Example: GridFS Rails File Server

In this example, we will create a bare bones, web-based file server that we can upload, store, get, and download contents from. The application will be backed by GridFS. Access to GridFS will be done through a model class implemented to work with the Rails scaffold. Much of the model will be assembled and tested using `rails console` prior to addig the controller and view.

Highlights

index page

show page

1. Files are uploaded using the browser and the `f.file_field` option (`app/views/grid_fs_files/_form.html.erb`).

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

2. The object type supplied for `contents` by Rails is an `ActionDispatch::Http::UploadedFile`.

   ```
   #<ActionDispatch::Http::UploadedFile:0x00000005597018>
   ```

3. The `UploadedFile` can be read directly into the `Grid::File` with the hash of file description properties. The root level keys in the hash are standard within GridFS. The keys below `metadata` are user-defined. Note that GridFS uses `snake_case` keys in the `Grid::File` object but uses `camelCase` in the hash info interface we will see later.

   ```
   description = {:filename=>@filename,
                  :content_type=>@contentType,
                  :metadata => {:author => @author, :topic => @topic}}
   grid_file = Mongo::Grid::File.new(@contents.read, description )
   id=self.class.mongo_client.database.fs.insert_one(grid_file)
   @id=id.to_s
   ```

4. The file is accessed by a URI that can be defined using an `img` tag (`app/views/grid_fs_files/show.html.erb`).

   ```
   <p>
     <strong>Contents:</strong>
     <img height="500px" width="650px" src= <%= contents_path("#{@grid_fs_file.id}")%>/>
   </p>
   ```

5. This URI is defined using a `GET` in the `routes.rb` file mapped to the controller `contents` action. (`config/routes.rb`). By defining this as `contents` resource, we get the helper method `contents_path` used above.

   ```
   get ’/grid_fs_files/contents/:id/’, to: ’grid_fs_files#contents’, as: ’contents’
   ```

6. The controller method accesses the data from the contents attribute and sends this back to the web caller with a few HTTP properties. For example, by supplying `filename`, the file will default to the name provided in the model.

   ```
   class GridFsFilesController < ApplicationController
     before_action :set_grid_fs_file, only: [:show, :edit, :update, :destroy, :contents]

     def contents
   ```

```
    send_data @grid_fs_file.contents,
              {filename: @grid_fs_file.filename,
               type: @grid_fs_file.contentType,
               disposition: 'inline'}
  end
```

7. The getter for `contents` is provided by a custom implementation that locates the GridFS content by `id` and returns a buffer with the data from each chunk.

def contents f=self.class.mongo_client.database.fs.find_one({:*id=>BSON::ObjectId.from*string(@id)}) buffer = “” f.chunks.reduce([]) { |x,chunk| buffer << chunk.data.data } return buffer end

Infrastructure

This section will lightly show the steps required to get the supporting part of the demo in place.

Create The Rails Application and Setup MongoDB Connection

1. Create the application

   ```
   $ rails new gridfsfiles
   $ cd gridfsfiles
   ```

2. Add gems to `Gemfile`

   Mongoid is required for the connection management. It will automatically bring in mongo (MongDB Ruby Driver) – which is still the focus of this lesson. However, you can specify both using the following.

   ```
   gem 'mongo', '~> 2.1.0'
   gem 'mongoid', '~> 5.0.0'
   ```

   This brought in the following versions when the example was written.

   ```
   $bundle
   Using mongo 2.1.2
   Using mongoid 5.0.1
   ```

3. Create the Mongoid Connection Configuration File

   ```
   $ rails g mongoid:config
         create  config/mongoid.yml
   ```

   This creates a configuration with usable defaults for the `development` (and `test`) profile.

   ```
   $ egrep -v '\#|^$' config/mongoid.yml
   development:
     clients:
       default:
         database: gridfsfiles_development
         hosts:
           - localhost:27017
         options:
     options:
     ...
   ```

4. Load the Mongoid Configuration File into Rails Application

   `config/application.rb`

```
module Gridfsfiles
  class Application < Rails::Application
    ...
    #bootstraps mongoid within applications -- like rails console
    Mongoid.load!('./config/mongoid.yml')
    ...
  end
end
```

Create GridFS Model Class for File Content

Create GridFS Model Class

1. Create a GridFsFile model class to implement interactions between our application and GridFS. Start with the core properties required by the Rails scaffold like we saw with the `zips` application.

   app/models/grid_fs_file.rb

   ```
   class GridFsFile
     include ActiveModel::Model
     attr_accessor :id

     def persisted?
       !@id.nil?
     end
     def created_at
       nil
     end
     def updated_at
       nil
     end
   end
   ```

2. Create some file attributes to track for the contents. Start by locating properties we get from GridFS. Know that the metadata property is user-defined.

