DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Action View Overview

After reading this guide, you will know:

- What Action View is and how to use it with Rails.
- How best to use templates, partials, and layouts.
- How to use localized views.

What is Action View?

In Rails, web requests are handled by Action Controller and Action View. Typically, Action Controller is concerned with communicating with the database and performing CRUD actions where necessary. Action View is then responsible for compiling the response.

Action View templates are written using embedded Ruby in tags mingled with HTML. To avoid cluttering the templates with boilerplate code, several helper classes provide common behavior for forms, dates, and strings. It's also easy to add new helpers to your application as it evolves.

NOTE: Some features of Action View are tied to Active Record, but that doesn't mean Action View depends on Active Record. Action View is an independent package that can be used with any sort of Ruby libraries.

Using Action View with Rails

For each controller, there is an associated directory in the `app/views` directory which holds the template files that make up the views associated with that controller. These files are used to display the view that results from each controller action.

Let's take a look at what Rails does by default when creating a new resource using the scaffold generator:

```
$ bin/rails generate scaffold article
      [...]
      invoke  scaffold_controller
      create    app/controllers/articles_controller.rb
      invoke    erb
      create      app/views/articles
      create      app/views/articles/index.html.erb
      create      app/views/articles/edit.html.erb
      create      app/views/articles/show.html.erb
      create      app/views/articles/new.html.erb
      create      app/views/articles/_form.html.erb
      [...]
```

There is a naming convention for views in Rails. Typically, the views share their name with the associated controller action, as you can see above. For example, the index controller action of the `articles_controller.rb` will use the `index.html.erb` view file in the `app/views/articles` directory. The complete HTML returned to the client is composed of a combination of this ERB file, a layout template that wraps it, and all the partials that the view may reference. Within this guide, you will find more detailed documentation about each of these three components.

Templates, Partials, and Layouts

As mentioned, the final HTML output is a composition of three Rails elements: `Templates` , `Partials` and `Layouts` . Below is a brief overview of each of them.

Templates

Action View templates can be written in several ways. If the template file has a `.erb` extension then it uses a mixture of ERB (Embedded Ruby) and HTML. If the template file has a `.builder` extension then the `Builder::XmlMarkup` library is used.

Rails supports multiple template systems and uses a file extension to distinguish amongst them. For example, an HTML file using the ERB template system will have `.html.erb` as a file extension.

ERB

Within an ERB template, Ruby code can be included using both `<% %>` and `<%= %>` tags. The `<% %>` tags are used to execute Ruby code that does not return anything, such as conditions, loops, or blocks, and the `<%= %>` tags are used when you want output.

Consider the following loop for names:

```
<h1>Names of all the people</h1>
<% @people.each do |person| %>
  Name: <%= person.name %><br>
<% end %>
```

The loop is set up using regular embedding tags (`<% %>`) and the name is inserted using the output embedding tags (`<%= %>`). Note that this is not just a usage suggestion: regular output functions such as `print` and `puts` won't be rendered to the view with ERB templates. So this would be wrong:

```
<%# WRONG %>
Hi, Mr. <% puts "Frodo" %>
```

To suppress leading and trailing whitespaces, you can use `<%-` `-%>` interchangeably with `<%` and `%>` .

Builder

Builder templates are a more programmatic alternative to ERB. They are especially useful for generating XML content. An XmlMarkup object named `xml` is automatically made available to templates with a `.builder` extension.

Here are some basic examples:

```
xml.em("emphasized")
xml.em { xml.b("emph & bold") }
xml.a("A Link", "href" => "https://rubyonrails.org")
xml.target("name" => "compile", "option" => "fast")
```

which would produce:

```
<em>emphasized</em>
<em><b>emph &amp;amp; bold</b></em>
<a href="https://rubyonrails.org">A link</a>
<target option="fast" name="compile" />
```

Any method with a block will be treated as an XML markup tag with nested markup in the block. For example, the following:

```
xml.div {
  xml.h1(@person.name)
  xml.p(@person.bio)
}
```

would produce something like:

```
<div>
  <h1>David Heinemeier Hansson</h1>
  <p>A product of Danish Design during the Winter of '79...</p>
</div>
```

