Views

Introduction

The view is really the simplest part of the MVC structure -- at the basic level, it's just a bunch of HTML boilerplate into which you insert the variables you've received from your controller and which will be sent to the browser. It's your actual "webpage". It will often have some snippets of code designed to properly present the variables it has received, for instance a loop that will display each one of the posts on your blog. Views are often called view templates.

Views live in the directory `app/views/controller_name/action_name.html.erb`, where `controller_name` is the name of the controller the view is linked to and `action_name.html.erb` is the corresponding method inside the controller that was run immediately prior to rendering the view.

So the `Posts` controller running the `#index` action will implicitly render the `app/views/posts/index.html.erb` view when it's done. You can explicitly tell your controller to render a differently named view by passing it as a parameter to the `render` function in your controller, but why? This directory and naming structure helps you (and Rails) to always know where to find a given view.

To use an instance variable from your controller, just call it the same way you would in the controller: `@user.first_name` or `@posts` or `@some_other_variable`.

As always, in this lesson we'll cover the high level stuff then have you read the Rails Guide for a more detailed understanding of how things work.

Points to Ponder

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

•What is a layout?

•What's the difference between a "view template" and a "layout"?

•What is a "Preprocessor"?

•Why are preprocessors useful?

•How do you make sure a preprocessor runs on your file?

•What's the outputted filetype of a preprocessed `*.html.erb` file? What about a `*.css.scss` file?

•What is the difference between the `<%=` and `<%` tags?

•What is a view partial?

•How do you insert a partial into your view?

•How can you tell that a view file is a partial?

•How do you pass a local variable to a partial?

•What's the magical Rails shortcut for rendering a User? A bunch of Users?

•What are asset tags and why are they used?

Layouts

The first thing to note is that the named view template we render from the controller is actually not the entire webpage. It doesn't contain the `<head>` tags or the `DOCTYPE` declaration or some of the other basic structure that's present in all pages. Precisely because those things are present in all of your pages, the Rails creators were smart enough to turn that code into its own file called a "layout". Layouts live in the directory `app/views/layouts`.

For a brand new Rails application, the `application.html.erb` layout is pretty basic. It's got the basic tags you need in all webpages (e.g. `<html>` and `<body>`) and a couple snippets of code that load up the javascript and css files your webpage will need. You'll want to put anything that's needed across all your webpages into the layout. Usually this is stuff like navbars and footers and snippets of code for displaying flash messages.

So if a layout is basically just a shell around the individual page, how does the page get inserted? That brings us back to the magic of the `#yield` method, which you saw when you learned about blocks. The view template at `app/views/posts/index.html.erb` gets inserted where the yield statement is. When you get more advanced, you'll be able to play around a bit with that statement but for now it's just that simple.

Preprocessors

The other thing you've undoubtedly noticed is the odd HTML code that goes inside `<%=` and `%>` tags. This is Embedded Ruby (ERB). It's a special way of executing ruby code inside your HTML. HTML is static, so you need to dial in some Ruby if you want to do anything dynamic like looping, `if` statements or working with variables. ERB (and another similar language you might see called HAML) do exactly that.

What those tags do is execute whatever you see inside them exactly as if it was normal Ruby. So `<%= "<em>I am emphasized</em>" %>` will output an emphasized piece of text like `<em>I am emphasized</em>` and `<%= @user.first_name %>` will output `joe`.

The difference between `<%` and `<%=` is that the `<%=` version actually displays whatever is returned inside the ERB tags. If you use `<%`, it will execute the code but, no matter what is returned by that line, it will not actually display anything in your HTML template.

Most of your tags will be `<%=` because you'll find yourself often just outputting important pieces of the instance variables that you received from your controller, like in the `<%= @user.first_name %>` example above. You use the `<%` for purely code-related stuff like `if` statements and `each` loops, where you don't actually WANT anything displayed (the stuff you display will occur inside the loop).