   ```
   > pp c.database.fs.find.first
   {"_id"=>BSON::ObjectId('5642f149e301d09ce9000009'),
    "chunkSize"=>261120,
    "uploadDate"=>2015-11-11 07:41:50 UTC,
    "contentType"=>"image/jpeg",
    "filename"=>"myfile.jpg",
    "metadata"=>{"author"=>"kiran", "topic"=>"nice spot"},
    "length"=>307797,
    "md5"=>"3468ca1c23cc13ac6af493c4642cc72a"}
   ```

   Define the above GridFS properties as attributes of the model class. Lets use the same camel case as GridFS to keep things consistent between Rails and GridFS hashes as possible. Add in metadata properties of `author` and `topic` as an example of tracking additional data. We also have refined the attributes into read/write, read-only, and write-only accesses. The GridFS descriptive information – including the metadata we define – is updatable at any time. `id`, `chunkSize`, `length`, and `md5` are all internally generated so we just define getters for those. `contents` is special. We will define a custom getter for it shortly.

   ```
   class GridFsFile
     include ActiveModel::Model
     attr_accessor :contentType, :filename, :author, :topic
     attr_writer :contents
     attr_reader :id, :uploadDate, :chunkSize, :length, :md5
   ```

Define an `initialize` method from a hash that can process hash keys produced by GridFS and Rails. Remember that MongoDB uses ’:_id`for its primary key and Rails scaffold expects to use`:id’. Note too that since our custom `author` and `topic` fields are scoped below the GridFS `metadata` property, we can leverage the same `id` parameter test to determine whether we are representing this internally or externally.

```ruby
def initialize(params={})
  if params[:_id]  #hash came from GridFS
    @id=params[:_id].to_s
    @author=params[:metadata].nil? ? nil : params[:metadata][:author]
    @topic=params[:metadata].nil? ? nil : params[:metadata][:topic]
  else              #assume hash came from Rails
    @id=params[:id]
    @author=params[:author]
    @topic=params[:topic]
  end
  @chunkSize=params[:chunkSize]
  @uploadDate=params[:uploadDate]
  @contentType=params[:contentType]
  @filename=params[:filename]
  @length=params[:length]
  @md5=params[:md5]
  @contents=params[:contents]
end
```

Add MongoDB Connection

```
‘‘‘ruby
  def self.mongo_client
    @@db ||= Mongoid::Clients.default
  end
‘‘‘
```

Add Save Capability

1. Add an instance method to save the current instance.

    - the file data will from from an IO object stored in the `contents` attribute
    - an optional description is populate with file info, includig user-defined metadata
    - the `Grid::File` is inserted into GridFS and a primary key is returned
    - Note that the optional description takes a snake_case `content_type`, rather than the camelCase used in the upcoming find results.

```ruby
def save
  description = {}
  description[:filename]=@filename         if !@filename.nil?
  description[:content_type]=@contentType  if !@contentType.nil?
  if @author || @topic
    description[:metadata] = {}
    description[:metadata][:author]=@author  if !@author.nil?
    description[:metadata][:topic]=@topic    if !@topic.nil?
  end

  if @contents
    grid_file = Mongo::Grid::File.new(@contents.read, description )
    id=self.class.mongo_client.database.fs.insert_one(grid_file)
    @id=id.to_s
    end
  end
```

2. Take the new method for a test drive.

   Launch the `rails console`

   ```
   $ rails c
   Loading development environment (Rails 4.2.4)
   ```

   Create an (File) IO object with the contents of a file

   ```
   > os_file=File.open("./db/image1.jpg")
    => #<File:./db/image1.jpg>
   ```

   New up a model instance, passing in the IO object as the `contents` and other user-provided fields

