Direct to S3 Image Uploads in Rails

Last Updated: 17 February 2015

Table of Contents

This article is not updated to use Amazon's v2.0 API.

This article demonstrates how to add direct S3 uploads to a Rails app. While there are many popular S3 image upload solutions for Ruby and Rails such as Paperclip and CarrierWave, these solutions use the server as a temporary cache.

They typically upload the file to Heroku and then stream it to S3. While this works well for small files, larger files may be deleted from the dyno before they can be uploaded to S3 due to Heroku's ephemeral filesystem.

A more stable alternative is to upload the image to S3 directly from the client side, and when the image is fully uploaded save the reference URL in the database. This way it does not matter if our dyno is restarted while the image is uploaded, your dyno does not need to handle the extra load of image uploads, and you don't have to wait for the file to upload twice, once to your dyno and once to S3.

The only downside is that all the logic must be performed on the client side. This article shows how to accomplish this.

Philosophy

This article uses the jQuery-File-Upload plugin and the AWS gem. There are other libraries such as carrier wave direct that may also be able to enable you to upload your images directly to S3, however without the low level knowledge of all the client side considerations implementing them can be difficult.

By using the jQuery-File-Upload plugin we will create a relatively readable and short JavaScript code that can be re-used on any form using an image upload input. The UI behavior is very customizable, and the behavior from a user perspective is very simple. On the Rails side create an AWS presigned-post and store the image URL in the database.

Example app

For the purpose of this example, we will assume you have a `User` model and that you want to store an avatar for each user on S3. This also assumes that you've got a `UsersController`. You can follow along from scratch if you do not yet have a project:

```
$ rails new direct-s3-example
$ cd direct-s3-example
$ rails generate scaffold user name avatar_url
      invoke  active_record
      create    db/migrate/20140519195131_create_users.rb
      create    app/models/user.rb
      invoke    test_unit
      create      test/models/user_test.rb
      create      test/fixtures/users.yml
      # ...
```

Now you can migrate your database:

```
$ rake db:migrate
```

Now open the project in your editor of choice. You will now need to enable your application to communicate with S3

S3

Before we can send a file to S3 you will need an S3 account and appropriately configured bucket.

Please follow the directions for getting an S3 account before continuing.

Now that you've got a Rails app and an S3 account you will need to interact with the files on the client side and you'll need a library for interacting with S3 on the Ruby side.

S3 SDK

We will be using the Amazon Ruby SDK for interacting with S3. In your application's Gemfile add:

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

Now run `bundle install`. For local development we will assume you are using a `.env` file and Foreman. Open up your `.env` file and ensure that you have set an `S3_BUCKET`, `AWS_ACCESS_KEY_ID`, and `AWS_SECRET_ACCESS_KEY` to the values from when you created an S3 bucket.

```
$ cat .env
S3_BUCKET=my-s3-development
AWS_ACCESS_KEY_ID=EXAMPLEKVFOOOWWPYA
AWS_SECRET_ACCESS_KEY=exampleBARZHS3sRew8xw5hiGLfroD/b21p2l
```

Make sure the values are being written into your environment by running this command and ensuring it matches the value in your `.env`.

```
$ foreman run rails runner "puts ENV['S3_BUCKET']"
my-s3-development
```

Once you've got your environment variables set up locally you'll need to instantiate a new S3 object to use in the controller. Create an initializer in `config/initializers/aws.rb`. Here we are going to configure AWS and create a global S3 constant:

```
AWS.config(access_key_id:     ENV['AWS_ACCESS_KEY_ID'],
           secret_access_key: ENV['AWS_SECRET_ACCESS_KEY'] )

S3_BUCKET = AWS::S3.new.buckets[ENV['S3_BUCKET']]
```

You will want to repeat the process of creating a new S3 bucket for your production environment running on Heroku. You should set these values using `heroku config:set` for example if your production bucket it named `my-s3-production` you could set the appropriate value on Heroku by running:

```
$ heroku config:set S3_BUCKET=my-s3-production
```

Repeat this process for `AWS_ACCESS_KEY_ID` and `AWS_SECRET_ACCESS_KEY` with your production credentials.

Cross origin support

By default browsers will not let you make JavaScript calls to other services other than the currently rendered page. If this protection did not exist, than when you log into any webpage they could in send requests to other services (such as Facebook or GitHub), and if you are currently logged in, they could receive private data. Luckily this security mechanism is built in by default but it also prevents us from sending files to any other URL than the one we're currently on. So by default we can not use JavaScript to send a file from our website to S3. To enable this functionality we must use CORS.

