Unit 5: Rails App Development

[Lesson 5.1: File Uploads With Paperclip](#h.x8nxadm737lb)

[Gem of the Day](#h.5aki30pwi6vr)

[Git Branch](#h.z2s5gt941ibb)

[Review Setting Up Paperclip](#h.fpjzb9pjnffw)

[Optimizing Paperclip](#h.pfy5awbu30vg)

[Center and Crop](#h.dlunw67o2uj6)

[Limiting File Size](#h.o537pg9lc4j8)

[Git Branch and Merge](#h.8w2vjzdauhwx)

[Stepping Away From Default Rails Views](#h.y0k896a67tqr)

[Panels on Dog Index](#h.mpv8budk6faf)

[Default Image](#h.geztsopwsah3)

[Green Panels for Dogs in Daycare](#h.lz4gbkox1fdi)

[Classroom Challenge: More Views](#h.e3sbz86m9hg6)

[Lesson 5.2: HTTP Methods, Routes, and Parameters](#h.lq4t4jg79j5a)

[Gem of the Day](#h.e0dulggh4jsj)

[Objectives](#h.l9258l6pr8jv)

[Rails Routing Review](#h.9ospeu9f7xs8)

[CRUD Review and HTTP Verbs](#h.nh5o16qvp6gl)

[Rails Routes](#h.1wd1l4t4can9)

[Root](#h.e6q74ntzea1x)

[Resources Default Routes](#h.802rzkj8d4cl)

[Parameters in Rails](#h.3liwf5zz2n7)

[Via Routes](#h.53trncoprzbf)

[Via GET](#h.lyo9euv6txt9)

[Via POST/PUT](#h.qd4tgkkoauzy)

[More on GET vs POST](#h.apahe6kcs767)

[Using Parameters in Our App](#h.ybcvno3xki7u)

[Simple Search Form using GET](#h.ehuwjcua51xt)

[Form For Check In/Out using PATCH](#h.xcvro79cwl64)

[Filter By Buttons](#h.v2zip33toxk4)

[Lesson 5.3: User Management Using Devise](#h.hjmfj650bt3f)

[Gem of the Day](#h.1hvyn7rlcqn4)

[Lesson Plan](#h.p3hpdk7ph7ps)

[Homework](#h.wl7shtaeihi8)

# Lesson 5.1: File Uploads With Paperclip

## Gem of the Day

PlaceGant! <https://github.com/AngieGreen/placegant>

## Git Branch

(feature branches - create a new branch for paperclip)

## Review Setting Up Paperclip

Paperclip is an awesome gem that handles file uploads and manipulates images. First, you must have ImageMagick installed. See the Lesson 4.3 homework for more info.

Take a look at the Paperclip gem documentation. The readme file is actually quite detailed. It can be overwhelming, but when you take it step-by-step it’s not too bad.

1. First, we need to let Paperclip have access to ImageMagick. See [here](https://github.com/thoughtbot/paperclip#image-processor). First, run which convert. Grab that path and put it in config/environments/development.rb:  
    # Telling Paperclip gem where to find ImageMagick  
    Paperclip.options[:command\_path] = "/usr/local/bin/"
2. Add gem ‘paperclip’, bundle, commit.  
    gem "paperclip", "~> 4.2"
3. Now we can generate our Paperclip stuff. Paperclip conveniently comes with an automatic migration generator to add an avatar to a table. Read more [here](https://github.com/thoughtbot/paperclip#migrations). Run:  
    rails g paperclip dog avatar
4. Check the migration then run rake db:migrate. Add :avatar to the Dog controller whitelist.
5. Now, let’s add some validations to Dog for avatar for safety.  
    has\_attached\_file :avatar, :styles => { :medium => "300x300>", :thumb => "100x100>" }, :default\_url => "/images/:style/missing.png"  
    validates\_attachment\_content\_type :avatar, :content\_type => /\Aimage\/.\*\Z/
6. Now we can add the file upload to our dog form:  
    <div class="field">  
    <%= f.label :avatar %><br>  
    <%= f.file\_field :avatar %>  
    </div>
7. And to show:

<p>

<%= image\_tag @dog.avatar.url %>

<%= image\_tag @dog.avatar.url(:medium) %>

<%= image\_tag @dog.avatar.url(:thumb) %>

</p>

Run your rails server, upload a file for a dog, and test everything out. So cool! If everything worked, **commit your changes**. Work through any errors during office hours and over the weekend.

