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

Layouts and Rendering in Rails

This guide covers the basic layout features of Action Controller and Action View.

After reading this guide, you will know:

- How to use the various rendering methods built into Rails.
- How to create layouts with multiple content sections.
- How to use partials to DRY up your views.
- How to use nested layouts (sub-templates).

Overview: How the Pieces Fit Together

This guide focuses on the interaction between Controller and View in the Model-View-Controller triangle. As you know, the Controller is responsible for orchestrating the whole process of handling a request in Rails, though it normally hands off any heavy code to the Model. But then, when it's time to send a response back to the user, the Controller hands things off to the View. It's that handoff that is the subject of this guide.

In broad strokes, this involves deciding what should be sent as the response and calling an appropriate method to create that response. If the response is a full-blown view, Rails also does some extra work to wrap the view in a layout and possibly to pull in partial views. You'll see all of those paths later in this guide.

Creating Responses

From the controller's point of view, there are three ways to create an HTTP response:

- Call `render` to create a full response to send back to the browser
- Call `redirect_to` to send an HTTP redirect status code to the browser
- Call `head` to create a response consisting solely of HTTP headers to send back to the browser

Rendering by Default: Convention Over Configuration in Action

You've heard that Rails promotes "convention over configuration". Default rendering is an excellent example of this. By default, controllers in Rails automatically render views with names that correspond to valid routes. For example, if you have this code in your `BooksController` class:

```ruby
class BooksController < ApplicationController
end
```

And the following in your routes file:

```ruby
resources :books
```

And you have a view file `app/views/books/index.html.erb`:

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

Rails will automatically render `app/views/books/index.html.erb` when you navigate to `/books` and you will see "Books are coming soon!" on your screen.

However, a coming soon screen is only minimally useful, so you will soon create your `Book` model and add the index action to `BooksController`:

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

Note that we don't have explicit render at the end of the index action in accordance with "convention over configuration" principle. The rule is that if you do not explicitly render something at the end of a controller action, Rails will automatically look for the `action_name.html.erb` template in the controller's view path and render it. So in this case, Rails will render the `app/views/books/index.html.erb` file.

If we want to display the properties of all the books in our view, we can do so with an ERB template like this:

```
<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
      </tr>
    <% end %>
  </tbody>
</table>

<br>

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

NOTE: The actual rendering is done by nested classes of the module `ActionView::Template::Handlers`. This guide does not dig into that process, but it's important to know that the file extension on your view controls the choice of template handler.

Using `render`

In most cases, the `ActionController::Base#render` method does the heavy lifting of rendering your application's content for use by a browser. There are a variety of ways to customize the behavior of `render`. You can render the default view for a Rails template, or a specific template, or a file, or inline code, or nothing at all. You can render text, JSON, or XML. You can specify the content type or HTTP status of the rendered response as well.

TIP: If you want to see the exact results of a call to `render` without needing to inspect it in a browser, you can call `render_to_string`. This method takes exactly the same options as `render`, but it returns a string instead of sending a response back to the browser.

Rendering an Action's View If you want to render the view that corresponds to a different template within the same controller, you can use `render` with the name of the view:

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

If the call to `update` fails, calling the `update` action in this controller will render the `edit.html.erb` template belonging to the same controller.

If you prefer, you can use a symbol instead of a string to specify the action to render:

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

Rendering an Action's Template from Another Controller What if you want to render a template from an entirely different controller from the one that contains the action code? You can also do that with `render`, which accepts the full path (relative to `app/views`) of the template to render. For example, if you're running code in an `AdminProductsController` that lives in `app/controllers/admin`, you can render the results of an action to a template in `app/views/products` this way:

```
render "products/show"
```

Rails knows that this view belongs to a different controller because of the embedded slash character in the string. If you want to be explicit, you can use the `:template` option (which was required on Rails 2.2 and earlier):

```
render template: "products/show"
```

Wrapping it up The above two ways of rendering (rendering the template of another action in the same controller, and rendering the template of another action in a different controller) are actually variants of the same operation.

In fact, in the `BooksController` class, inside of the update action where we want to render the edit template if the book does not update successfully, all of the following render calls would all render the `edit.html.erb` template in the `views/books` directory:

```
render :edit
render action: :edit
render "edit"
render action: "edit"
render "books/edit"
render template: "books/edit"
```

Which one you use is really a matter of style and convention, but the rule of thumb is to use the simplest one that makes sense for the code you are writing.

Using `render` with `:inline` The `render` method can do without a view completely, if you're willing to use the `:inline` option to supply ERB as part of the method call. This is perfectly valid:

```
render inline: "<% products.each do |p| %><p><%= p.name %></p><% end %>"
```

WARNING: There is seldom any good reason to use this option. Mixing ERB into your controllers defeats the MVC orientation of Rails and will make it harder for other developers to follow the logic of your project. Use a separate erb view instead.

By default, inline rendering uses ERB. You can force it to use Builder instead with the `:type` option:

```
render inline: "xml.p {'Horrid coding practice!'}", type: :builder
```

Rendering Text You can send plain text - with no markup at all - back to the browser by using the `:plain` option to `render`:

```
render plain: "OK"
```

TIP: Rendering pure text is most useful when you're responding to Ajax or web service requests that are expecting something other than proper HTML.

NOTE: By default, if you use the `:plain` option, the text is rendered without using the current layout. If you want Rails to put the text into the current layout, you need to add the `layout: true` option and use the `.text.erb` extension for the layout file.

Rendering HTML You can send an HTML string back to the browser by using the `:html` option to `render`:

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

TIP: This is useful when you're rendering a small snippet of HTML code. However, you might want to consider moving it to a template file if the markup is complex.

NOTE: When using `html:` option, HTML entities will be escaped if the string is not composed with `html_safe`-aware APIs.

Rendering JSON JSON is a JavaScript data format used by many Ajax libraries. Rails has built-in support for converting objects to JSON and rendering that JSON back to the browser:

```
render json: @product
```

TIP: You don't need to call `to_json` on the object that you want to render. If you use the `:json` option, `render` will automatically call `to_json` for you.

Rendering XML Rails also has built-in support for converting objects to XML and rendering that XML back to the caller:

```
render xml: @product
```

TIP: You don't need to call `to_xml` on the object that you want to render. If you use the `:xml` option, `render` will automatically call `to_xml` for you.

Rendering Vanilla JavaScript Rails can render vanilla JavaScript:

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

This will send the supplied string to the browser with a MIME type of `text/javascript`.

Rendering raw body You can send a raw content back to the browser, without setting any content type, by using the `:body` option to `render`:

```
render body: "raw"
```

TIP: This option should be used only if you don't care about the content type of the response. Using `:plain` or `:html` might be more appropriate most of the time.

NOTE: Unless overridden, your response returned from this render option will be `text/plain`, as that is the default content type of Action Dispatch response.

Rendering raw file Rails can render a raw file from an absolute path. This is useful for conditionally rendering static files like error pages.

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

This renders the raw file (it doesn't support ERB or other handlers). By default it is rendered within the current layout.

WARNING: Using the `:file` option in combination with users input can lead to security problems since an attacker could use this action to access security sensitive files in your file system.

TIP: `send_file` is often a faster and better option if a layout isn't required.

Rendering objects Rails can render objects responding to `:render_in`.

```
render MyRenderable.new
```

This calls `render_in` on the provided object with the current view context.

Options for `render` Calls to the `render` method generally accept six options:

- `:content_type`
- `:layout`
- `:location`
- `:status`
- `:formats`
- `:variants`

The `:content_type` Option By default, Rails will serve the results of a rendering operation with the MIME content-type of `text/html` (or `application/json` if you use the `:json` option, or `application/xml` for the `:xml` option.). There are times when you might like to change this, and you can do so by setting the `:content_type` option:

```
render template: "feed", content_type: "application/rss"
```

The `:layout` Option With most of the options to `render`, the rendered content is displayed as part of the current layout. You'll learn more about layouts and how to use them later in this guide.

You can use the `:layout` option to tell Rails to use a specific file as the layout for the current action:

```
render layout: "special_layout"
```

You can also tell Rails to render with no layout at all:

```
render layout: false
```

The `:location` Option You can use the `:location` option to set the HTTP `Location` header:

```
render xml: photo, location: photo_url(photo)
```

The `:status` Option Rails will automatically generate a response with the correct HTTP status code (in most cases, this is `200 OK`). You can use the `:status` option to change this:

```
render status: 500
render status: :forbidden
```

Rails understands both numeric status codes and the corresponding symbols shown below.

Response Class	HTTP Status Code	Symbol
Informational	100	:continue
	101	:switching_protocols
	102	:processing
Success	200	:ok
	201	:created
	202	:accepted
	203	:non_authoritative_information
	204	:no_content
	205	:reset_content
	206	:partial_content
	207	:multi_status

Response Class	HTTP Status Code	Symbol
	208	:already_reported
	226	:im_used
Redirection	300	:multiple_choices
	301	:moved_permanently
	302	:found
	303	:see_other
	304	:not_modified
	305	:use_proxy
	307	:temporary_redirect
	308	:permanent_redirect
Client Error	400	:bad_request
	401	:unauthorized
	402	:payment_required
	403	:forbidden
	404	:not_found
	405	:method_not_allowed
	406	:not_acceptable
	407	:proxy_authentication_required
	408	:request_timeout
	409	:conflict
	410	:gone
	411	:length_required
	412	:precondition_failed
	413	:payload_too_large
	414	:uri_too_long
	415	:unsupported_media_type
	416	:range_not_satisfiable
	417	:expectation_failed
	421	:misdirected_request
	422	:unprocessable_entity
	423	:locked
	424	:failed_dependency
	426	:upgrade_required
	428	:precondition_required
	429	:too_many_requests
	431	:request_header_fields_too_large
	451	:unavailable_for_legal_reasons
Server Error	500	:internal_server_error
	501	:not_implemented
	502	:bad_gateway
	503	:service_unavailable
	504	:gateway_timeout
	505	:http_version_not_supported

Response Class	HTTP Status Code	Symbol
	506	:variant_also_negotiates
	507	:insufficient_storage
	508	:loop_detected
	510	:not_extended
	511	:network_authentication_required

NOTE: If you try to render content along with a non-content status code (100-199, 204, 205, or 304), it will be dropped from the response.

The `:formats` Option Rails uses the format specified in the request (or `:html` by default). You can change this passing the `:formats` option with a symbol or an array:

```
render formats: :xml
render formats: [:json, :xml]
```

If a template with the specified format does not exist an `ActionView::MissingTemplate` error is raised.

The `:variants` Option This tells Rails to look for template variations of the same format. You can specify a list of variants by passing the `:variants` option with a symbol or an array.

An example of use would be this.

```
# called in HomeController#index
render variants: [:mobile, :desktop]
```

With this set of variants Rails will look for the following set of templates and use the first that exists.

- `app/views/home/index.html+mobile.erb`
- `app/views/home/index.html+desktop.erb`
- `app/views/home/index.html.erb`

If a template with the specified format does not exist an `ActionView::MissingTemplate` error is raised.

Instead of setting the variant on the render call you may also set it on the request object in your controller action.

```
def index
  request.variant = determine_variant
end

private
```

```
def determine_variant
  variant = nil
  # some code to determine the variant(s) to use
  variant = :mobile if session[:use_mobile]

  variant
end
```

Finding Layouts To find the current layout, Rails first looks for a file in `app/views/layouts` with the same base name as the controller. For example, rendering actions from the `PhotosController` class will use `app/views/layouts/photos.html.erb` (or `app/views/layouts/photos.builder`). If there is no such controller-specific layout, Rails will use `app/views/layouts/application.html.erb` or `app/views/layouts/application.builder`. If there is no `.erb` layout, Rails will use a `.builder` layout if one exists. Rails also provides several ways to more precisely assign specific layouts to individual controllers and actions.

Specifying Layouts for Controllers You can override the default layout conventions in your controllers by using the `layout` declaration. For example:

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

With this declaration, all of the views rendered by the `ProductsController` will use `app/views/layouts/inventory.html.erb` as their layout.

To assign a specific layout for the entire application, use a `layout` declaration in your `ApplicationController` class:

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

With this declaration, all of the views in the entire application will use `app/views/layouts/main.html.erb` for their layout.

Choosing Layouts at Runtime You can use a symbol to defer the choice of layout until a request is processed:

```
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
```

Now, if the current user is a special user, they'll get a special layout when viewing a product.

You can even use an inline method, such as a Proc, to determine the layout. For example, if you pass a Proc object, the block you give the Proc will be given the `controller` instance, so the layout can be determined based on the current request:

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

Conditional Layouts Layouts specified at the controller level support the `:only` and `:except` options. These options take either a method name, or an array of method names, corresponding to method names within the controller:

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

With this declaration, the `product` layout would be used for everything but the `rss` and `index` methods.

Layout Inheritance Layout declarations cascade downward in the hierarchy, and more specific layout declarations always override more general ones. For example:

- `application_controller.rb`

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

- `articles_controller.rb`

  ```
  class ArticlesController < ApplicationController
  end
  ```

- `special_articles_controller.rb`

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

- 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
```

In this application:

- In general, views will be rendered in the `main` layout
- `ArticlesController#index` will use the `main` layout
- `SpecialArticlesController#index` will use the `special` layout
- `OldArticlesController#show` will use no layout at all
- `OldArticlesController#index` will use the `old` layout

Template Inheritance Similar to the Layout Inheritance logic, if a template or partial is not found in the conventional path, the controller will look for a template or partial to render in its inheritance chain. For example:

```
# app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
end

# app/controllers/admin_controller.rb
class AdminController < ApplicationController
end

# app/controllers/admin/products_controller.rb
class Admin::ProductsController < AdminController
  def index
  end
end
```

The lookup order for an `admin/products#index` action will be:

- app/views/admin/products/
- app/views/admin/

- app/views/application/

This makes app/views/application/ a great place for your shared partials, which can then be rendered in your ERB as such:

```
<%# app/views/admin/products/index.html.erb %>
<%= render @products || "empty_list" %>

<%# app/views/application/_empty_list.html.erb %>
There are no items in this list <em>yet</em>.
```

Avoiding Double Render Errors Sooner or later, most Rails developers will see the error message "Can only render or redirect once per action". While this is annoying, it's relatively easy to fix. Usually it happens because of a fundamental misunderstanding of the way that `render` works.

For example, here's some code that will trigger this error:

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

If `@book.special?` evaluates to `true`, Rails will start the rendering process to dump the `@book` variable into the `special_show` view. But this will *not* stop the rest of the code in the `show` action from running, and when Rails hits the end of the action, it will start to render the `regular_show` view - and throw an error. The solution is simple: make sure that you have only one call to `render` or `redirect` in a single code path. One thing that can help is `and return`. Here's a patched version of the method:

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

Make sure to use `and return` instead of `&& return` because `&& return` will not work due to the operator precedence in the Ruby Language.

Note that the implicit render done by ActionController detects if `render` has been called, so the following will work without errors:

```
def show
  @book = Book.find(params[:id])
```

```
  if @book.special?
    render action: "special_show"
  end
end
```

This will render a book with `special?` set with the `special_show` template, while other books will render with the default `show` template.

Using `redirect_to`

Another way to handle returning responses to an HTTP request is with `redirect_to`. As you've seen, `render` tells Rails which view (or other asset) to use in constructing a response. The `redirect_to` method does something completely different: it tells the browser to send a new request for a different URL. For example, you could redirect from wherever you are in your code to the index of photos in your application with this call:

```
redirect_to photos_url
```

You can use `redirect_back` to return the user to the page they just came from. This location is pulled from the `HTTP_REFERER` header which is not guaranteed to be set by the browser, so you must provide the `fallback_location` to use in this case.

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

NOTE: `redirect_to` and `redirect_back` do not halt and return immediately from method execution, but simply set HTTP responses. Statements occurring after them in a method will be executed. You can halt by an explicit `return` or some other halting mechanism, if needed.

Getting a Different Redirect Status Code Rails uses HTTP status code 302, a temporary redirect, when you call `redirect_to`. If you'd like to use a different status code, perhaps 301, a permanent redirect, you can use the `:status` option:

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

Just like the `:status` option for `render`, `:status` for `redirect_to` accepts both numeric and symbolic header designations.

The Difference Between `render` and `redirect_to` Sometimes inexperienced developers think of `redirect_to` as a sort of `goto` command, moving execution from one place to another in your Rails code. This is *not* correct. Your code stops running and waits for a new request from the browser. It just happens that you've told the browser what request it should make next, by sending back an HTTP 302 status code.

