Better Views

Utilizing View Partials

When writing Ruby we break up complex methods into multiple, smaller methods. When writing view templates, partials are the means of encapsulating view components.

Setup

NOTE

Get the Blogger project from GitHub and run setup procedures:

```
git clone git://github.com/JumpstartLab/blogger_advanced.git
cd blogger_advanced
bundle
bundle exec rake db:setup
bundle exec rake
```

All existing tests should pass. Optionally, run the tests continuously while developing by running `guard`

Simple Partials

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Open `views/articles/show.html.erb` and look for the H3 that starts the comments section. The H3 line and everything below it are about comments. They are related to the article, but are not intrinsic to *showing* an article. They are a perfect candidate for a simple partial extraction.

We create partials by adding a file to the views folder and beginning the filename with an underscore.

As an example, create `views/articles/_comments.html.erb` and move the H3 and everything below it into that file. Save both files and look at an article's `show` in the browser. The comments should vanish.

Now, to render the partial we utilize the `render` method. At the bottom of `show.html.erb`, add:

```
1  <%= render partial: 'comments' %>
```

Refresh your browser and *you'll notice an error*.

The issue is because of the `form_for` method being called just above the `render` we've just added.

As you can see, it is creating a new comment in memory with `@article.comments.new`. This creates a new comment for our current article, but since that comment is not being saved there are certain attributes on that comment that haven't been set yet (`id`, `created_at`, and `updated_at`).

If we look at our partial, it is looping through each of the article's comments (this new one included) and using their `created_at` attribute to find the `distance_of_time_in_words` for each comment compared to the time in which the article was created.

Because the new comment created by the `form_for` hasn't been saved to the database - and as such doesn't have a value for `created_at`, our application is breaking.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

In order to fix this, in the partial you'll need to check each comment to make sure that it has indeed been saved to the database.

You can do this by wrapping the `<div class='comment'>` block with an conditional statement like so:

```
<% unless comment.new_record? %>
  <div class='comment'>
    ... the rest of the code ...
  </div>
<% end %>
```

The `new_record?` method will ask Rails if the object is a new record that hasn't been saved to the database. This will then allow us to skip over that new comment created by the `form_for` helper.

Relocating Partials

`render`, by default, looks for the partial in the same directory as the current view template. In this case, that means `app/views/articles`.

As this application grows, we might want to reuse the comment partial on other pages. Maybe our user can post images that are not articles. We would like readers to be able to comment on them, too! We can build in that flexibility now.

Create a directory `app/views/common` and move the `_comments.html.erb` into it.

Go to an article's `show` page in your browser, and it will crash because it cannot find the partial `app/views/articles/_comments.html.erb`

Open `app/views/articles/show.html.erb` and change this:

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
1  <%= render partial: 'comments' %>
```

to this:

```
1  <%= render partial: 'common/comments' %>
```

When `render` sees a `/` in the partial name, it interprets the first part as the folder name and the second as the file name.

Passing In a Variable

Sending variables into a partial is a little tricky.

To see how it works, first go into your partial and change all references from `@article` to the local variable `article`. The rendering will now break because it doesn't have a local variable named `article`.

Then, in the `show` template, modify the `render` call to this:

```
1  <%= render partial: 'common/comments', locals: {article: @article} %>
```

The `locals` option takes a hash. Each key will be setup as a local variable and the value stored into the variable. So, in the context of the partial, we'll now have an `article` variable holding `@article`.

Refresh the `show` page in your browser and it should render correctly.

To make the partial truly reusable, we should edit it to refer to a local variable named `subject` then, when rendering it, pass in `subject: @article`.

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Rendering Collections

The `render` method is incredibly overloaded. Let's see how it can work with collections of objects. Open `views/articles/index.html.erb`.

See the `@article.each` line? Whenever we have an iteration loop in a view template, it is a candidate for extraction to a collection partial.

To see how it works:

- Cut the `li` and everything beneath it to your clipboard
- Delete the `<% @articles.each do |article| %>` line
- Create a file `app/views/articles/_article_item.html.erb` and paste it in

Refresh your index page in the browser and the articles will disappear.

We want to render the LIs inside the `ul` with ID `articles`. Let's try it in one line:

```
1  <ul id='articles'><%= render partial: 'article_item'%></ul>
```

That's a good start, but we don't want to render it *once*, we need to render it *once for each article*. Add the `:collection` parameter like this:

```
1  <ul id='articles'><%= render partial: 'article_item', collection: @articles %></ul>
```

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

Refresh your browser and it still crashes. The partial is looking for a variable named `article` but can't find one.

When you call `render` using a collection, it will process the partial once for each element of the collection. While the partial is being rendered, Rails will provide the element being rendered and store it into a local variable *based on the filename of the partial*.

So in this case, our `_article_item.html.erb` partial will have a local variable named `article_item`.

To make our view work, we have two options.

1. Open the partial and change all references from `article` to `article_item` to match the filename.
2. Rename the partial to `_article.html.erb` so it'll have a local `article` variable.

Implement the second option, renaming the file. Then update the `render` call like this:

```
1  <ul id='articles'><%= render partial: 'article', collection: @articles %></ul>
```

Refresh your browser and the view should display correctly.

Magical Partial Selection

When we first rendered the comments partial, you might have known that instead of:

```
1  <%= render partial: 'comments' %>
```

We could have just written this:

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

```
1  <%= render 'comments' %>
```

If you give `render` a string, it will attempt to render a partial with that name. But, due to implementation details of the `render` method, you *cannot* leave off the `:partial` and still use `:locals`:

```
1  <%= render 'comments', locals: {article: @article} %>
```

Nor can you leave off `:partial` when rendering a collection. This *will not work*:

```
1  <ul id='articles'><%= render 'article', collection: @articles %></ul>
```

There is a shortened syntax that *will* work. You can do this:

```
1  <ul id='articles'><%= render @articles %></ul>
```

`render` accepts an object or a collection of objects. `render` will iterate through the objects and call the `.class_name` method on each one, convert the class name to `snake_case`, and will render a partial with that name. The individual object sent will still be named after the partial.

`render @articles` will thus render the `_article.html.erb` partial once for each article in `@articles`, assigning each one to the local variable `article`.

Closing Words on View Partials

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD

A few last thoughts on view partials:

- For consistency, use the syntax `render partial: x` and `render partial: x, collection: y`
- An `app/views/common` folder is helpful on most projects to hold reusable partials
- Generally, don't nest partials more than two levels deep. For example:
  - `show.html.erb` can render a `_comments.html.erb`
  - `_comments.html.erb` can render `_comment_form.html.erb`
  - Don't make `_comment_form.html.erb` render `_comment_form_elements.html.erb`, otherwise it gets too difficult to understand the template structure

All materials licensed Creative Commons Attribution-NonCommercial-ShareAlike 3.0 

Create PDF in your applications with the Pdfcrowd HTML to PDF API PDFCROWD