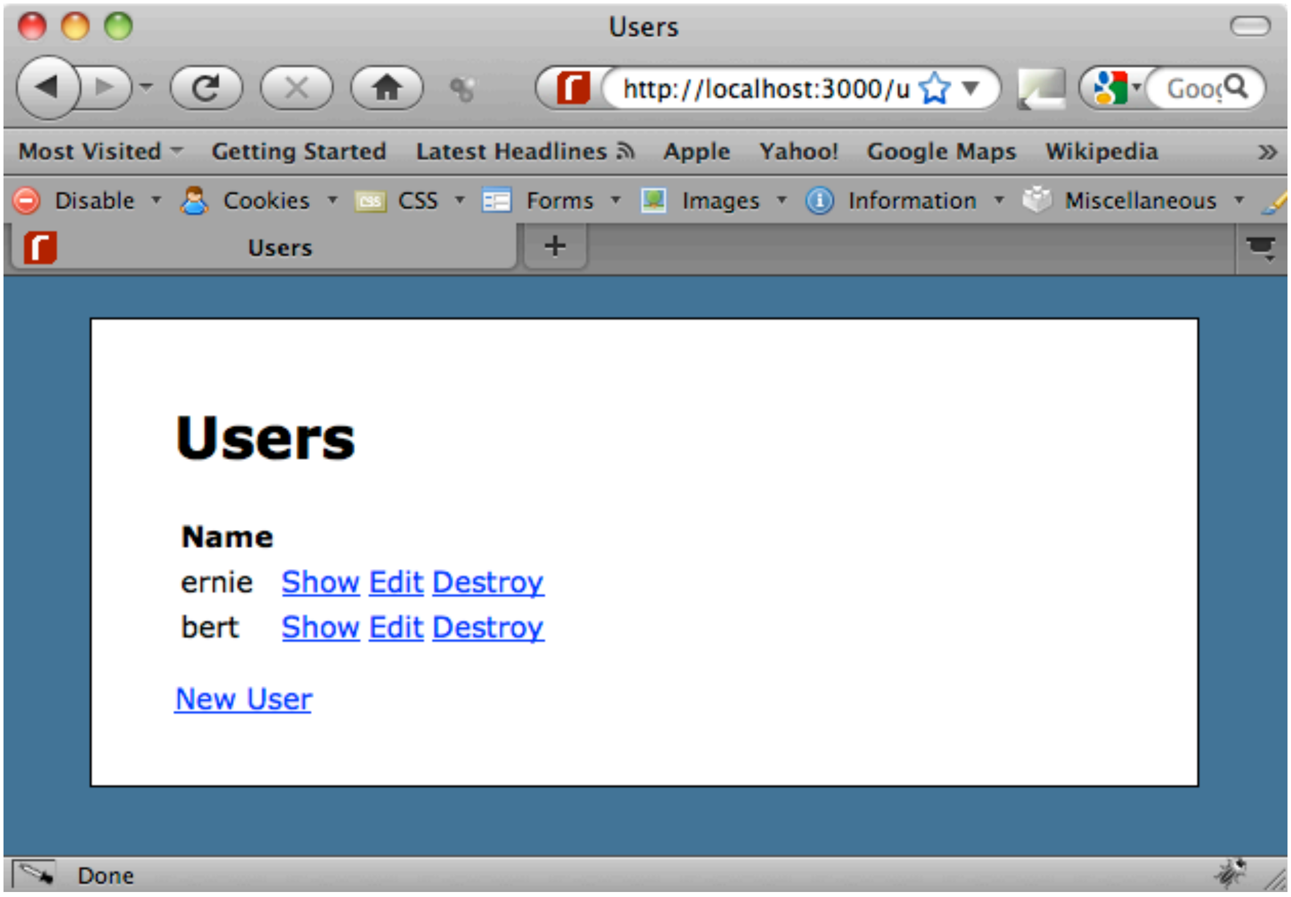
Delete the scaffolded views from the apps/views/uploads directory and create a file called apps/views/uploads/_upload.html.erb and then insert the following code:

```
<div class="upload">
  <%= image_tag upload.photo.url %>
</div>
```

Add the following to the config/routes.rb and delete the index.html.erb from the public directory

```
map.root :controller => 'users'
```

You should now be able to run localhost:3000/users. Create some users, e.g.