If this is confusing, here's an example. Say we want to display the first names of all the users in our application but only if the current user is signed in. This might look something like:

```
<% if current_user.signed_in? %>
  <ul>
    <% @users.each do |user| %>
      <li><%= user.first_name %></li>
    <% end %>
  </ul>
<% else %>
  <strong>You must sign in!</strong>
<% end %>
```

Remember to close your statements and loops with `<% end %>`! (You'll forget a few times).

In the code above, if the user is signed in it will actually render to the web something like:

```
<ul>
  <li>Bob</li>
  <li>Joe</li>
  <li>Nancy</li>
</ul>
```

If the user isn't signed in, it'll be the much shorter:

```
<strong>You must sign in!</strong>
```

In the above code, if we had accidentally used `<%=` in the loop line, e.g. `<%= @users.each do |user| %>` it would run the code fine, but because `each` returns

the original collection, we'd also see a dump of our `@users` variable on our page (not very professional). It'll happen to you several times and you'll learn quick.

How Do Preprocessors Work?

The important thing to note about the above code execution is that it is all done on the server BEFORE the final HTML file is shipped over to the browser (part of the Asset Pipeline, covered in the next lesson). That's because, when you render your template in Rails, it first runs "preprocessors" like ERB. It knows you want to preprocess the file because it has the extension `.html.erb`.

Rails starts from the outside in with extra extensions. So it first processes the file using ERB, then treats it as regular HTML. That's fine because ERB by definition outputs good clean HTML, like we saw above.

There are other preprocessors you'll run into as well. `.css.scss` files use the SASS preprocessor and become regular CSS files. `.js.coffee` files, which the Coffeescript preprocessor, become regular Javascript after it is run. In both these cases, the preprocessor's language makes your life easier by giving you some additional tools you can use (like having loops and working with variables) and compiles back down into a plain vanilla CSS or Javascript or HTML.

The point is, there are many different preprocessors. They are usually gems that either already come with Rails or can easily be attached to it. Rails then runs them automatically, so all you have to worry about is whether your file has the right extension(s) to tell the preprocessor to run.

View Partials

Another nice thing you can do in Rails is break apart your views into partials. This helps you on several levels -- it makes your code more concise and easier to read, and also lets you reuse certain common

patterns. One example is the form for creating or editing users. Both the `#new` and `#edit` actions need to render some sort of form for the user, and usually that form is almost exactly the same. So often people will turn that form into a new file called something like `_user_form.html.erb` and then just call that in both the `new.html.erb` and `edit.html.erb` view templates where it's needed.

Pulling back a bit, partials are just HTML files that aren't meant to be complete but can be shared by other files. You would call a partial by writing something like:

```
# app/views/users/new.html.erb
<div class="new-user-form">
  <%= render "user_form" %>
<div>
```

There are a couple of syntax oddities you need to pay attention to. The view partial file is named with an underscore like `_user_form.html.erb` but gets called using just the core portion of the name, e.g. `user_form` in the example above.

If there is no directory specified in partial's name, Rails will only look in the same folder as whichever view called it, e.g. `app/views/users`. Sometimes it makes sense to share partials across multiple view templates that are in multiple controllers, so you save them in their own folder called `app/views/shared` and would then render them using the code `<%= render "shared/some_partial"%>`.

Passing Local Variables to Partials

There's a lot you can do with partials and we won't dive into it all here, but the last thing that you might find yourself doing a lot is passing variables to partials. A partial has access to all the variables that the calling view template does, but do NOT rely on them! What if your partial is used by a different controller that uses a different structure for its instance variables?

It's bad code to expect an instance variable like `@user` to be there in the partial all the time. That means you've got to explicitly pass the partial whichever variables you want it to have access to.

In the example above, you most likely want to pass the `@user` variable to the partial so your code can render the right kind of form. `render` is just a regular method and it lets you pass it an options hash. One of those options is the `:locals` key, which will contain the variables you want to pass. Your code might change to look like:

```
<%= render "shared/your_partial", :locals => { :user => user } %>
```

To use the variable in your partial file, you drop the `@` and call it like a normal variable.