   ```
   > f=GridFsFile.new(:author => "kiran", :topic => "cool place", :contentType=>"image/jpeg",
                      :filename=>"town1.jpg", :contents=>os_file)
    => #<GridFsFile:0x00000005a836d0 @id=nil, @author="kiran", @topic="cool place",
        @chunkSize=nil, @uploadDate=nil, @contentType="image/jpeg", @filename="town1.jpg",
        @length=nil, @md5=nil, @contents=#<File:./db/image1.jpg>>
   ```

   Save the file info and contents to GridFS

   ```
   > f.save
    => "56458c18e301d0d09c000004"
   ```

Add a Find to Return a Single Model Instance

1. Declare a set of helper methods (one a class method and the other an instance method) to convert the string form of a BSON::ObjectId back to object form and return that in a query hash since we will be making use of the `id` mostly in that manner. The instance method operates on the `@id` attribute. The class method operates on the `id` passed in as an argument.

   ```
   def self.id_criteria id
     {_id:BSON::ObjectId.from_string(id)}
   end
   def id_criteria
     self.class.id_criteria @id
   end
   ```

2. Declare a class method to use the `fs.find` method to locate the file info in GridFS. Use the `id_criteria` helper method we just created to build a query hash expression for the primary key. Note that we are not yet querying for the file object. That will not occur until we need the contents.

   ```
   def self.find id
     f=mongo_client.database.fs.find(id_criteria(id)).first
     return f.nil? ? nil : GridFsFile.new(f)
   end
   ```

3. Take the new method for a test drive.

   Reload the new class implementation into `rails console`.

   ```
   > reload!
   ```

   If you do not remember your file ID, use the `mongo_client` and the `find.first` command to get a sample file.

   ```
   > GridFsFile.mongo_client.database.fs.find.first[:_id].to_s
    => "56458c18e301d0d09c000004"
   ```

Get the file info from GridFS and wrap in a Model instance.

```
> f=GridFsFile.find "56458c18e301d0d09c000004"
 => #<GridFsFile:0x000000059bf870 @id="56458c18e301d0d09c000004",
        @author="kiran",
        @topic="cool place",
        @chunkSize=261120,
        @uploadDate=2015-11-13 07:07:04 UTC,
        @contentType="image/jpeg",
        @filename="town1.jpg",
        @length=307797,
        @md5="3468ca1c23cc13ac6af493c4642cc72a",
        @contents=nil>

> f.filename
 => "town1.jpg"
> f.length
 => 307797
```

Get Data Contents from GridFS

1. Add an instance method to implement a custom getter for the `contents` attribute. This method will use `fs.find_one` to locate the file object matching the criteria generated by the `id_criteria` helper method and the instance's primary key. The array of chunks is reduced to a single buffer returned to the caller.

```
def contents
  Rails.logger.debug {"getting gridfs content #{@id}"}
  f=self.class.mongo_client.database.fs.find_one(id_criteria)
  if f
    buffer = ""
    f.chunks.reduce([]) do |x,chunk|
        buffer << chunk.data.data
    end
    return buffer
  end
end
```

2. Take the new method for a test drive.

   With a handle to the file, we can obtain the bytes of the file data content and simply return the size of the buffer used.

```
> reload
> GridFsFile.mongo_client.database.fs.find.first[:_id].to_s
 => "56458c18e301d0d09c000004"
> f=GridFsFile.find "56458c18e301d0d09c000004"
> f.contents.length
 => 319998
```

Add a Find of All Model Instances

1. Create an instance method to return a collection of model instances that represent the files in GridFS. Note that this is just the file information and not the file data content.

```
  def self.all
    files=[]
    mongo_client.database.fs.find.each do |r|
      files << GridFsFile.new(r)
    end
    return files
  end
```

2. Take the new method for a test drive.

   ```
   > reload
   > pp GridFsFile.all.to_a
   [#<GridFsFile:0x000000039beda0
     @author="kiran",
     @chunkSize=261120,
     @contentType="image/jpeg",
     @contents=nil,
     @filename="town1.jpg",
     @id="56458c18e301d0d09c000004",
     @length=307797,
     @md5="3468ca1c23cc13ac6af493c4642cc72a",
     @topic="cool place",
     @uploadDate=2015-11-13 07:07:04 UTC>]
   ```

Add an Update of the Model Instance

1. Just leave this empty for now. We will not be updating files.

   ```
   def update params
     #TODO
   end
   ```

Add a Delete of the Model Instance

1. Add an instance method to destroy the file associated with the instance's primary key. We use the `fs.find` method and our helper `id_criteria` to locate and delete the file info and contents from GridFS.