Below is a full-length RSS example actually used on Basecamp:

```
xml.rss("version" => "2.0", "xmlns:dc" => "http://purl.org/dc/elements/1.1/") do
  xml.channel do
    xml.title(@feed_title)
    xml.link(@url)
    xml.description "Basecamp: Recent items"
    xml.language "en-us"
    xml.ttl "40"

    for item in @recent_items
      xml.item do
        xml.title(item_title(item))
        xml.description(item_description(item)) if item_description(item)
        xml.pubDate(item_pubDate(item))
        xml.guid(@person.firm.account.url + @recent_items.url(item))
        xml.link(@person.firm.account.url + @recent_items.url(item))
        xml.tag!("dc:creator", item.author_name) if item_has_creator?(item)
      end
    end
  end
end
```

Jbuilder

Jbuilder is a gem that's maintained by the Rails team and included in the default Rails `Gemfile`. It's similar to Builder but is used to generate JSON, instead of XML.

If you don't have it, you can add the following to your `Gemfile`:

```
gem 'jbuilder'
```

A Jbuilder object named `json` is automatically made available to templates with a `.jbuilder` extension.

Here is a basic example:

```
json.name("Alex")
json.email("alex@example.com")
```

would produce:

```
{
  "name": "Alex",
  "email": "alex@example.com"
}
```

See the Jbuilder documentation for more examples and information.

Template Caching

By default, Rails will compile each template to a method to render it. In the development environment, when you alter a template, Rails will check the file's modification time and recompile it.

Partials

Partial templates - usually just called "partials" - are another device for breaking the rendering process into more manageable chunks. With partials, you can extract pieces of code from your templates to separate files and also reuse them throughout your templates.

Naming Partials

To render a partial as part of a view, you use the `render` method within the view:

```
<%= render "menu" %>
```

This will render a file named `_menu.html.erb` at that point within the view that is being rendered. Note the leading underscore character: partials are named with a leading underscore to distinguish them from regular views, even though they are referred to without the underscore. This holds true even when you're pulling in a partial from another folder:

```
<%= render "shared/menu" %>
```

That code will pull in the partial from `app/views/shared/_menu.html.erb`.

Using Partials to simplify Views

One way to use partials is to treat them as the equivalent of subroutines; a way to move details out of a view so that you can grasp what's going on more easily. For example, you might have a view that looks like this:

```
<%= render "shared/ad_banner" %>

<h1>Products</h1>

<p>Here are a few of our fine products:</p>
<% @products.each do |product| %>
  <%= render partial: "product", locals: { product: product } %>
<% end %>

<%= render "shared/footer" %>
```

Here, the `_ad_banner.html.erb` and `_footer.html.erb` partials could contain content that is shared among many pages in your application. You don't need to see the details of these sections when you're concentrating on a particular page.

`render` without `partial` and `locals` options

In the above example, `render` takes 2 options: `partial` and `locals`. But if these are the only options you want to pass, you can skip using these options. For example, instead of:

```
<%= render partial: "product", locals: { product: @product } %>
```

You can also do:

```
<%= render "product", product: @product %>
```

The `as` and `object` options

By default `ActionView::Partials::PartialRenderer` has its object in a local variable with the same name as the template. So, given:

```
<%= render partial: "product" %>
```

within `_product` partial we'll get `@product` in the local variable `product`, as if we had written:

```
<%= render partial: "product", locals: { product: @product } %>
```

The `object` option can be used to directly specify which object is rendered into the partial; useful when the template's object is elsewhere (e.g. in a different instance variable or in a local variable).

For example, instead of:

```
<%= render partial: "product", locals: { product: @item } %>
```

we would do:

```
<%= render partial: "product", object: @item %>
```

With the `as` option, we can specify a different name for the said local variable. For example, if we wanted it to be `item` instead of `product` we would do:

```
<%= render partial: "product", object: @item, as: "item" %>
```

This is equivalent to

```
<%= render partial: "product", locals: { item: @item } %>
```

Rendering Collections

Commonly, a template will need to iterate over a collection and render a sub-template for each of the elements. This pattern has been implemented as a single method that accepts an array and renders a partial for each one of the elements in the array.

So this example for rendering all the products:

```
<% @products.each do |product| %>
  <%= render partial: "product", locals: { product: product } %>
<% end %>
```

can be rewritten in a single line:

```
<%= render partial: "product", collection: @products %>
```

When a partial is called with a collection, the individual instances of the partial have access to the member of the collection being rendered via a variable named after the partial. In this case, the partial is `_product`, and within it, you can refer to `product` to get the collection member that is being rendered.