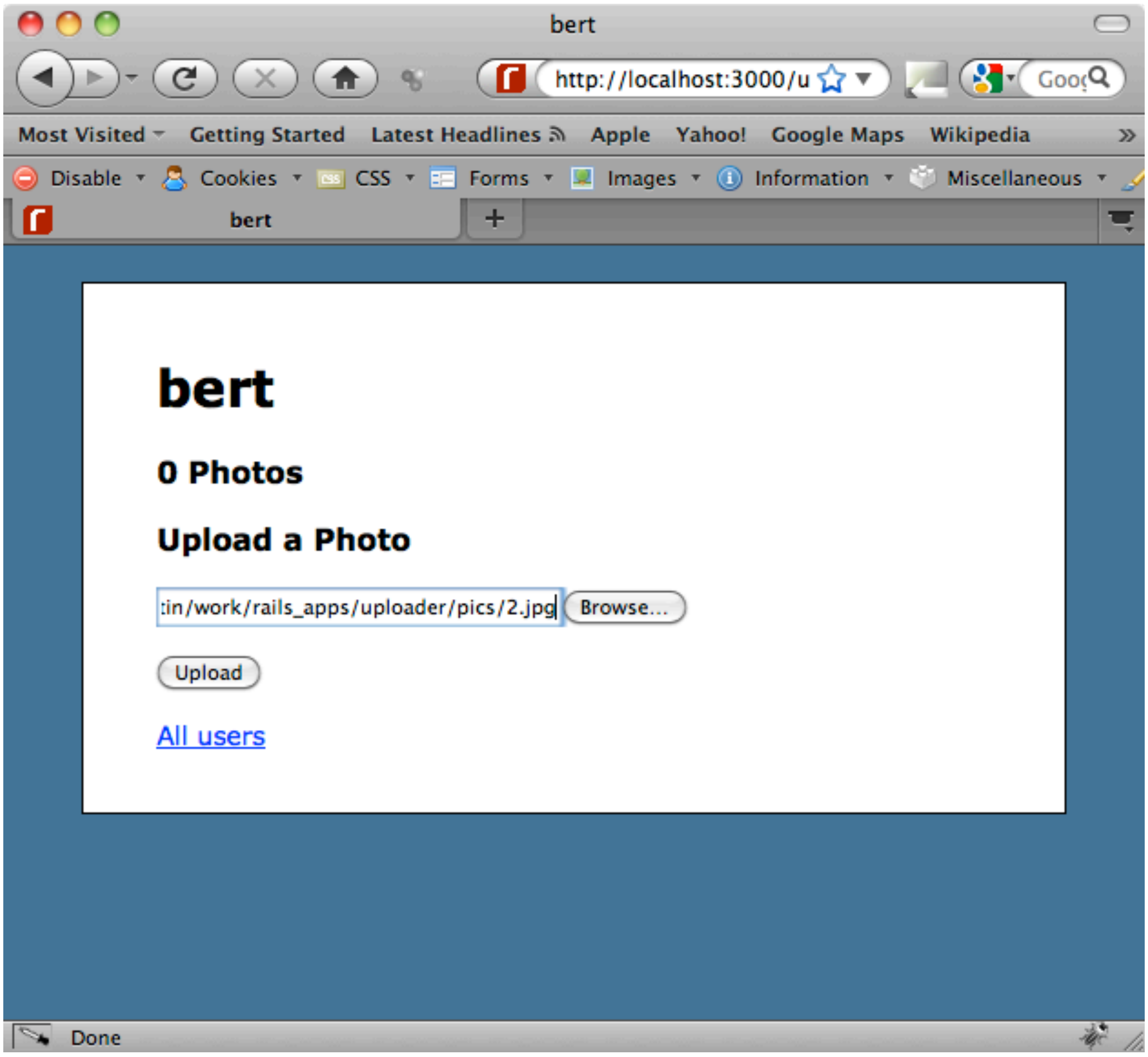
Users

Name	
ernie	Show Edit Destroy
bert	Show Edit Destroy

New User

Then navigate and create images one by one in the traditional html manner.

You'll need some images to upload; I've created some small ones and stored them in a folder called `pics` off the rails application directory. You can pick them up from the sample code.