## Optimizing Paperclip

### Center and Crop

So far, Paperclip is really impressive. But, it would be even better if we could center and crop our photos down to a square. Go to your dogs model and edit it to look like this:

has\_attached\_file :avatar,

:styles => {

:medium => "300x300#",

:thumb => "100x100#"

},

:default\_url => "/images/:style/missing.png"

What did we do here? Default behavior is to resize the image and maintain aspect ratio . Some commonly used options are:

* **trailing #:** thumbnail will be centrally cropped, ensuring the requested dimensions.
* **trailing >:** thumbnail will only be modified if it is currently larger requested dimensions. (e.g the :small thumb for a 120×80 original image will be unchanged)

### Limiting File Size

We can also limit the file size by adding another validation:

validates\_attachment\_size :avatar, :less\_than => 3.megabytes,

:unless => Proc.new {|m| m[:avatar\_file\_name].blank?}

Once you’re satisfied with your changes, commit them.

## Git Branch and Merge

Now that we know paperclip is working, let’s merge our changes back into master.

As a review, we created a **branch** by typing “git branch branchname”. To see a full list of branches, we typed “git branch” alone. To start working on a different branch, first make sure you don’t have changes that need to be committed for the current branch, then type “git checkout branchname”.

Once you are finished developing on your branch and are ready to **merge** the changes back into master, first make sure you have committed your final changes, then:

git checkout master

git merge branchname --no-ff

The --no-ff flag stands for “no fast-forward” and will preserve our commit history from the old branch.

Once we are done merging, if we are finished with a branch, we typically delete it. You can delete a branch with:

git branch -d branchname

## Stepping Away From Default Rails Views

### Panels on Dog Index

We now have a really cool tool with the dog photos, but we’re not really using it to its full potential. These index pages are nice but kind of really ugly and not useful. Let’s make them pretty by getting away from the table format, and using Bootstrap divs and [panels](http://getbootstrap.com/components/#panels) instead!

Read about [panels](http://getbootstrap.com/components/#panels), and then let’s update our Dogs index.html.erb:

<h1>Our Doggies</h1>

<div class="row">

<% @dogs.each do |dog| %>

<div class="col-md-3">

<div class="panel panel-default">

<div class="panel-heading">

<h3 class="panel-title"><%= dog.name %> <%= dog.owner.last\_name %></h3>

</div>

<div class="panel-body">

Panel content

</div>

<div class="panel-footer">

In daycare? <%= dog.in\_daycare %><br>

<%= link\_to content\_tag(:i, nil, class: "fa fa-eye"), dog %>

<%= link\_to content\_tag(:i, nil, class: "fa fa-pencil"), edit\_dog\_path(dog) %>

<%= link\_to content\_tag(:i, nil, class: "fa fa-trash-o"), dog, method: :delete, data: { confirm: 'Are you sure?' } %>

</div>

</div>

</div>

<% end %>

</div>

Refresh your page. That’s so cool! Now let’s replace the panel content with our photos:

<%= image\_tag dog.avatar.url(:medium), class: "img-responsive" %>

### Default Image

So cool! But, it doesn’t look pretty if an image hasn’t yet been uploaded for a dog.

1. Find an image to use as a default image, and upload it so that Paperclip will generate all three versions of it for us. You can find it in the public folder (keep digging in all the subfolders).
2. Copy all three versions to your assets/images folder. Rename them in that folder to missing\_original, missing\_medium, and missing\_thumb, with the correct file type extension.
3. Remember in the model when we added validation? Take a look at that code. Edit the default\_url to this, except adjust the file type extension if your images were not jpgs:  
   :default\_url => "missing\_:style.jpg"
4. Commit your changes!

### Green Panels for Dogs in Daycare

We can even change the panel color based on whether a dog is in daycare using erb:

<div class="panel panel-default <%= 'panel-success' if dog.in\_daycare %>">

Comment out the In Daycare (both html and erb). Commit your changes!

### Classroom Challenge: More Views

On the Owner show view, show a list of dogs but as panels instead of a bulleted list. Make sure you have at least one link to go to that dog’s show page.

Make the Dog show view prettier. Pick one picture to show and reformat the view. Make the owner a link back to that owner’s show view.