CORS stands for Cross Origin Resource Sharing and essentially allows you to whitelist where an HTTP request can come from. So we need to tell our S3 bucket that it is okay to accept file a from our server via JavaScript.

We will need two different buckets for production and development. In your development bucket you will need to modify your CORS settings. Here's an example setting to allow your local machine running at `localhost:3000` to send files to AWS:

```
<?xml version="1.0" encoding="UTF-8"?>
<CORSConfiguration xmlns="http://s3.amazonaws.com/doc/2006-03-01/">
  <CORSRule>
    <AllowedOrigin>http://localhost:3000</AllowedOrigin>
    <AllowedMethod>GET</AllowedMethod>
    <AllowedMethod>POST</AllowedMethod>
    <AllowedMethod>PUT</AllowedMethod>
    <AllowedHeader>*</AllowedHeader>
  </CORSRule>
</CORSConfiguration>
```

You will need to make sure that your production bucket has an appropriate origin set for `AllowedOrigin`.

Pre-signed post

We will be using a pre-signed POST generated for us from the AWS ruby gem. Pre-signed URLs are useful if you want your user/customer to be able upload a specific object to your bucket, but you don't require them to have AWS security credentials or permissions. You can read more about pre-signed posts in the documentation. Essentially they are an easy way for us to configure all the settings of where the S3 object (in this case an image) will live, and any constraints on it.

Open your `app/controllers/users_controller.rb` It should look like this:

```
# GET /users/new
def new
  @user = User.new
end
```

Add a line of code here to create a new pre-signed post object:

```
# GET /users/new
def new
  @s3_direct_post = S3_BUCKET.presigned_post(key: "uploads/#{SecureRandom.uuid}/${filename}", succe
  @user = User.new
end
```

There is quite a bit going on here, so here is the breakdown of options to the `presigned_post` method:

Pre-signed post options: key

The key is where the object will live in S3. In S3 you cannot have duplicate keys, so each must be unique. AWS supports a custom `${filename}` directive. While the syntax looks similar to the way Ruby code is inserted into strings (using `"#{}`) this is string that AWS understands has special connotation. This `${filename}` directive tells S3 that if a user uploads a file named `pic.png`, then S3 should store the final file with the same name of `pic.pn`. We want to make sure two different users with the same named file can both store their avatars so we add a random string using `SecureRandom.uuid`. Finally all images will be delegated to a base level of `uploads` so that if you are storing other items in your bucket it is easier to keep user uploaded files separate.

Pre-signed post options: success_action_status

This is the HTTP status code that you want AWS to return when an image is stored successfully. Since we are using the `${filename}` directive, we will rely on AWS telling us the name of the key that the file is stored in. To do that we need to set `success_action_status` to 201. From the AWS docs on creating new objects.

If the value is set to 201, Amazon S3 returns an XML document with a 201 status code.

We want to receive an XML response that we can parse the key. If we omit the 201 status code, we cannot do that. Setting this value to 201 is very important.

Pre-signed post options: acl

Access control lists or ACL is how AWS allows you to control who can see, edit and delete files. Since we want our avatar URL's to be visible by everyone we can set this value to `:public_read`. This means that the files the user uploads to your S3 bucket are visible by everyone but only administrators of the S3 bucket can edit the file. If you do not set an `acl` then no one will be able to view your user's avatar.

Client side code

Now that we have a pre-signed post we can use the information in the object to send images to S3 on the client side.

Since we cannot rely on our server as a temporary cache, we must use client side code (JavaScript) to deliver the files to S3. HTML 5 introduced a file API, unfortunately it was no supported by IE until version 10. To get around this we will use the jQuery File Upload plugin. Before we can use this library though we will first need JQuery UI":

```
$ curl -O \
https://raw.githubusercontent.com/jquery/jquery-ui/master/ui/widget.js \
>> app/assets/javascripts/jquery.ui.widget.js
```

Next get the fileupload JavaScript into your Rails project:

```
$ curl -O \
https://raw.githubusercontent.com/blueimp/jQuery-File-Upload/master/js/jquery.fileupload.js \
>> app/assets/javascripts/z.jquery.fileupload.js
```

We are forcing the file to be loaded after any other jquery files by appending a `z` to its name.

