Controllers

Introduction

The controller's job is really to act as the ultimate middleman. It knows which questions it wants to ask the Model, but lets the model do all the heavy lifting for actually solving those questions. It knows which view it wants to render and send back to the browser, but lets the view itself take care of putting all that HTML together. That's why it's a "controller"... smart enough to know what to do and then delegate all the hard work. All it does is collect the proper batch of instance variables for sending over to the view.

When does the controller get used? After an HTTP request comes into your application and the router decides which controller and action to map it to, Rails packages up all the parameters that were associated with that request and runs the specified method in the specified controller. Once that method is done, Rails will take any instance variables you've given it in that controller method and ship them over to the appropriate view file so they can be inserted into your HTML template ("View") and ultimately sent back to the browser.

It's pretty straightforward. Typical controllers are pretty lightweight and don't have a whole lot of code but are able to do a lot of work with that code. What if you want to show all your blog posts in your site's index page? Run the `#index` action of your Posts controller and it will grab all your posts and send them over to the `index.html.erb` view file, which figures out how you actually want them displayed (in a giant bulleted list? With slick looking panels?).

The controller's `#index` action would actually look as simple as:

```
PostsController < ApplicationController
  ...
```

```
    def index
      @posts = Post.all
    end
    ...
end
```

In this simple action, we have the controller asking the model for something ("Hey, give me all the posts!"), packaging them up in an instance variable `@posts` so the view can use them, then will automatically render the view at `app/views/posts/index.html.erb` (we'll talk about that in a minute).

Points to Ponder

Look through these now and then use them to test yourself after doing the assignment

- Why is it important what you name your models, controllers, and views?
- Where is the view file located that's rendered by default for a given controller?
- What's the difference between a `#render` and a `#redirect_to`?
- What happens to the controller's instance variables in each case?
- What is a shortcut for redirecting to a specific Post (tip: this works in all kinds of places like `#link_to` and `#*_path`)
- Does a method finish executing or get interrupted when it hits a `#render` or `#redirect_to`?
- What happens if you have multiple renders or redirects?
- What are Strong Parameters?
- When can you just use the `params` hash directly and when do you need to specifically "whitelist" its contents?

- What are "scalar" values?
- What does `#require` do? `#permit`?
- What's the `#flash`?
- What's the difference between `#flash` and `#flash.now`?

Naming Matters

One way that Rails makes your life a bit easier is that it assumes things are named a certain way and then executes them behind the scenes based on those names. For instance, your controller and its action have to be named whatever you called them in your `routes.rb` file when you mapped a specific type of HTTP request to them.

The other end of the process is what the controller does when it's done. Once rails gets to the `end` of that controller action, it grabs all the instance variables from the controller and sends them over the view file *which is named the same thing as the controller action* and which lives in a folder named after the controller, e.g. `app/views/posts/index.html.erb`. This isn't arbitrary, this is intentional to make your life a lot easier when looking for files later. If you save your files in a different folder or heirarchy, you'll have to explicitly specify which ones you want rendered.

Rendering and Redirecting

Although Rails will implicitly render a view file that is named the same thing as your controller action, there are plenty of situations when you might want to override it. A main case for this is when you actually want to completely redirect the user to a new page instead of rendering the result of your controller action.

Redirects typically occur after controller actions where you've submitted information like to create a new Post. There's no reason to have

a `create.html.erb` view file that gets displayed once a post has been created... we usually just want to see the post we created and so we'll redirect over to the Show page for that post. The distinction here is that your application treats a redirect as *a completely new HTTP request*... so it would enter through the router again, look for the Show page corresponding to that post, and render it normally. That also means that any instance variables you set in your original `#create` controller action are wiped out along the way.

If that's the common way to deal with successfully creating an object, how about when it fails for some reason (like the user entered a too-short post title)? In that case, you can just render the view for another controller action, often the same action that created the form you just submitted (so the `#new` action).

The trick here is that the view page gets passed the instance variables from your *current* controller action. So let's say that you tried to `#create` a Post and stored it to `@post` but it failed to save. You then rendered the `#new` action's view and that view will receive the `@post` you were just working with in the `#create` action. This is great because you don't have to wipe the form completely clean (which is really annoying as a user) -- you can just identify the fields that failed and have the user resubmit. It may sound a bit abstract now but you'll see the difference quickly when building.