bert

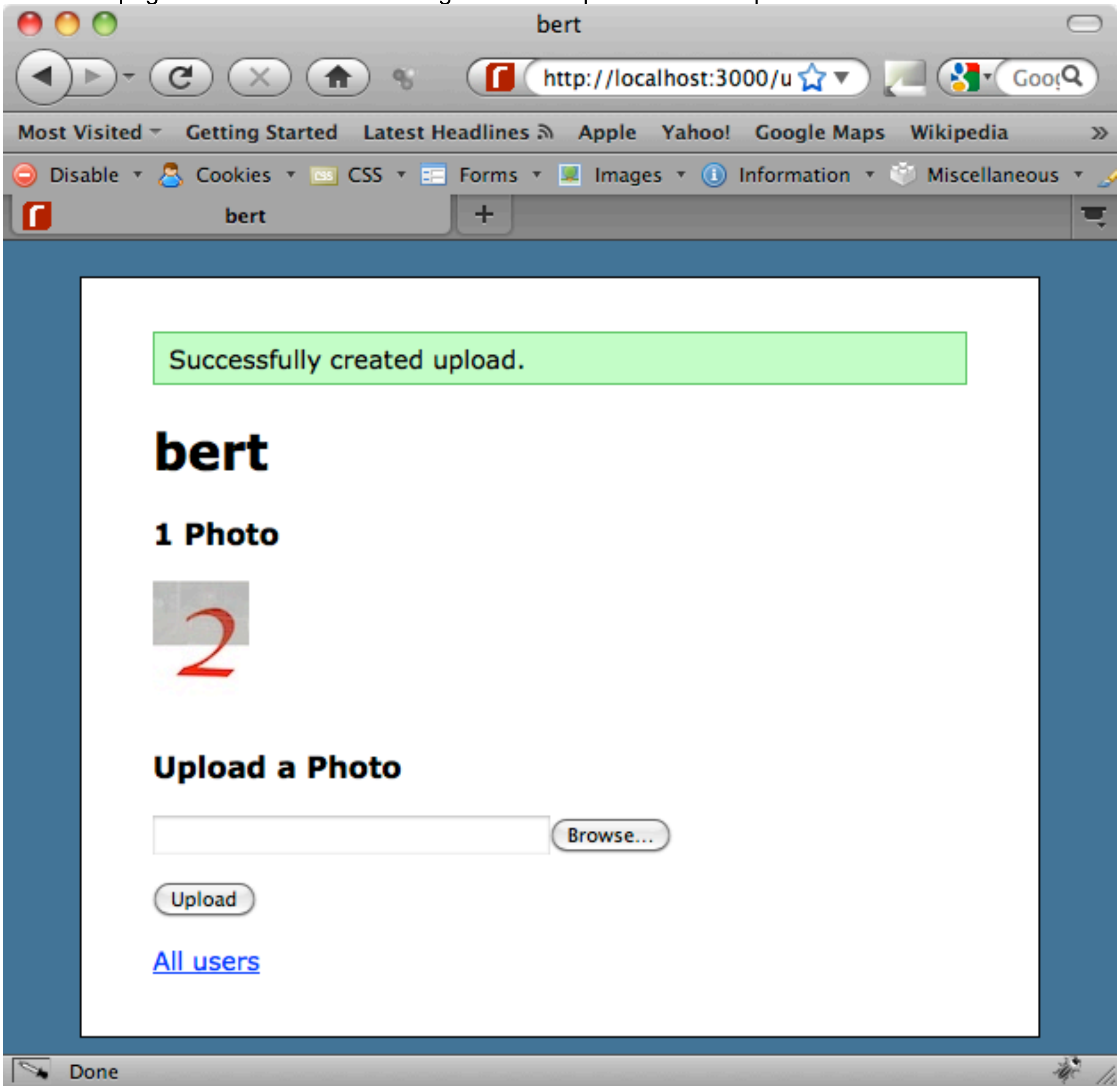
0 Photos

Upload a Photo

:in/work/rails_apps/uploader/pics/2.jpg

All users

The show page will look like the following once the ‘Upload’ button is pressed:



Successfully created upload.

bert

1 Photo

Upload a Photo

Now we need to start adding in the uploadify functionality. Uploadify works with jQuery 1.2.x or greater. I’ve downloaded jQuery v1.4.2 and copied it the `public/javascripts` directory.

Uploadify can be found at http://www.uploadify.com/. You need to expand the zip download and copy the files

`jquery.uploadify.v2.1.0.js`
`swfobject.js` and
`uploadify.swf`

Create public/javascripts/uploadify and store them in there.

The uploadify package also includes an `uploadify.css` file which should be stored in public/stylesheets and `cancel.png` which should be stored in public/images.

Replace the following line in `app/views/layouts/application.html.erb`

```
<%= stylesheet_link_tag 'application' %>
```

with the following two lines:

```
<%= stylesheet( 'application', 'uploadify') %>
<%= javascript( 'jquery-1.4.2') %>
```

These lines make use of some helper methods that were included when the the nifty_layout generator ran. (`app/helpers/layout_helper.rb`). See http://github.com/ryanb/nifty-generators if you’ve not used these before.

The idea was to add javascript functionality ‘unobtrusively’ - making zero changes to the html/erb. Unfortunately, I’ve not found that to be possible, so I’ve opted for running a partial from the app/views/user/show view that contains all the javascript. That way, if javascript is disabled, response degrades nicely to html only as the javascript partial will be ignored.

Add
`<%= render :partial => "show" %>` to the beginning of the `app/views/user/show.html.erb` file and create the partial `_show.html.erb` in the same `app/views/user` folder.

Add the following line to the show partial:

```
<%=
 javascript_include_tag "uploadify/swfobject", "uploadify/jquery.uploadify.v2.1.0.js" %>
```
Note that the detail here depends upon the version of uploadify that you have downloaded.

It’s sensible to test this now - run the app and choose ‘show’ for a user from the opening page. Check the log - if there are any routing errors for the two files, check the folder and file names now.

Assuming all is ok, add the following lines to the show partial:
```
<script type="text/javascript" charset="utf-8">
  $(document).ready(function() {
    $('#upload_photo').click(function(event){
      event.preventDefault();
    });
  $('#upload_photo').uploadify({
    'uploader'            : '/javascripts/uploadify/uploadify.swf',
    'script'              : '/uploads/create',
    'multi'               : true,
    'cancelImg'           : '/images/cancel.png'
    });
  $('#upload_submit').click(function(event){
    event.preventDefault();
    $('#upload_photo').uploadifyUpload();
    });
  });

</script>
```

The `$(document).ready(function() )` is a very common jQuery function (if you have not met jQuery, take a look at Ryan Bates Railscast #136 (http://railscasts.com/episodes/136-jquery) as an intro).

The
```
$('#upload_photo').click(function(event){
      event.preventDefault();
```
stops the default html browse button functionality and the next part (`$('#upload_photo').uploadify`), fairly obviously, loads the uploadify functionality, hence the change of appearance .

The
```
$('#upload_submit').click(function(event){
    event.preventDefault();
    $('#upload_photo').uploadifyUpload();
});
```
stops the default html submit functionality and instead, fires the uploadify function.