Implicit Partials

As usual, there are some things you would end up doing so many times that Rails has given you a short cut. One of these is the act of rendering a model object like a User or a Post. If you want a list of all your users, you could write out the HTML and ERB code for displaying a single user's first name, last name, email etc many times directly in your `app/views/users/index.html.erb` file or you could keep that code in some sort of `each` loop.

But it's usually best to make the User into its own partial called `_user.html.erb` so you can re-use it in other cases as well. The basic way of calling this might be something just like we saw above, which looks like:

```
# app/views/index.html.erb
<h1>Users</h1>
<ul>
  <% @users.each do |user| %>
    <%= render "user", :locals => {:user => user} %>
  <% end %>
</ul>
```

And in your partial:

```
# app/views/_user.html.erb
<li><%= "#{user.first_name} #{user.last_name}, #{user.email}" %></li>
```

It may seem strange to have only one line in a partial, but trust me that it usually doesn't stay that way for long so it's worth getting the hang of.

So if that's the basic way, what's the magical Rails way? Just tell it to render the User object directly, e.g.

```
# app/views/index.html.erb
<h1>Users</h1>
<ul>
  <% @users.each do |user| %>
    <%= render user %>     <!-- Lots less code -->
  <% end %>
</ul>
```

Rails then looks for the `_user.html.erb` file in the current directory and passes it the `user` variable automatically.

What if you want to render a whole bunch of users like we just did? Rails also does that for you the same way, saving you the trouble of writing out your own `each` loop like we did above. Simply write:

```
# app/views/index.html.erb
<h1>Users</h1>
<ul>
  <%= render @users %>
</ul>
```

In that situation, Rails not only finds the `_user.html.erb` file and passes it the correct `user` variable to use, it also loops over all the users in your `@user` collection for you. Pretty handy.

Helper Methods

`render`ing partials isn't the only method you can call from within a view. Rails has a bunch of really handy helper methods that are available for you to use in the view. A few of the most common:

`#link_to`

`link_to' creates an anchor tag URL. Instead of writing:

```
<a href="<%= users_path %>">See All Users</a>
```

You write:

```
<%= link_to "See All Users", users_path %>
```

It's the Rails way. And recall that `users_path` generates a relative URL like `/users` whereas `users_url` generates a full URL like `http://www.yourapp.com/users`. In most cases, it isn't an important distinction because your browser can handle both but make sure you understand the difference.

Asset Tags

As you may have seen in the application layout file we talked about above, Rails gives you helper methods that output HTML tags to grab CSS or Javscript files. You can also grab images. These are called Asset Tags. We'll get into the "Asset Pipeline" a bit later, but basically these tags locate those files for you based on their name and render the proper HTML tag.

```
<%= stylesheet_link_tag "your_stylesheet" %>
<%= javascript_include_tag "your_javascript" %>
<%= image_tag "happy_cat.jpg" %>
```

Will render something like:

```
<link href="/assets/your_stylesheet.css" media="all" rel="stylesheet">
```

```
    <script src="/assets/your_stylesheet.js"></script>
    <img src="/assets/happy_cat.jpg">
```

note: in production, your stylesheet and javascripts will all get mashed into one strangely-named file, so don't be alarmed if it's named something like `/assets/application-485ea683b962efeaa58dd8e32925dadf`

Forms

Rails offers several different helpers that help you create forms, and we'll go over those in depth in upcoming lessons.

Your Assignment

Now that you've got a taste of the high level stuff, read through the Rails Guides for a more detailed look at things. The chapter below will actually start in the controller, where you need to let it know WHICH view file you want to render. The second half of the chapter gets more into the view side of things.

1.Read the Rails Guide chapter on Layouts and Rendering, sections 1 through 3.4. You can certainly skim when they start going over all the many different options you can pass to a given function... it's good to know what they are and where you can find them, but you don't need to memorize all of them. Usually you'll have something that you want to do, Google it, and find a Stack Overflow post that shows you the option you can use.

Conclusion

Views in general make up the user-facing side of your app. It can be a bit tricky at first to imagine how you choose which view to render, what to include in that view and how to use partials, but a few iterations of working with Rails will show you the conventions pretty quickly. Views will become second nature to you.