# Lesson 5.2: HTTP Methods, Routes, and Parameters

<http://www.theodinproject.com/ruby-on-rails/routing>

<http://guides.rubyonrails.org/routing.html>

## Gem of the Day

Carrierwave <https://github.com/carrierwaveuploader/carrierwave>

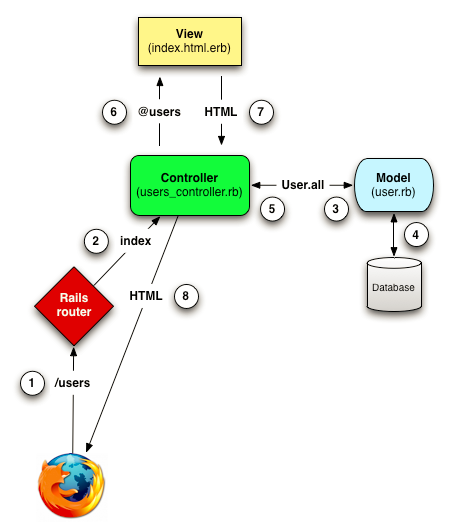
## Objectives

Be able to answer:

* What is the "Root" route?
* What are the seven routes for a resource?
* Which routes share the same URL but use different verbs?
* How do you specify an ID or other variable in a route?
* How can you easily write all seven routes in Rails?

## Rails Routing Review

Remember when we walked through our introduction to Rails using an example user interaction? (draw this on board again)



As you can see, the **router** is the **doorman** of your application. When an **HTTP request** arrives from the user's browser, it needs to know which **controller action** (method) should be run. Should we display the "new user" webpage? Should we edit an existing user with whatever data got sent along?

The Router is basically just a **matching service**. It looks at the **HTTP verb** (GET, POST, PUT, DELETE) and the **URL** that it being requested and matches it with the appropriate **controller action** to run. It's a pretty simple function but an essential one.

The other cool thing that happens is that Rails grabs all the **parameters** that came with a request and makes them available for you in a special hash called **params** that you can later use in your controller. It's used for things like form submissions so that you later can use that form data to create or modify objects.

You can see how your routes are set up in config/routes.rb. You can also run “rake routes” to get the full list of routes available in your app.

## CRUD Review and HTTP Verbs

Who remembers what CRUD stands for? What about some of the ways we used CRUD in ActiveRecord? (write on board)

Well, we also use the concept of CRUD to explain HTTP verbs or methods.

|  |  |  |
| --- | --- | --- |
| **CRUD** | **ActiveRecord** | **HTTP Verbs** |
| Create | create/new | post |
| Read | find, where | get |
| Update | save | patch/put |
| Delete | destroy | delete |

## Rails Routes

### Root

One of the most important routes is the root route. It also happens to be one of the simplest. In our routes.rb file, find the root route:

root 'pages#index'

This simple line states that whenever someone goes to the home domain, Rails should **GET** the **pages** controller and the **index** action, or method.

### Resources Default Routes

When it comes to models that are scaffolded, you can see that Rails just takes a shortcut and lists:

resources :dogs

What does Rails really do? Well, there are seven main types of **actions**, or **methods**, that you can (and should) do to a "resource", or something with its own database model. Let’s open the controller for one of our scaffolds and try to figure out what they all are (have them give all 7 methods, then try to guess the HTTP verbs).

|  |  |  |
| --- | --- | --- |
| **Action/ Method** | **HTTP Verb** | **Description** |
| index | GET | GET all the posts (aka "**index**" the posts) |
| show | GET | GET just one specific post (aka "**show**" that post) |
| new | GET | GET the page that lets you create a new post (aka view the "**new**" post page) |
| create | POST | POST the data you just filled out for a new post back to the server so it can create that post (aka "**create**" the post) |
| edit | GET | GET the page that lets you edit an existing post (aka view the "**edit**" post page) |
| update | PUT | PUT the data you just filled out to edit the post back to the server so it can actually perform the update (aka "**update**" the post) |
| destroy | DELETE | DELETE one specific post by sending a delete request to the server (aka "**destroy**" the post) |

If Rails wrote these out the long way, they would be:

get "/dogs" => "dogs#index"  
get "/dogs/:id" => "dogs#show"  
get "/dogs/new" => "dogs#new"  
post "/dogs" => "dogs#create" # usually a submitted form  
get "/dogs/:id/edit" => "dogs#edit"  
put "/dogs/:id" => "dogs#update" # usually a submitted form  
delete "/dogs/:id" => "dogs#destroy"

Each of these routes is basically a **Ruby method** that matches that particular **URL** and **HTTP verb** with the correct **controller action**. Notice that several of those routes submit to the SAME URL... they just use different HTTP verbs, so Rails can send them to a different controller action.