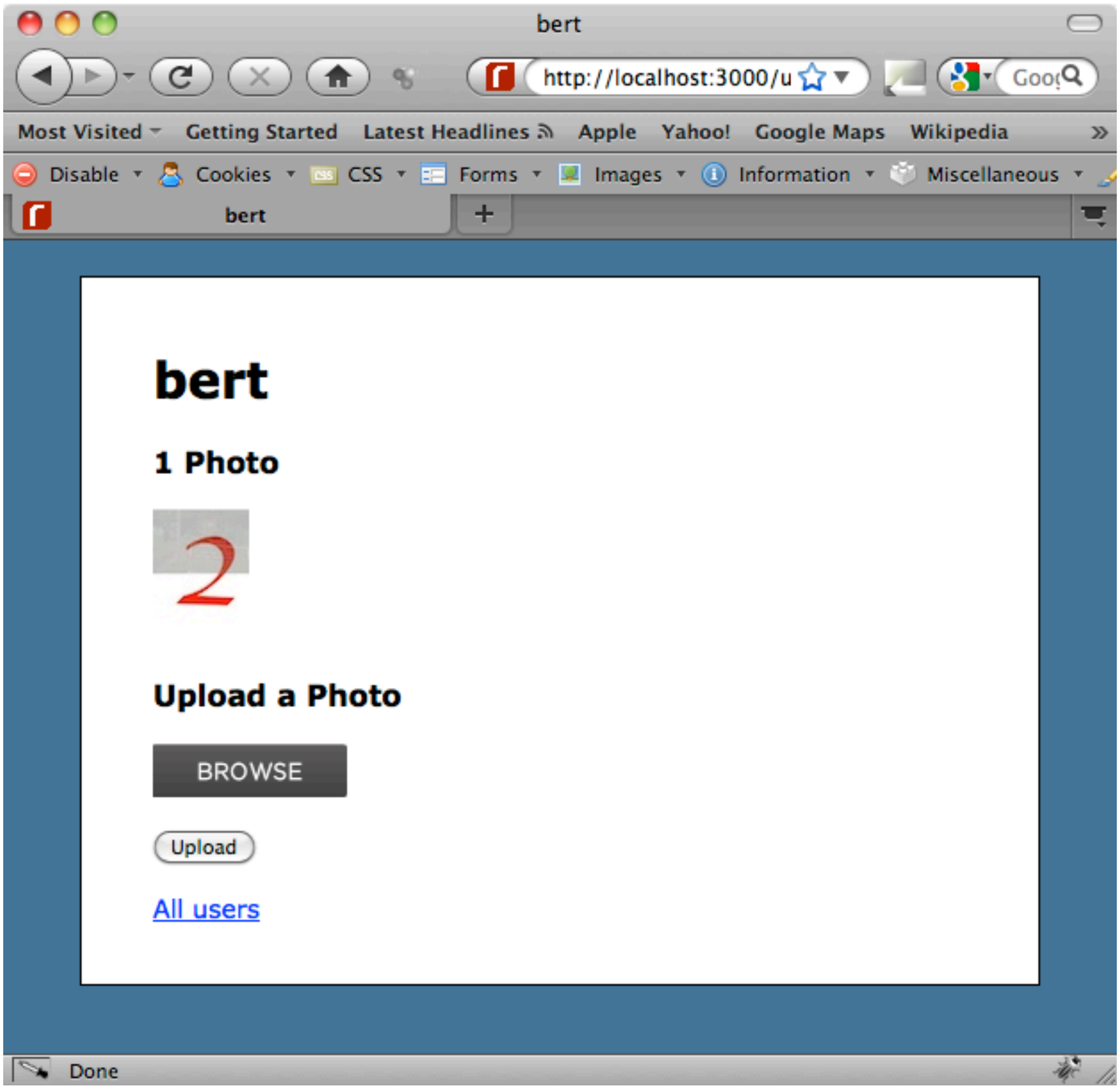
Taking a look at the uploadify properties individually:

`'uploader'    : '/javascripts/uploadify/uploadify.swf'` - path to the Flash file.

`'script'      : '/uploads/create'` - this is the Rails script that will handle the uploading of the files. Note this is not users/show, as the form_for in that script is for Upload.new.

`'multi'       : true` - enables the uploading of multiple files `'cancelImg'   : '/images/cancel.png'` - this is the image that appears in the top right corner of each file listing once 'select' has been clicked, allowing the user to cancel a file selection if the wrong one has been clicked.

So, assuming that javascript is enabled, you should see the following:

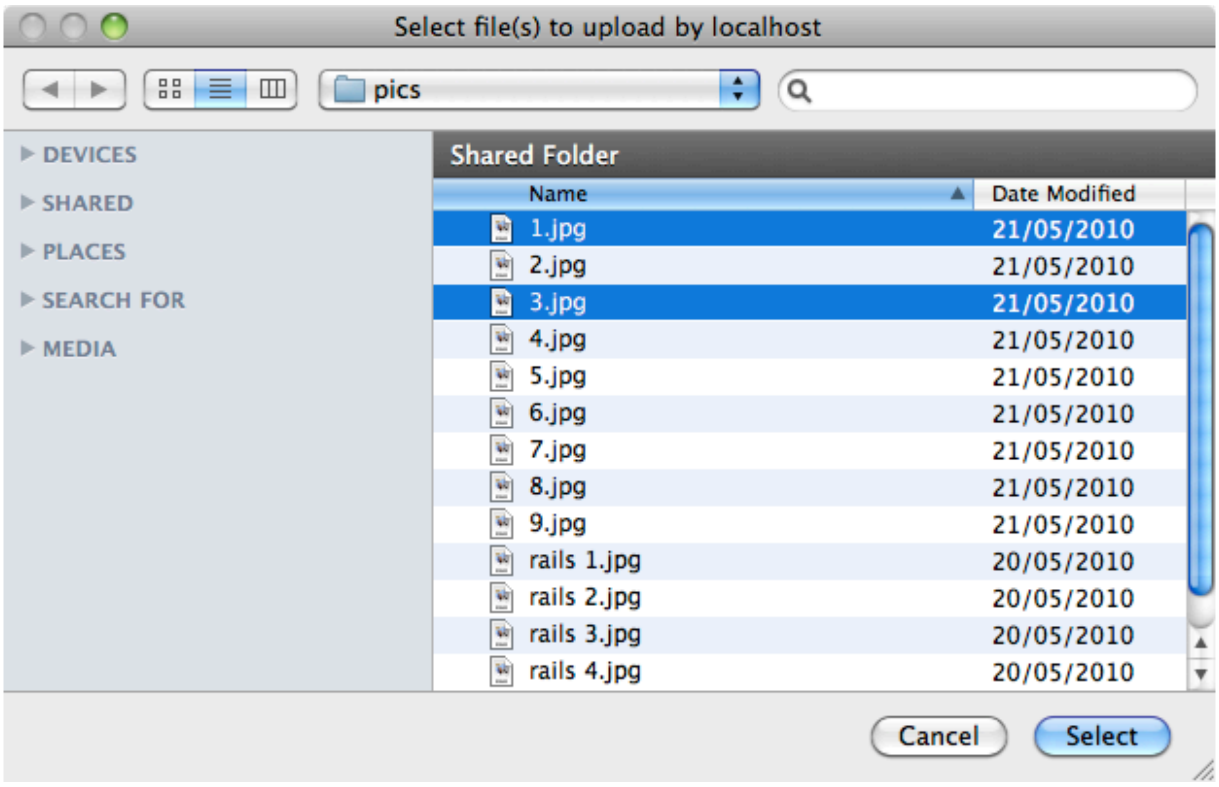


bert

1 Photo

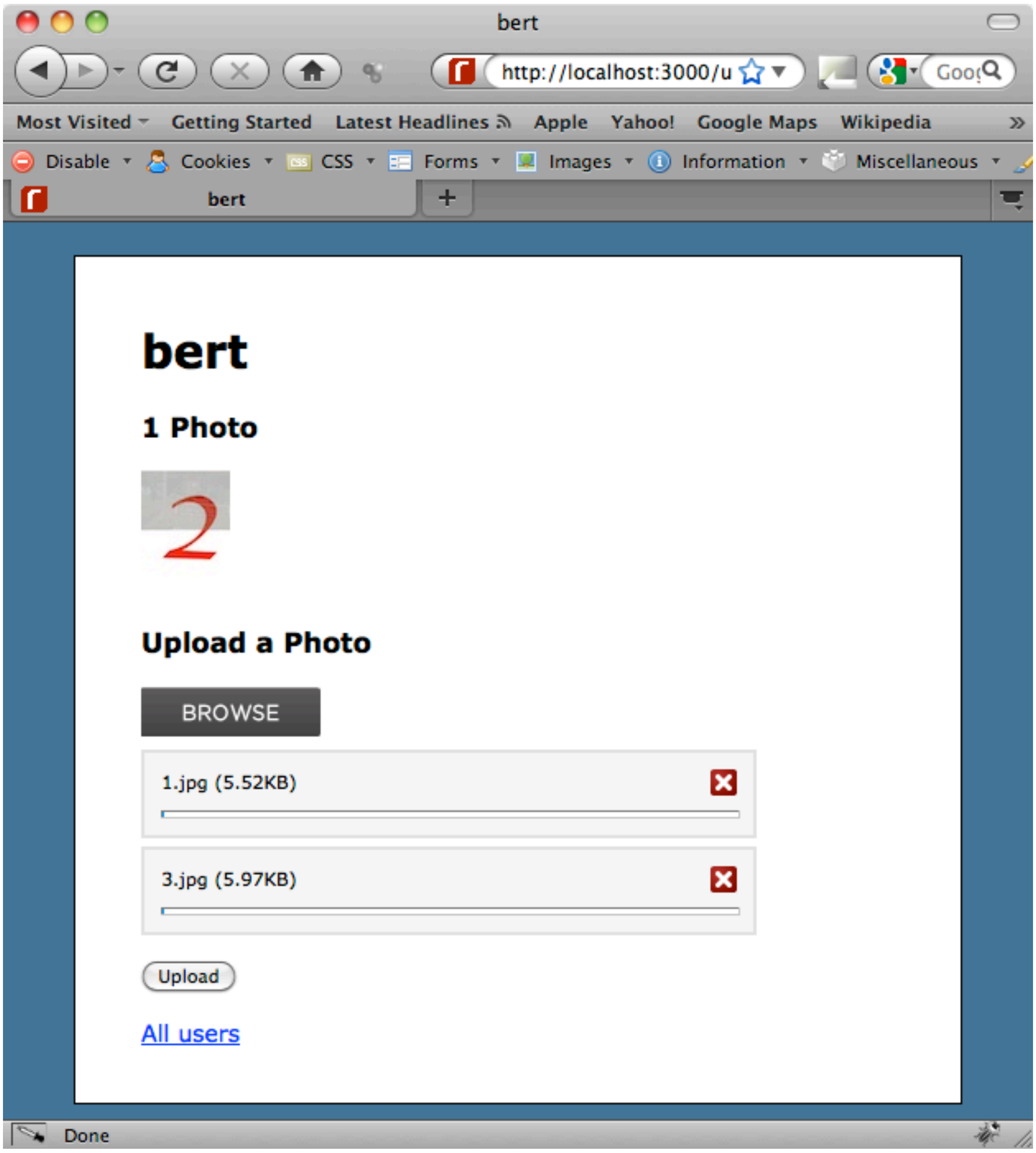
Upload a Photo

Click the 'BROWSE' button and you should be able to select more than file, e.g.:



Name	Date Modified
1.jpg	21/05/2010
2.jpg	21/05/2010
3.jpg	21/05/2010
4.jpg	21/05/2010
5.jpg	21/05/2010
6.jpg	21/05/2010
7.jpg	21/05/2010
8.jpg	21/05/2010
9.jpg	21/05/2010
rails 1.jpg	20/05/2010
rails 2.jpg	20/05/2010
rails 3.jpg	20/05/2010
rails 4.jpg	20/05/2010

Clicking ‘Select’ will produce the following:



bert

1 Photo

Upload a Photo

BROWSE

1.jpg (5.52KB)