If you are loading all JavaScript files in your `application.js` with a `//= require_tree .` directive than this JavaScript will be automatically available otherwise you will need to explicitly require it:

```
//= require z.jquery.fileupload
```

Start your local server:

```
$ rails s
```

Load localhost:3000/users/new in your browser and verify that your fileupload JavaScript file is present. Open up a JavaScript console, in Chrome you can press `CMD+Option+C`. In the console verify that jQuery is loaded correctly:

```
> console.log($)
function $(selector, [startNode]) { [Command Line API] }
```

Now verify that the `fileupload` function is available

```
> console.log($().fileupload)
function ( options ) {
          var isMethodCall = typeof options === "string",
               args = slice.call( arguments, 1 ),
               returnValue = this;
//...
```

If you get no result back, verify that there are no errors in your JavaScript console, that all your required files are listed when you view source, that they are non-empty and are in the correct order.

Prepare the view

To complete the S3 upload process we must first send the file to S3, then store the URL in the database. For this task we have an `avatar_url` string field in the Users table.

In JavaScript we will need some way of identifying fields that contain images we want to upload directly to S3. First we can open the form that we know has the file upload field `app/views/users/_form.html.erb`. Here you should see a form:

```
<%= form_for(@user) do |f| %>
  <% if @user.errors.any? %>
    <div id="error_explanation">
  <!-- ... -->
```

We need a way to target this form with JavaScript, so we can add a custom class named `directUpload`:

```
<%= form_for(@user, html: { class: "directUpload" }) do |f| %>
  <% if @user.errors.any? %>
    <div id="error_explanation">
  <!-- ... -->
```

We will also need a file field so the user can upload a photo. Look for the `avatar_url` field which should be a `text_field`:

```
<div class="field">
  <%= f.label :avatar_url %><br>
  <%= f.text_field :avatar_url %>
</div>
```

Change this so that it is now a `file_field`:

```
<div class="field">
  <%= f.label :avatar_url %><br>
  <%= f.file_field :avatar_url %>
</div>
```

Detecting file field on the client side

At this point we have a JavaScript library for uploading images, we have an S3 bucket, a valid pre-signed post object, User model with an `avatar_url` string field and a view with a file input field. We need to get our user's image to S3, and store the URL back to `avatar_url`, this will be a very manual process, mostly via JavaScript. You may wish to customize your experience later, and we will discuss available options.

Either in a set of `<script></script>` tags for debugging or in your `application.js` file we will first find any forms with our custom class and then iterate through all file fields:

```
$(function() {
  $('.directUpload').find("input:file").each(function(i, elem) {
    var fileInput    = $(elem);
    console.log(fileInput);
  });
});
```

Now when you load your page, you should see a console output for each file field (there should only be one for `avatar_url`). From here we can pull out other useful elements, we will need the form that holds the file input, the submit button to the form, we will also create a progress bar for each file element:

```
$(function() {
  $('.directUpload').find("input:file").each(function(i, elem) {
    var fileInput    = $(elem);
    var form         = $(fileInput.parents('form:first'));
    var submitButton = form.find('input[type="submit"]');
    var progressBar  = $("<div class='bar'></div>");
    var barContainer = $("<div class='progress'></div>").append(progressBar);
    fileInput.after(barContainer);
  });
});
```

Now is a good time to make sure your progress bar has some styling. In your `app/assets/stylesheets/screen.css` add this:

```
.progress {
  max-width: 600px;
  margin:    0.2em 0 0.2em 0;
}

.progress .bar {
  height:  1.2em;
  padding: 0.2em;
  color:   white;
  display: none;
}
```

Finished jquery-file-upload code

Now that we have all the elements required you can call `fileInput.fileupload({})` on each file input element and pass in options (we will add callbacks later):

```
$(function() {
  $('.directUpload').find("input:file").each(function(i, elem) {
    var fileInput    = $(elem);
    var form         = $(fileInput.parents('form:first'));
    var submitButton = form.find('input[type="submit"]');
    var progressBar  = $("<div class='bar'></div>");
    var barContainer = $("<div class='progress'></div>").append(progressBar);
    fileInput.after(barContainer);
    fileInput.fileupload({
      fileInput:       fileInput,
      url:             '<%= @s3_direct_post.url %>',
      type:            'POST',
```

```
      autoUpload:       true,
      formData:         <%= @s3_direct_post.fields.to_json.html_safe %>,
      paramName:        'file', // S3 does not like nested name fields i.e. name="user[avatar_url]"
      dataType:         'XML',  // S3 returns XML if success_action_status is set to 201
      replaceFileInput: false
    });
  });
});
```

This is everything you need to send files to S3, however it does not allow you to retrieve the image URL, or update the UI to let your user know what is going on. Those will all be added in the 「callbacks」 section below.