Let's see it in code:

```
# app/controllers/posts_controller.rb
class PostsController < ApplicationController
  ...
  # Make (but don't save) an empty Post so the form we render
  # knows which fields to use and where to submit the form
  # This action will render app/views/posts/new.html.erb once
  # it's done
  def new
    @post = Post.new
```

```
    end

    # We know this will get run once we receive the submitted
    # form from our NEW action above (remember your REST actions??)
    # We'll just use pseudo-code for now to illustrate the point
    def create
      ... code here to set up a new @post based on form info ...
      if @post.save
        ... code to set up congratulations message ...
        redirect_to post_path(@post.id) # go to show page for @post
      else
        ... code to set up error message ...
        render :new
      end
    end
  end
```

So the thing to pay attention to is that, if we successfully are able to save our new post in the database, we redirect to that post's show page. Note that a shortcut you'll see plenty of times is, instead of writing `redirect_to post_path(@post.id)`, just write `redirect_to @post` because Rails knows people did that so often that they gave you the option of writing it shorthand. We'll use this in the example as we develop it further.

The error condition in the `#create` action above is going to render the same form that we rendered in the `#new` action, though this time `@post` will be the Post object that we tried and failed to save, so it will also have some errors attached to it which can be used to highlight in red which form fields were the culprits.

Multiple Render/Redirects

It's important to note that `render` and `redirect_to` do NOT immediately stop your controller action like a `return` statement would. So you have to be really careful that your logic doesn't result in you running more than one of those statements. If you do, you'll get hit with an error. They're usually pretty straightforward to debug.

If you write something like:

```
def show
  @user = User.find(params[:id])
  if @user.is_male?
    render "show-boy"
  end
  render "show-girl"
end
```

In any case where the user is male, you'll get hit with a multiple render error because you've told Rails to render both "show-boy" and "show-girl".

Params and Strong Parameters

In the example above, we saw `... code here to set up a new @post based on form info ...`. Okay, how do we grab that info? We keep saying that the router packages up all the parameters that were sent with the original HTTP request, but how do we access them?

With the `params` hash! It acts just like a normal Ruby hash and contains the parameters of the request, stored as `:key => value` pairs. So how do we get the ID of the post we're asking for? `params[:id]`. You can access any parameters this way which have "scalar values", e.g. strings, numbers, booleans, `nil`... anything that's "flat".

Some forms will submit every field as a top level scalar entry in the params hash, e.g. `params[:post_title]` might be "test post" and `params[:post_body]` might be "body of post" etc and these you can access with no issues. You have control over this, as you'll learn in the lessons on forms.

Strong Parameters

Often times, though, you want to send parameters from the browser that are all packaged nicely into a hash or nested into an array. It can make

your life a lot easier because you can just pass that hash straight into `Post.new(your_hash_of_attributes_here)` because that's what `Post.new` expects anyway! We won't really get too deep into this stuff until the lessons on Models and Forms, but you should be aware that the structure of the data you're being sent from a form depends entirely on how you choose to set up that form (it's really based on how you choose to name your fields using the HTML `name=''` attribute).

In our example, we will assume that our `params[:post]` is giving us a hash of Post attributes like `{ :title => "test post", :body => "this post rocks", :author_id => "1"}`, which is exactly what `Post.new` is expecting. Whether you get the parameters packaged up in a hash or all on the top level as individual attributes (like `params[:id]`), again, is up to you and how you create your form. But know it's usually easier for you to handle receiving a nicely packaged hash of Post attributes in your controller.

The important distinction between the "scalar" parameter values like strings and more complex parameters like hashes and arrays is that Rails 4 implemented some protections in the controller, called "Strong Parameters". This is so the user can't send you harmful data (like automatically setting themselves as an admin user when they create an account). To do this, Rails makes you explicitly verify that you are willing to accept certain items of a hash or array.

Note: This used to be done in Rails 3 by setting `attr_accessible` in the model to whitelist attributes, so you will probably see that in a lot of Stack Overflow posts and earlier applications.