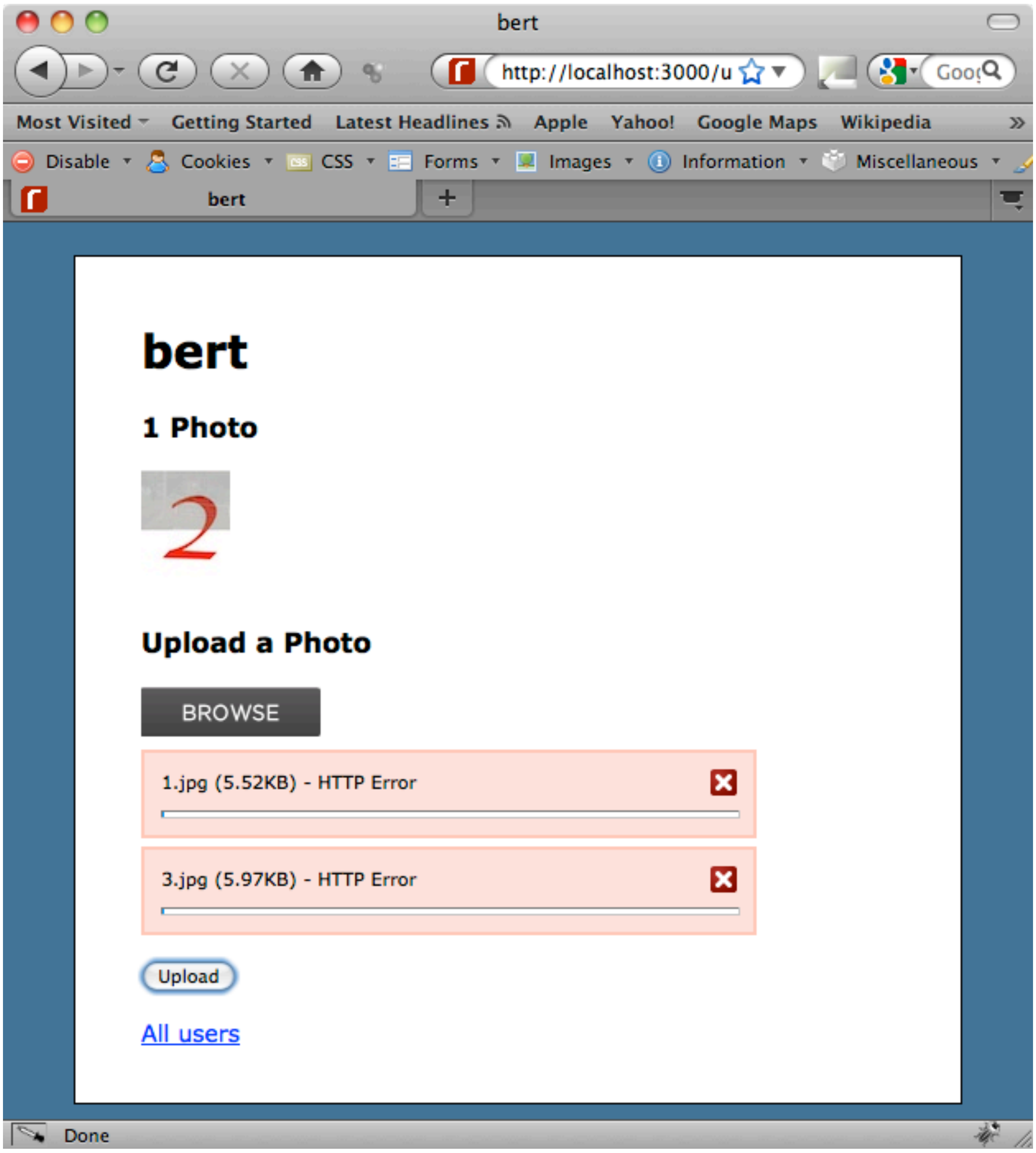
3.jpg (5.97KB)

Upload

All users

The red crosses are the ‘cancel.png’.

Click the ‘Upload’ button and you should see:



bert

1 Photo

Upload a Photo

BROWSE

1.jpg (5.52KB) - HTTP Error

3.jpg (5.97KB) - HTTP Error

Upload

All users

Oops .. well you didn't expect it to be that easy, did you?

A look at the log suggests some problems to be solved:

```
Processing UploadsController#create (for 127.0.0.1 at 2010-06-10 10:43:32) [POST]
  Parameters: {"Filename"=>"1.jpg", "folder"=>"/users/", "Upload"=>"Submit Query",
"Filedata"=>#<File:/var/folders/o5/o5TU3m-EE5asAuG89-sQcU+++TI/-Tmp-/
RackMultipart20100610-301-hg8dmj-0>}

ActionController::InvalidAuthenticityToken (ActionController::InvalidAuthenticityToken):
```

The parameters hash being delivered is very different from the usual one that Rails expects, so we're going to have to do some work with that. The InvalidAuthenticityToken problem often happens with Ajax calls as the hidden field that embeds the AuthenticityToken is not present. Ajax calls are having no understanding of the Rails session.

One other thing we've omitted here is to use a respond_to block in the uploads controller, as we're submitting via javascript, right?

One thing at a time - let's deal with the Authenticity issues first.

We need to return the form_authenticity_token, or the baked in Rails 'protect_from_forgery' stuff is rendered useless. Also, Flash knows nothing about Rails sessions, so we need to address this too; this is where Rack helps out.

Rack intercepts calls before they reach the server and can do good things with them for you. (See Ryan Bates Railscast #151 http://railscasts.com/episodes/151-rack-middleware).

We're going to use Rack Middleware to overcome this issue.

Start by creating a '`middleware`' folder under app.

Make the app aware of the new folder by adding the following to config/environment.rb

```
%w(middleware).each do |dir|
  config.load_paths << "#{RAILS_ROOT}/app/#{dir}"
end
```

Create a file called '`flash_session_cookie_middleware.rb`' in the new middleware folder and add the following lines:

```
require 'rack/utils'

class FlashSessionCookieMiddleware
  def initialize(app, session_key = '_session_id')
    @app = app
    @session_key = session_key
  end

  def call(env)
    if env['HTTP_USER_AGENT'] =~ /^(Adobe|Shockwave) Flash/
      req = Rack::Request.new(env)
      env['HTTP_COOKIE'] = "#{@session_key}=#{req.params[@session_key]}".freeze unless
req.params[@session_key].nil?
    end

    @app.call(env)
  end
end
```

Unfortunately, documentation and examples relating Rack in the Rails world are a bit thin on the ground. See http://rack.rubyforge.org/doc/Rack/Request.html for information relating to the `Rack::Request.new` in the above. A call from uploadify will satisfy the 'if' condition above, and the cookie will be modified as a result.

