# **Layouts and Rendering in Rails**

## **Creating Responses**

3 ways to create an HTTP response

1. ‘render’ to create a full response to send back to browser
2. ‘redirect’ to to send HTTP redirect status code to browser
3. ‘head’ to create response consisting only HTTP headers to send back

## **2.1 Rendering by Default: Convention Over Configuration in Action**

Controller automatically renders views with names corresponding to valid routes

BooksController class:

```
class BooksController < ApplicationController
end
```

Routes file:

```
resources :books
```

View file (apps/views/books/index.html.erb):

```
<h1>Books are coming soon!</h1>
```

Rails automatically renders apps/views/books/index.html.erb when we navigate to /books

```
class BooksController < ApplicationController
  def index
        @books = Book.all
  end
End
```

For displaying properties of all books in the view:

```
<h1>Listing Books</h1>

<table>
  <thead>
        <tr>
        <th>Title</th>
        <th>Content</th>
        <th colspan="3"></th>
        </tr>
  </thead>
  <tbody>
        <% @books.each do |book| %>
        <tr>
        <td><%= book.title %></td>
        <td><%= book.content %></td>
        <td><%= link_to "Show", book %></td>
        <td><%= link_to "Edit", edit_book_path(book) %></td>
        <td><%= link_to "Destroy", book, data: { turbo_method: :delete, turbo_confirm: "Are you
sure?" } %></td>
        </tr>
        <% end %>
  </tbody>
</table>

<br>

<%= link_to "New book", new_book_path %>
```

**<u>Using render</u>**

Text, JSON, XML etc can be rendered and content type can be specified.

To render a view corresponding to a different template with same controller (use render with view name)

```
def update
        @book = Book.find(params[:id])
        if @book.update(book_params)
                redirect_to(@book)
        else
                render = “edit”
        end
end
OR

def update
  @book = Book.find(params[:id])
  if @book.update(book_params)
        redirect_to(@book)
  else
        render :edit, status: :unprocessable_entity
  end
End
```

**2.2.2 Rendering an Action's Template from Another Controller**

Can render full different controller path too (accepts full path)
render "products/show"

All work:
render :edit
render action: :edit
render "edit"
render action: "edit"
render "books/edit"
render template: "books/edit"

**Render with :inline**

Render without a view (give inline erb html)
render inline: "<% products.each do |p| %><p><%= p.name %></p><% end %>"

(Not recommended)

**Rendering text**

render plain: "OK"

**Rendering JSON**

render json: @product

**Rendering HTML**

render html: helpers.tag.strong('Not Found')

**Rendering XML**

render xml: @product

**Rendering Vanilla JavaScript**

render js: "alert('Hello Rails');"

**Rendering Raw Body**

render body: "raw"

**Rendering Raw File**

render file: "#{Rails.root}/public/404.html", layout: false

(Useful for conditionally rendering static files like error pages)

**Rendering Objects**

Render objects responding to : render_in
render MyRenderable.new

render renderable: MyRenderable.new

**Options for render**

    :content_type (json, text etc)
    :layout (false, special layout etc)
    :location (HTTP location header)
    :status (200 OK, 500 Forbidden etc)
    :formats (html by default, can be given as array or symbol e.g. :xml or [:json, :xml]
    :variants (mobile, desktop)

**Specifying Layouts for Controllers**

To override the default layout convention:

```
class ProductsController < ApplicationController
  layout "inventory"
end
```

To assign a specific layout for the entire app:

```
class ApplicationController < ActionController::Base
  layout "main"
  #...
end
```

**Layouts at runtime**

For a special user:

```
class ProductsController < ApplicationController
  layout :products_layout
  def show
        @product = Product.find(params[:id])
  end

  private
        def products_layout
        @current_user.special? ? "special" : "products"
        end
end
```

Proc: A Proc object is an encapsulation of a block of code, which can be stored in a local variable.