   ```
   def destroy
     self.class.mongo_client.database.fs.find(id_criteria).delete_one
   end
   ```

2. Take the new method for a test drive.

   ```
   > reload!
   > f=GridFsFile.find "56458c18e301d0d09c000004"
   > f.destroy
    => #<Mongo::Operation::Result:50114800 documents=[{"ok"=>1, "n"=>1}]>
   > pp GridFsFile.all.to_a
    => []
   ```

Add Rails Scaffold

1. Generate a controller and view that can process all attributes. Note that we are violating the Rails standard by using the camelCase attribute names provided by GridFS here to save some field making (i.e., mapping content_type <-> contentType). It may be worth it to cut down on transation code in a demo like this, but add the mapping in a real application. Note also that we are declaring uploadDate as a string. That is because this field is internally generated by GridFS at upload time and we will treat it as a read-only attribute. The default text display of a date looks much better than the default date widget added by Rails when this is a read-only field.

```
$ rails g scaffold_controller GridFsFile filename contentType author topic \
uploadDate length:integer chunkSize:integer md5 contents
```

2. Update the `routes.rb` to add our resource and make it the root URI for the application.

```
Rails.application.routes.draw do
  root to: 'grid_fs_files#index'
  resources :grid_fs_files
```

```
$ rake routes
           Prefix Verb   URI Pattern                            Controller#Action
             root GET    /                                      grid_fs_files#index
    grid_fs_files GET    /grid_fs_files(.:format)               grid_fs_files#index
                  POST   /grid_fs_files(.:format)               grid_fs_files#create
 new_grid_fs_file GET    /grid_fs_files/new(.:format)           grid_fs_files#new
edit_grid_fs_file GET    /grid_fs_files/:id/edit(.:format)      grid_fs_files#edit
     grid_fs_file GET    /grid_fs_files/:id(.:format)           grid_fs_files#show
                  PATCH  /grid_fs_files/:id(.:format)           grid_fs_files#update
                  PUT    /grid_fs_files/:id(.:format)           grid_fs_files#update
                  DELETE /grid_fs_files/:id(.:format)           grid_fs_files#destroy
```

Serve Up File Contents

1. Add an additional route to our controller for data content

```
  get '/grid_fs_files/contents/:id/', to: 'grid_fs_files#contents', as: 'contents'
```

```
$ rake routes
           Prefix Verb   URI Pattern                            Controller#Action
         contents GET    /grid_fs_files/contents/:id(.:format) grid_fs_files#contents
```

2. Implment the `contents` action in terms of getting the model instance associated with the `id` and returning the image contents.

   Add the contents method to the `before_action`

```
class GridFsFilesController < ApplicationController
  before_action :set_grid_fs_file, only: [:show, :edit, :update, :destroy, :contents]
```

   Add the contents method that sends the data from the model.contents with several HTTP content properties.

```
def contents
  send_data @grid_fs_file.contents,
            {filename: @grid_fs_file.filename, type: @grid_fs_file.contentType, disposition: 'inline'}
end
```

3. Start the server and take the new controller method for a test drive.

   Start the server

```
$ rails s
```

   Populate GridFS with an image from `rails console`

```
> os_file=File.open("./db/image1.jpg")
 => #<File:./db/image1.jpg>
> f=GridFsFile.new(:author => "kiran", :topic => "cool place", :contentType=>"image/jpeg",
                   :filename=>"town1.jpg", :contents=>os_file)
> f.save
 => "5645a2b3e301d0d09c000017"
```

   Access the image from the following URL, replacing the BSON::ObjectId with whatever image you wish to access.

```
http://localhost:3000/grid_fs_files/contents/5645a2b3e301d0d09c000017
```

Add Upload and UI Display

Update View References to `content`

1. Update the fields displayed on the HTML index page (`app/views/grid_fs_files/index.html.erb`) to include a thumbnail version of the image and remove some of the larger fields (e.g., md5) that are available on the show page.