In config/initializers/session_store.rb, add this (this is all one line) to insert the modification so that the controller receives it:

```
ActionController::Dispatcher.middleware.insert_before
(ActionController::Session::CookieStore, FlashSessionCookieMiddleware,
ActionController::Base.session_options[:key])
```

(Credit due to to Timmy Crawford http://timmyc.posterous.com/uploadify-on-rails-with-paperclip for the above code).

We've changed some of the startup files, so stop and restart the server to make the changes stick. Running the app now and trying to upload will still cause an error (as we've still work to do!) but you should see a reference to the `app/middleware/flash_session_cookie_middleware` file, so we know it's being called.

The question is - how do we use it?

One of the unused properties in the uploadify() function is called '`scriptData`' which is 'An object containing name/value pairs of additional information you would like sent to the upload script. {'name': 'value'}'.

We can use this by editing the _show partial: (The bold text are changes)

```
<%- session_key_name = ActionController::Base.session_options[:key] -%>
<%= javascript_include_tag "uploadify/swfobject" %>
<%= javascript_include_tag "uploadify/jquery.uploadify.v2.1.0.js" %>

<script type="text/javascript" charset="utf-8">

  $(document).ready(function() {
      $('#upload_photo').click(function(event){
     event.preventDefault();
    });
      $('#upload_photo').uploadify({
              'uploader'          : '/javascripts/uploadify/uploadify.swf',
              'script'            : '/uploads/create',
              'multi'             : true,
              'cancelImg'                : '/images/cancel.png',
              'scriptData': {
                   '<%= session_key_name %>' : encodeURIComponent('<%= u cookies
[session_key_name] %>'),
                              'authenticity_token'   : encodeURIComponent('<%= u
form_authenticity_token if protect_against_forgery? %>')
                }
       });
       $('#upload_submit').click(function(event){
      event.preventDefault();
      $('#upload_photo').uploadifyUpload();
    });
  });

</script>
```

(The `encodeURIComponent` used above is a core javascript function that escapes characters.)

Note that we could bake the session name in, but in the cause of making our code less brittle, the session named is defined by the first line as shown. Note also, the check to see if protect_against_forgery is enabled, again in the cause of making the code less brittle.

Run the app now and try to save uploads - you'll see that while we still get an error, the Authenticity issue has gone away.

You might want to check your database, because we'll have been saving some null records so far - zero the database.

We'll now address the issue of the params hash being in the incorrect format for Rails to deal with.

Rails is expecting the data to packed into a params[:upload] hash and it isn't coming back from uploadify in the correct form. The first issue is that the user_id isn't even being returned, so we'll add another line to the scriptData properties in the _show partial (don't forget to put a comma at the end of the `authenticity_token` line):

```
'user_id'  : '<%= @user.id %>'
```

If you check the development log by trying to upload, you'll see the user_id is now passed, although not in the correct form.

We need to coerce the parameters into the form that Rails wants to play with, so make sure the beginning of the uploads_controller/create action reads as follows:

```
newparams = coerce(params)
@upload = Upload.new(newparams[:upload])
```

and add a private method to the uploads controller. This will check to see if the `params[:upload]` hash exists; if it does not, then the call is from `uploadify` and the `params[:upload]` hash is constructed and passed back. If `params[:upload]` does exists then the method returns it.

```
private
  def coerce(params)
    if params[:upload].nil?
      h = Hash.new
      h[:upload] = Hash.new
      h[:upload][:user_id] = params[:user_id]
      h[:upload][:photo] = params[:Filedata]
      h
    else
      params
    end
  end
```

If you’ve been checking the database, you’ll see that the content type for the file is being incorrectly reported as ‘application/octet-stream’. We can change that by using the mime-types gem (which we installed at the beginning but haven’t used yet).

Add (one line)

```
params[:upload][:photo].content_type = MIME::Types.type_for(params[:upload]
[:photo].original_filename).to_s
```

to the coerce method

```
def coerce(params)
  if params[:upload].nil?
    h = Hash.new
    h[:upload] = Hash.new
    h[:upload][:user_id] = params[:user_id]
    h[:upload][:photo] = params[:Filedata]
    h[:upload][:photo].content_type =
      MIME::Types.type_for(h[:upload][:photo].original_filename).to_s
    h
  else
    params
  end
end
```

(I think the above really lives in the model, but to keep focussed on the issue...)

If you try uploading files now, you’ll still see HTTP errors being reported by uploadify in the browser, but a look at the database will show us that the files are being correctly reported to the database (including the mime-type) and a check of public/system shows us that the files are being stored in the default location.

What we haven’t done is amend the uploads_controller to use a respond_to block, after all, we’re using javascript, right?

Rails decides what format to use in the respond_to block depending on the value of format in the request.

If you add a debugger after the flash[:notice] line in uploader_controller/create you can inspect the value that is being submitted for this by querying

```
@_request.format.to_sym
```

If you do this, you will see ‘`:multipart_form`’, which needs changing. To do so, we can add yet another line to the scriptData properties in the uploadify call the in _show partial. Let’s try javascript; we could then write a short script that would update the show view with the new number of photos and append the image to the list being displayed.