Can also use procs for dynamic layout rendering:

```
class ProductsController < ApplicationController
  layout Proc.new { |controller| controller.request.xhr? ? "popup" : "application" }
end
```

**Conditional Rendering**

Layouts at controller level support only, except options.

```
class ProductsController < ApplicationController
  layout "product", except: [:index, :rss]
end
```

**Layout inheritance**

Layout declarations cascade downward in the hierarchy
Specific layout declarations override general layouts

1. application_controller.rb

```
class ApplicationController < ActionController::Base
  layout "main"
end
```

2. articles_controller.rb

```
class ArticlesController < ApplicationController
end
```

3. special_articles_controller.rb

```
class SpecialArticlesController < ArticlesController
  layout "special"
end
```

4. old_articles_controller.rb

```
class OldArticlesController < SpecialArticlesController
  layout false

  def show
        @article = Article.find(params[:id])
  end

  def index
        @old_articles = Article.older
        render layout: "old"
  end
  # ...
end
```

**Template Inheritance**

If a template or partial isn't found in a path, controller looks in the inheritance chain.

**Avoiding Double Render Errors**

Ensure: Have only one call to render or redirect in a single code path (can use return to help in this)

```
def show
  @book = Book.find(params[:id])
  if @book.special?
        render action: "special_show"
        return
  end
  render action: "regular_show"
end
```

**Using redirect_to**

This command sends a new request for a different url

```
redirect_to photos_url
```

**redirect_back** can return back to the page the user just came from.

```
redirect_back(fallback_location: root_path)
```

**Getting a Different Redirect Status Code**

```
redirect_to photos_path, status: 301
```

Can use status for using a different status code.

**Render vs redirect_to**

If it’s null render won’t run any code in target action, use redirect instead.

```
def index
  @books = Book.all
end

def show
  @book = Book.find_by(id: params[:id])
  if @book.nil?
        redirect_to action: :index
  end
end
```

**head to Builder Header-Only Responses**

head can send responses with only headers to browser.
```
head :bad_request
```

It can also convey other information.

```
head: created, location: photo_path(@photo )
```

**Structuring Layouts**

Rails combines the view with the current layout using:
1. Asset tags
2. yield and content for
3. Partials

**Asset Tag Helpers**

They provide methods for generating HTML that link views to feeds, JS, Stylesheets, images etc

Auto_discovery_link_tag (builds HTML to detect presence of rss, atom, json)
Javascript_include_tag (linking to javascript files, returns html script tag)
stylesheet_link_tag (links to css files)
Image_tag (<img /> tag, public/images)
Video_tag (<video> tag, public/videos)
audio_tag (<audio> tag, public/audios)

For example:

```
<%= audio_tag "music/first_song.mp3" %>
<%= video_tag ["trailer.ogg", "movie.ogg"] %>
<%= image_tag "home.gif", alt: "Go Home",
              id: "HomeImage",
              class: "nav_bar" %>
<%= stylesheet_link_tag "http://example.com/main.css" %>
<%= javascript_include_tag "http://example.com/main.js" %>
```

**Understanding yield**

A section where content from the view should be inserted.

```
<html>
  <head>
  <%= yield :head %>
  </head>
  <body>
  <%= yield %>
  </body>
</html>
```

**Using the content_for Method**

Insert content into a named yield block.

```
<% content_for :head do %>
  <title>A simple page</title>
<% end %>

<p>Hello, Rails!</p>
```

Very helpful when the layout has distinct regions e.g. sidebars, footers etc.

**Using Partials**

(Covered in last chapter)

**Counter Variables**

Counter variable is present within a collection and is named after the partial title

E.g. _product.html.erb can access product_counter.

It’s 0 on the first render. 1 for the second product and so on.

**Spacer Template**

:spacer_template option can be used to specify a second partial between instances of the main partial.

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

**Collection Partial Layouts**

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

Now the layout will be rendered together with the partial for each item in the collection.

**Nested Layouts**

Sub-templates / nested layouts allow us to work without repeating the main layout and editing it. Small chunks can be used.

To hide the top menu and add a right menu inside the “content” div for the News view, this can be done:

```
<% content_for :stylesheets do %>
  #top_menu {display: none}
  #right_menu {float: right; background-color: yellow; color: black}
<% end %>
<% content_for :content do %>
  <div id="right_menu">Right menu items here</div>
  <%= content_for?(:news_content) ? yield(:news_content) : yield %>
<% end %>
<%= render template: "layouts/application" %>
```

No limit for nesting levels.
If News layout is not to be subtemplates, can simply use “yield” instead of:
content_for?(:news_content) ? yield(:news_content) : yield