   `html <th>Contents</th> <th>Filename</th> <th>Contenttype</th> <th>Author</th> <th>Topic</th> <th>Uploaddate</th> <th>Length</th>` Include a "thumbnail-sized" version of the contents on each line using the `img` tag.

   `html <td><img height="100px" width="130px" src= <%= contents_path("#{grid_fs_file.id}")%>/></td> <td><%= grid_fs_file.filename %></td> <td><%= grid_fs_file.contentType %></td> <td><%= grid_fs_file.author %></td> <td><%= grid_fs_file.topic %></td> <td><%= grid_fs_file.uploadDate %></td> <td><%= grid_fs_file.length %></td>`

2. Remove the `contents` from the JSON view. `app/views/grid_fs_files/index.json.jbuilder`

   `ruby #TODO: fix this json.extract!  grid_fs_file, :id, :filename, :contentType, :author, :topic, :uploadDate, :length, :chunkSize, :md5`

3. Update the `app/views/grid_fs_files/_form.html.erb` from a `text_field` to a `file_field`

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

4. Update the show page to display the image from our contents URI with an `img` tag in `app/views/grid_fs_files/show.html.e`

   ```
   <p>
     <strong>Contents:</strong>
     <img height="1000px" width="1300px" src= <%= contents_path("#{@grid_fs_file.id}")%>/>
   </p>
   ```

5. Mark the GridFS-managed fields readonly.

   ```
   <div class="field">
     <%= f.label :uploadDate %><br>
     <%= f.text_field :uploadDate, :readonly => true %>
   </div>
   <div class="field">
     <%= f.label :length %><br>
     <%= f.number_field :length, :readonly => true %>
   </div>
   <div class="field">
     <%= f.label :chunkSize %><br>
     <%= f.number_field :chunkSize, :readonly => true %>
   </div>
   <div class="field">
     <%= f.label :md5 %><br>
     <%= f.text_field :md5, :readonly => true %>
   </div>
   ```

Take for a Test Drive

1. Navigate to root URL

   ```
   http://localhost:3000/
   ```

2. Click `New Grid_fs_file`
3. File in the following fields. Do not bother typing in the other fields that are supplied by GridFS.

   - Filename
   - Contenttype
   - Author
   - Topic

4. Select `Choose File` and select image
5. Click `Create Grid_fs_file`
6. Click `Back` to go back to index.

Heroku Deployment

This deployment assumes that you have already deployed the `Zips` and `GeoZips` applications and will quickly go thru the steps taken to reach Heroku deployment.

1. Register your application with Heroku by changing to the directory with a git repository and invoking `heroku apps:create (appname)`.

   Note that your application must be in the root directory of the development folder hosting the git repository.

   ```
   $ cd fullstack-course3-module2-gridfsfiles
   $ heroku apps:create appname
   Creating appname... done, stack is cedar-14
   https://appname.herokuapp.com/ | https://git.heroku.com/appname.git
   Git remote heroku added
   ```

   This will add an additional remote to your git repository.

   ```
   $ git remote --verbose
   heroku  https://git.heroku.com/appname.git (fetch)
   heroku  https://git.heroku.com/appname.git (push)
   ...
   ```

2. Verify the Gemfile is setup to support ActiveRecord when deployed to Heroku. This is required because we have not removed it from our application.

   ```
   # Use sqlite3 as the database for Active Record
   gem 'sqlite3', group: :development
   ...
   group :production do
     #use postgres on heroku
     gem 'pg'
     gem 'rails_12factor'
   end
   ```

3. Create a new MongoDB database and database user on MongoLabs
4. Add a `MONGOLAB_URI` environment variable to the environment to define the databaase connection when deployed to Heroku. `dbhost` is both host and port# concatenated together, separated by a ":" (host:port) in this example.

   ```
   $ heroku config:add MONGOLAB_URI=mongodb://dbuser:dbpass@dbhost/dbname
   ```

5. Verify the `config/mongoid.yml` has a `production` profile to accept the newly added environment variable.

```
production:
  clients:
    default:
      uri: <%= ENV['MONGOLAB_URI'] %>
      options:
        connect_timeout: 15
```

6. Run bundle and commit any changes.

7. Deploy application

   ```
   $ git push heroku master
   ```

Access Application

1. Access URL

   ```
   http://appname.herokuapp.com/
   ```

2. Access “New Grid gs file” to upload a new image into GridFS.