So ... the scriptData property should now look like this:

```
{ '<%= session_key_name %>' : encodeURIComponent('<%= u cookies[session_key_name] %>'),
  'authenticity_token'      : encodeURIComponent('<%= u form_authenticity_token if
protect_against_forgery? %>'),
  'user_id'                 : '<%= @user.id %>',
  ‘format‘                  : ‘js’ }
```

If you try that and use the debugger in the uploads_controller you’ll see that, indeed,

```
@_request.format.to_sym => :js
```

Now we'll edit the create action to respond to the js request. In the uploads_controller create method, replace the

```
redirect_to @upload.user
```

with

```
respond_to do |format|
  format.html {redirect_to @upload.user}
  format.js  {render :js => "alert('Uploadify');" }
end
```

and try the app to test it. (Don't forget to get rid of the debugger if it's still there).

You will see in the development.log that the create function has indeed been called twice and the database will show that the uploads have been applied. Unfortunately, as no alert appears, the format.js didn't fire.

Going back to the uploadify documentation, we see that one of the uploadify() properties is a function call by the name of `onComplete` that triggers after each file upload. The `onComplete()` function receives four arguments, one of which is '`response`', which is the response from the server.

If we change our format from `:js` to `:json`, we can send what we want back to the browser.

Change the

`'format' : 'js'` to

`'format' : 'json'` in the scriptData property list in the `_show` partial and change the

`format.js  {render :js => "alert('Uploadify');" }` line in the uploads_controller create method to

`format.json { render :json => { :result => 'success', :upload => upload_path(@upload) } }`

As `onComplete` is called after each file upload, the '`response`' property in the call will then be set to whatever

`{ :result => 'success', :upload => upload_path(@upload) } }` evaluates as.

We'll use `console.log` to check that the browser is receiving the correct information. (Console.log works with Safari if you enable the debug features and with Firefox through Firebug. If you develop for the web, I highly recommend using Firefox with Firebug).

`_show.html.erb` should now look like this:

```
<%- session_key_name = ActionController::Base.session_options[:key] -%>
<%= javascript_include_tag "uploadify/swfobject" %>
<%= javascript_include_tag "uploadify/jquery.uploadify.v2.1.0.js" %>

<script type="text/javascript" charset="utf-8">
  $(document).ready(function() {
    $('#upload_photo').click(function(event){
      event.preventDefault();
    });
    $('#upload_photo').uploadify({
      'uploader'          : '/javascripts/uploadify/uploadify.swf',
      'script'            : '/uploads/create',
      'multi'             : true,
      'cancelImg'         : '/images/cancel.png',
       onComplete         : function(event, queueID, fileObj, response, data)
{ console.log(response);},
      'scriptData': {
        'format'          : 'json',
        '<%= session_key_name %>' : encodeURIComponent('<%= u cookies[session_key_name]
%>'),
        'authenticity_token'      : encodeURIComponent('<%= u form_authenticity_token if
protect_against_forgery? %>'),
        'user_id'                 : '<%= @user.id %>'
         }
      });
    $('#upload_submit').click(function(event){
      event.preventDefault();
    $('#upload_photo').uploadifyUpload();
    });
  });
</script>
```

and if you try to upload again, with Firebug enabled, you'll see

```
{"upload":"/uploads/38","result":"success"}
```

in the Firebug console. There should be one of these lines for each file you uploaded and the number ("38" here) represents the id of the filedata in the uploads table.

Now '`/uploads/38`' should route us to the show method in `uploads_controller` for the record with id = 38 if we can get the javascript to run it. jQuery has a function called getScript that will do just that, so let's see what happens. As the response is in JSON format, we need to convert it to a javascript value, so

Change the `onComplete` line to read

```
onComplete : function(event, queueID, fileObj, response, data) { var dat = eval('(' +
response + ')');$.getScript(dat.upload);},
```

Try uploading again and you will see from the development.log that the show action is indeed being called, although it is failing. The call is a javascript call (you can verify this by using the debugger and checking the value of `@_request.format.to_sym`, which will show you that this is a `:js` request.

Now we're finally in a position to update the browser using some javascript.

You don't need to add a respond_to block to the `uploads_controller show` method as it is looking for a `js.erb` file and we can provide one.

If you've followed this so far, you'll know that we want to update the number of upload in the view, and append the new upload to those displayed.

Updating the number requires a knowledge of the total number of uploads in the database, so edit the `show` method in `uploads_controller` to look like:

```
  @upload = Upload.find(params[:id], :include => :user)
  @total_uploads = Upload.find(:all, :conditions => { :user_id => @upload.user.id})
```

and add the following to a `show.js.erb` file created in `app/views/uploads`.

```
$('#photos_count').html('<%= pluralize(@total_uploads.count, "Photo") %>');
```

This uses jQuery's html function to replace the contains of the div called photos_count

To append the new upload to the list, add the line

```
$('#uploads').append("<%= escape_javascript(render(:partial => "uploads/upload")) %>");
```

to the `show.js.erb` file.

I got a deprecation warning here:

```
DEPRECATION WARNING: @upload will no longer be implicitly assigned to upload. (called
from _run_erb_app47views47uploads47_upload46html46erb_locals_object_upload at /Users/
martin/work/rails_apps/uploader/app/views/uploads/_upload.html.erb:2)
```

So edit the append line we just added to:

```
$('#uploads').append("<%= escape_javascript(render(:partial => "uploads/upload", :locals
=> {:upload => @upload})) %>");
```

You should now be able to upload files successfully and see the users/show page updated as you do it.

What I've show is well short of a functioning application of course, but I hope it makes the steps required to make uploadify work with paperclip and rails clear.

Thanks are due to Ryan Bates for his never ending stream of top class Railscasts (http://railscasts.com); to Peter Kordel (http://pkordel.wordpress.com) for getting me thinking along the right lines in the first place.