First we will cover the options passed into `fileupload`. You can reference the jquery-file-upload options documentation for additional information.

Options: fileInput

This option is a reference to where jQuery-File-Upload can find the file, in this case we are grabbing it from the file input field of our form.

Options: url

This is the URL that we will submit the image to. We want to send it to the URL generated from our prior work with setting up an AWS presigned-post:

```
url: '<%= @s3_direct_post.url %>'
```

Options: type

This is the type of HTTP request that will be used, in this case we want to 「create」 an object on S3 so we need to send a `POST` HTTP request.

Options: autoUpload

When your user selects a file, it will begin to automatically upload. We do this so that if they have a lot of other information to fill out the image can upload while they are filling out the rest of the form. We need to manually render progress bars to relay the status of the image upload.

If you set this value to `false` then you must manually trigger the upload action via JavaScript.

Options: formData

In addition to generating a URL, the pre-signed post sets up all the required data needed to send our image such as our AWS access keys. To make use of the information already in this object we can call:

```
formData: <%= @s3_direct_post.fields.to_json.html_safe %>,
```

Options: paramName

This is the 「name」 of the field that is being submitted, by default Rails will generate html that looks like `name="user[avatar_url]"`. S3 does not like this type of a nested name field, so we name it to anything else, in this case we use 「file」

```
paramName: 'file'
```

Options: dataType

This is where we specify our the type of data we expect back after our image upload. AWS will return `XML` if our `success_action_status` is set to 201, so here we tell jQuery-File-Upload to expect XML back.

Options: replaceFileInput

By default jQuery-File-Upload will duplicate image inputs and replace the image inputs on page with the duplicates. This creates some unexpected visual behavior, and makes it harder to manipulate the `fileInput` object later if you want. To get around this, we simply disable this feature.

jQuery-File-Upload callbacks

We use a number of callbacks set via options `progressall`, `start`, `done`, and `fail`. We need them to show the progress of uploads to our user, control the UI and to set appropriate value for `avatar_url` when we submit the form. The full code with callbacks is avaliable here:

```
$(function() {
  $('.directUpload').find("input:file").each(function(i, elem) {
    var fileInput    = $(elem);
    var form         = $(fileInput.parents('form:first'));
    var submitButton = form.find('input[type="submit"]');
    var progressBar  = $("<div class='bar'></div>");
    var barContainer = $("<div class='progress'></div>").append(progressBar);
    fileInput.after(barContainer);
    fileInput.fileupload({
      fileInput:       fileInput,
      url:             '<%= @s3_direct_post.url %>',
      type:            'POST',
      autoUpload:       true,
      formData:         <%= @s3_direct_post.fields.to_json.html_safe %>,
      paramName:        'file', // S3 does not like nested name fields i.e. name="user[avatar_url]"
      dataType:         'XML',  // S3 returns XML if success_action_status is set to 201
      replaceFileInput: false,
      progressall: function (e, data) {
        var progress = parseInt(data.loaded / data.total * 100, 10);
        progressBar.css('width', progress + '%')
      },
      start: function (e) {
        submitButton.prop('disabled', true);

        progressBar.
          css('background', 'green').
          css('display', 'block').
          css('width', '0%').
          text("Loading...");
      },
      done: function(e, data) {
        submitButton.prop('disabled', false);
        progressBar.text("Uploading done");

        // extract key and generate URL from response
        var key   = $(data.jqXHR.responseXML).find("Key").text();
        var url   = '//<%= @s3_direct_post.url.host %>/' + key;

        // create hidden field
        var input = $("<input />", { type:'hidden', name: fileInput.attr('name'), value: url })
        form.append(input);
      },
      fail: function(e, data) {
        submitButton.prop('disabled', false);

        progressBar.
          css("background", "red").
          text("Failed");
      }
    });
  });
});
```

This is everything you need to set the appropriate `avatar_url` in the database on form submission, show a progress loading bar while the image uploads to S3, and prevent the user from accidentally submitting the form while an image upload is in progress.

It can be helpful to add debugging `console.log("done");` code inside of your callbacks to be sure they are being called properly, just don't forget to take them out before you deploy to production.