## Parameters in Rails

### Via Routes

Another thing to notice is that for show, edit, update, and delete, the "id" field is prepended by a colon... that just tells Rails to save whatever is in place of :id as the id in the **params** hash. This is the first example of how Rails uses **params** - via the route.

In other words, when the user sends a GET request to /dogs/3, Rails puts “3” in the params hash for :id, which can be shown like this:

params[:id] = 3

Then, once it goes to the dogs controller, it matches the id from the params hash to any dog with an id of 3 and loads the page with the full details of that dog.

### Via GET

The params can also be encoded in the url via the GET request. For example, if a user's browser requested:

<http://www.example.com/?foo=1&boo=octopus>

then params[:foo] would be "1" and params[:boo] would be "octopus".

In HTTP/HTML, the params are really just a **series of key-value pairs** where the key and the value are strings, but Ruby on Rails has a special syntax for making the params be a hash with hashes inside. For example, if the user's browser requested:

<http://www.example.com/?vote[item_id]=1&vote[user_id]=2>

then params[:vote] would be a hash, params[:vote][:item\_id] would be "1" and params[:vote][:user\_id] would be "2".

### Via POST/PUT

The third way Rails uses params is through **POST** requests. When you submit a form, the control is thrown back to the application. But how do you get the values you have submitted to the form? params is how. Let’s take a closer look at GET vs POST...

### More on GET vs POST

Two commonly used methods for a request-response between a client and server are: GET and POST.

* GET - Requests data from a specified resource
* POST - Submits data to be processed to a specified resource

**Query or parameter strings** (name/value pairs) are sent in the URL of a **GET** request:

/test/demo\_form.asp?name1=value1&name2=value2

However, for **POST** requests, **query or parameter strings** are sent in the HTTP message body:

POST /test/demo\_form.asp HTTP/1.1

Host: w3schools.com

name1=value1&name2=value2

Some other notes on GET requests:

* GET requests can be cached
* GET requests remain in the browser history
* GET requests can be bookmarked
* GET requests should never be used when dealing with sensitive data
* GET requests have length restrictions
* GET requests should be used only to retrieve data

Some other notes on POST requests:

* POST requests are never cached
* POST requests do not remain in the browser history
* POST requests cannot be bookmarked
* POST requests have no restrictions on data length

## Using Parameters in Our App

### Simple Search Form using GET

Let’s start our foray into parameters with a simple search form. Always use "GET" as the method for search forms. This allows users to bookmark a specific search and get back to it. More generally Rails encourages you to use the right HTTP verb for an action. (not POSTing data to the model when searching)

We would like to find all the dogs with a particular name. Let’s add a search box to the navbar so that employees have easy access to it no matter where they are.

First, let’s **add our search form to our navbar** right under the closing </ul> for our links (explain each part):

<%= form\_tag("/dogs", method: "get", role: "search", class: "navbar-form navbar-left") do %>

<div class="form-group">

<%= text\_field\_tag(:search, nil, placeholder: "Fido", class: "form-control input-sm") %>

</div>

<%= submit\_tag("Search", class: "btn btn-default btn-sm") %>

<% end %>

That’s cool, but it doesn’t actually search for any dogs yet. Where do you think we need to add some code for the search? In the **dogs controller, index action**. Modify the dogs controller like so:

def index

if params[:search]

@dogs = Dog.where("name LIKE '%#{params[:search]}%'")

if @dogs.size.zero?

flash[:notice] = "No result found"

@dogs = Dog.all

end

else

@dogs = Dog.all

end

end

Try searching for a few dogs. So cool! Now, commit your changes.

Classroom Challenge: Add a search box on the owner index page that searches for an owner by last name.

### Form For Check In/Out using PATCH

On our dog index, let’s add a very small form that just let’s us check in or check out a dog. For now, we will use a checkbox paired with a submit button. Once we learn more about JavaScript, we can make it much better.