You can use a shorthand syntax for rendering collections. Assuming `@products` is a collection of `Product` instances, you can simply write the following to produce the same result:

```
<%= render @products %>
```

Rails determines the name of the partial to use by looking at the model name in the collection, `Product` in this case. In fact, you can even render a collection made up of instances of different models using this shorthand, and Rails will choose the proper partial for each member of the collection.

Spacer Templates

You can also specify a second partial to be rendered between instances of the main partial by using the `:spacer_template` option:

```
<%= render partial: @products, spacer_template: "product_ruler" %>
```

Rails will render the `_product_ruler` partial (with no data passed to it) between each pair of `_product` partials.

Layouts

Layouts can be used to render a common view template around the results of Rails controller actions. Typically, a Rails application will have a couple of layouts that pages will be rendered within. For example, a site might have one layout for a logged in user and another for the marketing or sales side of the site. The logged in user layout might include top-level navigation that should be present across many controller actions. The sales layout for a SaaS app might include top-level navigation for things like "Pricing" and "Contact Us" pages. You would expect each layout to have a different look and feel. You can read about layouts in more detail in the Layouts and Rendering in Rails guide.

Partial Layouts

Partials can have their own layouts applied to them. These layouts are different from those applied to a controller action, but they work in a similar fashion.

Let's say we're displaying an article on a page which should be wrapped in a `div` for display purposes. Firstly, we'll create a new `Article`:

```
Article.create(body: 'Partial Layouts are cool!')
```

In the `show` template, we'll render the `_article` partial wrapped in the `box` layout:

articles/show.html.erb

```
<%= render partial: 'article', layout: 'box', locals: { article: @article } %>
```

The `box` layout simply wraps the `_article` partial in a `div`:

articles/_box.html.erb

```
<div class='box'>
  <%= yield %>
</div>
```

Note that the partial layout has access to the local `article` variable that was passed into the `render` call. However, unlike application-wide layouts, partial layouts still have the underscore prefix.

You can also render a block of code within a partial layout instead of calling `yield`. For example, if we didn't have the `_article` partial, we could do this instead:

articles/show.html.erb

```
<% render(layout: 'box', locals: { article: @article }) do %>
  <div>
    <p><%= article.body %></p>
  </div>
<% end %>
```

Supposing we use the same `_box` partial from above, this would produce the same output as the previous example.

View Paths

When rendering a response, the controller needs to resolve where the different views are located. By default, it only looks inside the `app/views` directory.

We can add other locations and give them certain precedence when resolving paths using the `prepend_view_path` and `append_view_path` methods.

Prepend view path

This can be helpful for example when we want to put views inside a different directory for subdomains.

We can do this by using:

```
prepend_view_path "app/views/#{request.subdomain}"
```

Then Action View will look first in this directory when resolving views.

Append view path

Similarly, we can append paths:

```
append_view_path "app/views/direct"
```

This will add `app/views/direct` to the end of the lookup paths.

Helpers

Rails provides many helper methods to use with Action View. These include methods for:

- Formatting dates, strings and numbers
- Creating HTML links to images, videos, stylesheets, etc...
- Sanitizing content
- Creating forms
- Localizing content

You can learn more about helpers in the Action View Helpers Guide and the Action View Form Helpers Guide.

Localized Views

Action View has the ability to render different templates depending on the current locale.

For example, suppose you have an `ArticlesController` with a show action. By default, calling this action will render `app/views/articles/show.html.erb`. But if you set `I18n.locale = :de`, then `app/views/articles/show.de.html.erb` will be rendered instead. If the localized template isn't present, the undecorated version will be used. This means you're not required to provide localized views for all cases, but they will be preferred and used if available.

You can use the same technique to localize the rescue files in your public directory. For example, setting `I18n.locale = :de` and creating `public/500.de.html` and `public/404.de.html` would allow you to have localized rescue pages.

Since Rails doesn't restrict the symbols that you use to set I18n.locale, you can leverage this system to display different content depending on anything you like. For example, suppose you have some "expert" users that should

see different pages from "normal" users. You could add the following to `app/controllers/application_controller.rb`:

```
before_action :set_expert_locale

def set_expert_locale
  I18n.locale = :expert if current_user.expert?
end
```

Then you could create special views like `app/views/articles/show.expert.html.erb` that would only be displayed to expert users.

You can read more about the Rails Internationalization (I18n) API here.