In the next section we will cover each callback and the code it contains.

Callbacks: progressall

The `progressall` callback is called with an event object, and a data object. The data object contains the total size of files being uploaded `data.total` and the size of files that has currently been uploaded `data.loaded`. With these two we can calculate the percentage progress and show it with our progress bar:

```
progressall: function (e, data) {
  var progress = parseInt(data.loaded / data.total * 100, 10);
  progressBar.css('width', progress + '%')
}
```

Note that we're creating a progress bar for every image input and this `progressall` callback contains only information about one image as we're initializing a seperate `fileupload` for each image.

Callbacks: start

This callback is called at the beginning of a file upload. We use this to show our progress bar, and to disable the submit button so that a user cannot accidentally submit the form when an image is only half way uploaded:

```
start: function (e) {
  submitButton.prop('disabled', true);

  progressBar.
    css('background', 'green').
    css('display', 'block').
    css('width', '0%').
    text("Loading...");
}
```

Callbacks: done

This callback is called on a successful image being sent to S3.

The entire callback together looks like this:

```
done: function(e, data) {
  submitButton.prop('disabled', false);
  progressBar.text("Uploading done");

  // extract key and generate URL from response
  var key   = $(data.jqXHR.responseXML).find("Key").text();
  var url   = '//<%= @s3_direct_post.url.host %>/' + key;

  // create hidden field
  var input = $("<input />", { type:'hidden', name: fileInput.attr('name'), value: url })
  form.append(input);
}
```

While short, the funcitonality is dense, here is what is going on.

First we want to re-enable submit functionality now that the file is done uploading:

```
submitButton.prop('disabled', false);
```

We also want to update the progress bar text:

```
progressBar.text("Uploading done");
```

Most importantly we want to parse out the XML response from S3 to get the Key.

```
var key = $(data.jqXHR.responseXML).find("Key").text();
```

Once we have the key we build a protocol relative url to the image:.

```
var url = '//<%= @s3_direct_post.url.host %>/' + key;
```

With this data we want to create a hidden field that has the url string as a value and the name of the original element. We can pull out the name from our element using `fileInput.attr('name')`.

```
var input = $("<input />", { type:'hidden', name: fileInput.attr('name'), value: url })
```

So now the new input element will have the same name (in this case it would be`name="user[avatar_url]"`). So when the user submits the form the URL is transmitted back to the `create` action in the UserController and saved:

```
form.append(input);
```

Although the `user[avatar_url]` name has already been used in the file field, the hidden element will take precedence as it

appears later in the form. Now when the user submits the form, the URL will be present in the params.

Callbacks: fail

If something goes wrong, and the image is not uploaded we need to alert the user of the failed upload, and re-enable the submit button (just incase they need to submit even if the image didn't work).

We will do this by turning the progress bar red and emitting a failure text:

```
fail: function(e, data) {
  submitButton.prop('disabled', false);

  progressBar.
    css("background", "red").
    text("Failed");
}
```

Submitting and rendering the images

Once the image is successfully uploaded to S3 an a hidden form element is attached to the form, a user can now submit the form. You should see the `user["avatar_url"]` in the parameters of the logs when this works correctly:

```
Started POST "/users" for 127.0.0.1 at 2014-05-19 17:47:01 -0500
Processing by UsersController#create as HTML
  Parameters: {"utf8"=>"✓", "user"=>{"avatar_url"=>"//my-development.s3.amazonaws.com/uploads/220f5
```

Now when you wish to show the url and you have an `@user` object you can render the image:

```
<%= image_tag @user.avatar_url %>
```

Debugging

While implementing direct to S3 uploads, it is very helpful to keep the JavaScript console open, add a `console.log` statement to increase viability of callbacks in development. They should be removed before deploying to production. If there are any exceptions in the console, they must be resolved before your code will work. If the file uploads are working but the URL is not saving, check your logs and ensure that the code in the `create` action of the UsersController is properly saving and permitting the `avatar_url` column.

Extending

This example represents one way to do implement direct to S3 uploads. Your application may have different requirements and demands. You should take this example as a template and then modify and extend it to achieve the behavior you want.

For example you may want to enable client side photo cropping and editing, maybe you want different progress bars, drag and drop interfaces or soemthing else. You can extend the code here by referencing the jQuery-File-Upload documentation.

If you want to impose limits on the types of files you can accept or maybe you want user images to be expired after a certain time please reference AWS presigned post documentation.