We can use the \_form.html.erb as a guide for building our mini-form in dogs index.html.erb:

<%= form\_for(dog, url: dog\_path(dog), :html => {class: "form-inline"}) do |f| %>

<%= f.check\_box :in\_daycare, class: "checkbox-inline" %> <%= f.label :in\_daycare %>

<%= f.submit "Check In/Out", class: "btn btn-success btn-xs" %>

<% end %>

Let’s also make it easier for the staff by redirecting back to the index page upon update:

def update

respond\_to do |format|

if @dog.update(dog\_params)

format.html { redirect\_to dogs\_path, notice: 'Dog was successfully updated.' }

format.json { render :show, status: :ok, location: @dog }

else

format.html { render :edit }

format.json { render json: @dog.errors, status: :unprocessable\_entity }

end

end

end

That was fun, but we can make it even easier by eliminating the checkbox and making the buttons vary depending on the current value of in\_daycare. Thinking theoretically, how would you do that? (get them to answer before starting to code)

<%= form\_for(dog, url: dog\_path(dog), :html => {class: "form-inline"}) do |f| %>

<% if dog.in\_daycare %>

<%= f.hidden\_field :in\_daycare, :value => false %>

<%= f.submit "Check Out", class: "btn btn-warning btn-xs" %>

<% else %>

<%= f.hidden\_field :in\_daycare, :value => true %>

<%= f.submit "Check In", class: "btn btn-success btn-xs" %>

<% end %>

<% end %>

Commit your changes!

### Filter By Buttons

If time allows, let’s add some filter by owner buttons to our dog index. They aren’t too practical here, so let’s do this in a feature branch so we can scrap it if we don’t like it. This would be a great feature if you were tracking which rooms dogs were in. They are good practice for passing parameters.

1. Create and checkout a feature branch named owner-filter
2. In the Dog controller, we need to add this to the index method:  
   @owners = Owner.all  
   owner\_ids = params[:owner\_ids]
3. Then in the Dogs index view, we can add:

<div class="row">

<div class="col-md-12">

<p>

Filter by:

<% @owners.each do |owner| %>

<a href="?owner\_ids=<%= owner.id%>" class="filter"><button class="btn btn-default btn-xs"><%= owner.last\_first %></button></a>

<% end %>

</p>

</div>

</div>

1. Next, we need to go back to our controller and add the logic. Let’s pretend that we really only use this when we aren’t searching for a dog, so embed it inside the final else, like so:

if params[:search]

@dogs = Dog.where("name LIKE '%#{params[:search]}%'")

if @dogs.size.zero?

flash[:notice] = "No result found"

@dogs = Dog.all

end

elsif params[:owner\_ids]

@dogs = Dog.where(owner\_id: params[:owner\_ids].split(","))

else

@dogs = Dog.all

end

# Lesson 5.3: User Management Using Devise

## Gem of the Day

Devise <https://github.com/plataformatec/devise>

(welcome to add another if you like)

Other Options: [Clearance](https://github.com/thoughtbot/clearance), [Pundit](https://github.com/elabs/pundit), CanCan, CanCanCan, Rolify

* Clearance - simpler version of Devise
* CanCan - generates user role abilities
* CanCanCan - using this instead because developer for CanCan has dropped off face of earth
* Rolify - roles library without authorization enforcement?

## Lesson Plan

1. What is Devise and how does it work? What are some best practices?
2. Create a devise feature branch.
3. Install gem and generate initializer. Discuss the initializer and the options.
4. Generate the user model. Notes:
   1. We already have an Owner table, but that’s fine because we will have three types of users (if we get that far): Admins (can make any changes), Employees (can check in/check out dogs and charge for services), and General Public.
   2. The general public will be able to buy products off of our website once we build out that option.
5. Add controller filters for authenticating users.
6. Add sign in/sign up/sign out links based on logic. Put them in the navbar on the right. If you have time, separate the navbar into a partial. <https://github.com/plataformatec/devise/wiki/How-To:-Add-sign_in,-sign_out,-and-sign_up-links-to-your-layout-template>
7. Make the devise views and links prettier. Note: Bootstrap is already in our apps though we have slightly different themes.
8. Commit any final changes, then merge (no fast forward) back into master.

OPTIONAL: Allow email or username: <https://github.com/plataformatec/devise/wiki/How-To:-Allow-users-to-sign-in-using-their-username-or-email-address>

Roles? Talk about role types, binary, bitmask roll, string rolls etc.

Pundit vs CanCan vs CanCanCan vs Authority vs Rolify?

## Homework

Practice by adding Devise to another app - either one we already created or another. Add the user model and also update the views with logic for sign in/out/up. READ and understand the differences of all the authorization options available.