To "whitelist", or explicitly allow, parameters, you use the methods `require` and `permit`. Basically, you `require` the name of your array or hash to be in Params (otherwise it'll throw an error), and then you `permit` the individual attributes inside that hash to be used. For example:

```
  def whitelisted_post_params
    params.require(:post).permit(:title,:body,:author_id)
  end
```

This will whitelist and return the hash of only those params that you specified (e.g. `{:title => "your title", :body => "your body", :author_id => "1"}`). If you didn't do this, when you tried to access params[:post] nothing would show up! Also, if there were any additional fields submitted inside the hash, these will be stripped away and made inaccessible (to protect you).

It can be inconvenient, but it's Rails protecting you from bad users. You'll usually package these strong parameter helpers up in their own private method at the bottom of your controllers, then call that method where you need to get those specific params.

So our `#create` action above can now be filled out a bit more:

```
# app/controllers/posts_controller.rb
class PostsController < ApplicationController
  ...

  # We know this will get run once we've received the submitted
  # form from our new action above (remember your REST actions??)
  def create
    @post = Post.new(whitelisted_post_params) # see method below
    if @post.save
      ... code to set up congratulations message ...
      redirect_to post_path(@post.id) # go to show page for @post
    else
      ... code to set up error message ...
      render :new
    end
  end

  private  # Best to make helper methods like this one private

  # gives us back just the hash containing the params we need to
  # to create or update a post
  def whitelisted_post_params
    params.require(:post).permit(:title,:body,:author_id)
  end
end
```

Flash

The last piece of code we need to write there is how to set our special messages for the user. Rails gives you a neat tool for sending success and error messages (like the little green message that briefly appears at the top of an application to congratulate you for signing up) called the "flash". It acts just like a hash -- you can set its keys to a specific message and then that will be available for you to access if you want to display it in your views.

You can use any keys you want for the flash, but it's conventional to just stick to three, `:success`, `:error`, and `:notify`. So the success message above might look like `flash[:success] = "Great! Your post has been created!"`.

The reason you can use any key is because you will have to write a snippet of code in your view anyway to display the flash, but sticking to the conventional ones is good practice. The other sneaky trick with the flash is that it automatically erases itself once you've used it, so you don't have to worry about it displaying every time you visit a new page... one time use. Like Snapchat.

One last distinction, though, goes back to the difference between a redirect and a render. Remember, a redirect submits a completely new HTTP request, effectively leaving our application in the dust and starting over from the top. We lose all our data... except for the flash. The flash is specifically designed to travel with that HTTP request so you have access to it when you get redirected to the next page.

Render doesn't go that far -- it just uses a view file that's part of your application's normal flow and you have access to all your instance variables in that file. Because the flash is special, you actually have to use `flash.now` instead of `flash` when you are just rendering a view instead of submitting a whole new request. That would look like `flash.now[:error] = "Rats! Fix your mistakes, please.`.

The distinction between `flash` and `flash.now` just lets Rails know when it will need to make the flash available to you... if you used `flash` when you should have used `flash.now`, you'll just start seeing your messages showing up a "page too late" and it should be obvious what went wrong.

Now the full controller code can be written out for our `#create` action:

```
# app/controllers/posts_controller.rb
class PostsController < ApplicationController
  ...

  # We know this will get run once we've received the submitted
  # form from our new action above (remember your REST actions??)
  def create
    @post = Post.new(whitelisted_post_params)
    if @post.save
      flash[:success] = "Great! Your post has been created!"
      redirect_to @post # go to show page for @post
    else
      flash.now[:error] = "Rats! Fix your mistakes, please."
      render :new
    end
  end

    private

      def whitelisted_post_params
        params.require(:post).permit(:title,:body,:author_id)
      end
end
```

So that action did a fair bit of stuff -- grab the form data, make a new post, try to save the post, set up a success message and redirect you to the post if it works, and handle the case where it doesn't work by berating you for your foolishness and re-rendering the form. A lot of work for only 10 lines of Ruby. Now that's smart controlling.

Your Assignment

That's really just a taste of the Rails controller, but you should have a pretty good idea of what's going on and what tricks you can use.

1. Read the Rails Guides chapter on Controllers, sections 1 - 4.5.3 and 5.2. We'll cover sessions (section 5.1) more in the future so don't worry about them now.

Additional Resources

This section contains helpful links to other content. It isn't required, so consider it supplemental for if you need to dive deeper into something

- Rails 3 Rendering and Partials via YouTube