[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 5.2: 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>

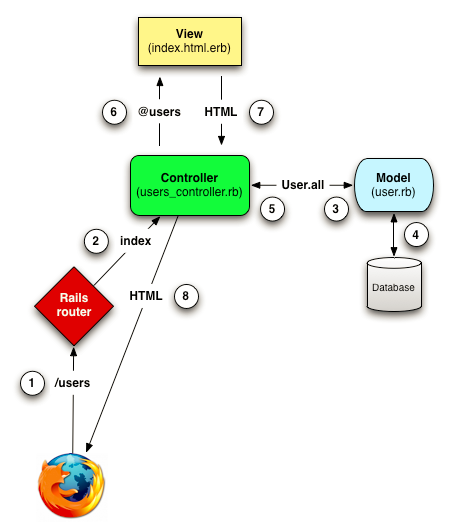
## Objectives

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