Consider these actions to see the difference:

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

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

With the code in this form, there will likely be a problem if the `@book` variable is `nil`. Remember, a `render :action` doesn't run any code in the target action, so nothing will set up the `@books` variable that the `index` view will probably require. One way to fix this is to redirect instead of rendering:

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

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

With this code, the browser will make a fresh request for the index page, the code in the `index` method will run, and all will be well.

The only downside to this code is that it requires a round trip to the browser: the browser requested the show action with `/books/1` and the controller finds that there are no books, so the controller sends out a 302 redirect response to the browser telling it to go to `/books/`, the browser complies and sends a new request back to the controller asking now for the `index` action, the controller then gets all the books in the database and renders the index template, sending it back down to the browser which then shows it on your screen.

While in a small application, this added latency might not be a problem, it is something to think about if response time is a concern. We can demonstrate one way to handle this with a contrived example:

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

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

```
    @books = Book.all
    flash.now[:alert] = "Your book was not found"
    render "index"
  end
end
```

This would detect that there are no books with the specified ID, populate the `@books` instance variable with all the books in the model, and then directly render the `index.html.erb` template, returning it to the browser with a flash alert message to tell the user what happened.

Using `head` To Build Header-Only Responses

The `head` method can be used to send responses with only headers to the browser. The `head` method accepts a number or symbol (see reference table) representing an HTTP status code. The options argument is interpreted as a hash of header names and values. For example, you can return only an error header:

```
head :bad_request
```

This would produce the following header:

```
HTTP/1.1 400 Bad Request
Connection: close
Date: Sun, 24 Jan 2010 12:15:53 GMT
Transfer-Encoding: chunked
Content-Type: text/html; charset=utf-8
X-Runtime: 0.013483
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

Or you can use other HTTP headers to convey other information:

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

Which would produce:

```
HTTP/1.1 201 Created
Connection: close
Date: Sun, 24 Jan 2010 12:16:44 GMT
Transfer-Encoding: chunked
Location: /photos/1
Content-Type: text/html; charset=utf-8
X-Runtime: 0.083496
Set-Cookie: _blog_session=...snip...; path=/; HttpOnly
Cache-Control: no-cache
```

Structuring Layouts

When Rails renders a view as a response, it does so by combining the view with the current layout, using the rules for finding the current layout that were covered earlier in this guide. Within a layout, you have access to three tools for combining different bits of output to form the overall response:

- Asset tags
- `yield` and `content_for`
- Partials

Asset Tag Helpers

Asset tag helpers provide methods for generating HTML that link views to feeds, JavaScript, stylesheets, images, videos, and audios. There are six asset tag helpers available in Rails:

- `auto_discovery_link_tag`
- `javascript_include_tag`
- `stylesheet_link_tag`
- `image_tag`
- `video_tag`
- `audio_tag`

You can use these tags in layouts or other views, although the `auto_discovery_link_tag`, `javascript_include_tag`, and `stylesheet_link_tag`, are most commonly used in the `<head>` section of a layout.

WARNING: The asset tag helpers do *not* verify the existence of the assets at the specified locations; they simply assume that you know what you're doing and generate the link.

Linking to Feeds with the `auto_discovery_link_tag` The `auto_discovery_link_tag` helper builds HTML that most browsers and feed readers can use to detect the presence of RSS, Atom, or JSON feeds. It takes the type of the link (`:rss`, `:atom`, or `:json`), a hash of options that are passed through to url_for, and a hash of options for the tag:

```
<%= auto_discovery_link_tag(:rss, {action: "feed"},
  {title: "RSS Feed"}) %>
```

There are three tag options available for the `auto_discovery_link_tag`:

- `:rel` specifies the `rel` value in the link. The default value is "alternate".
- `:type` specifies an explicit MIME type. Rails will generate an appropriate MIME type automatically.
- `:title` specifies the title of the link. The default value is the uppercase `:type` value, for example, "ATOM" or "RSS".

Linking to JavaScript Files with the `javascript_include_tag` The `javascript_include_tag` helper returns an HTML `script` tag for each source provided.

If you are using Rails with the Asset Pipeline enabled, this helper will generate a link to `/assets/javascripts/` rather than `public/javascripts` which was used in earlier versions of Rails. This link is then served by the asset pipeline.

A JavaScript file within a Rails application or Rails engine goes in one of three locations: `app/assets`, `lib/assets` or `vendor/assets`. These locations are explained in detail in the Asset Organization section in the Asset Pipeline Guide.

You can specify a full path relative to the document root, or a URL, if you prefer. For example, to link to a JavaScript file that is inside a directory called `javascripts` inside of one of `app/assets`, `lib/assets` or `vendor/assets`, you would do this:

```
<%= javascript_include_tag "main" %>
```

Rails will then output a `script` tag such as this:

```
<script src='/assets/main.js'></script>
```

The request to this asset is then served by the Sprockets gem.

To include multiple files such as `app/assets/javascripts/main.js` and `app/assets/javascripts/columns.js` at the same time:

```
<%= javascript_include_tag "main", "columns" %>
```

To include `app/assets/javascripts/main.js` and `app/assets/javascripts/photos/columns.js`:

```
<%= javascript_include_tag "main", "/photos/columns" %>
```

To include `http://example.com/main.js`:

```
<%= javascript_include_tag "http://example.com/main.js" %>
```

Linking to CSS Files with the `stylesheet_link_tag` The `stylesheet_link_tag` helper returns an HTML `<link>` tag for each source provided.

If you are using Rails with the "Asset Pipeline" enabled, this helper will generate a link to `/assets/stylesheets/`. This link is then processed by the Sprockets gem. A stylesheet file can be stored in one of three locations: `app/assets`, `lib/assets`, or `vendor/assets`.

You can specify a full path relative to the document root, or a URL. For example, to link to a stylesheet file that is inside a directory called `stylesheets` inside of one of `app/assets`, `lib/assets`, or `vendor/assets`, you would do this:

```
<%= stylesheet_link_tag "main" %>
```

To include `app/assets/stylesheets/main.css` and `app/assets/stylesheets/columns.css`:

```
<%= stylesheet_link_tag "main", "columns" %>
```

To include app/assets/stylesheets/main.css and app/assets/stylesheets/photos/columns.css:

```
<%= stylesheet_link_tag "main", "photos/columns" %>
```

To include http://example.com/main.css:

```
<%= stylesheet_link_tag "http://example.com/main.css" %>
```

By default, the stylesheet_link_tag creates links with rel="stylesheet". You can override this default by specifying an appropriate option (:rel):

```
<%= stylesheet_link_tag "main_print", media: "print" %>
```

Linking to Images with the image_tag The image_tag helper builds an HTML <img /> tag to the specified file. By default, files are loaded from public/images.

WARNING: Note that you must specify the extension of the image.

```
<%= image_tag "header.png" %>
```

You can supply a path to the image if you like:

```
<%= image_tag "icons/delete.gif" %>
```

You can supply a hash of additional HTML options:

```
<%= image_tag "icons/delete.gif", {height: 45} %>
```

You can supply alternate text for the image which will be used if the user has images turned off in their browser. If you do not specify an alt text explicitly, it defaults to the file name of the file, capitalized and with no extension. For example, these two image tags would return the same code:

```
<%= image_tag "home.gif" %>
<%= image_tag "home.gif", alt: "Home" %>
```

You can also specify a special size tag, in the format "{width}x{height}":

```
<%= image_tag "home.gif", size: "50x20" %>
```

In addition to the above special tags, you can supply a final hash of standard HTML options, such as :class, :id, or :name:

```
<%= image_tag "home.gif", alt: "Go Home",
                          id: "HomeImage",
                          class: "nav_bar" %>
```

Linking to Videos with the video_tag The video_tag helper builds an HTML5 <video> tag to the specified file. By default, files are loaded from public/videos.

```
<%= video_tag "movie.ogg" %>
```

Produces

```
<video src="/videos/movie.ogg" />
```

Like an `image_tag` you can supply a path, either absolute, or relative to the `public/videos` directory. Additionally you can specify the `size: "#{width}x#{height}"` option just like an `image_tag`. Video tags can also have any of the HTML options specified at the end (`id`, `class` et al).

The video tag also supports all of the `<video>` HTML options through the HTML options hash, including:

- `poster: "image_name.png"`, provides an image to put in place of the video before it starts playing.
- `autoplay: true`, starts playing the video on page load.
- `loop: true`, loops the video once it gets to the end.
- `controls: true`, provides browser supplied controls for the user to interact with the video.
- `autobuffer: true`, the video will pre load the file for the user on page load.

You can also specify multiple videos to play by passing an array of videos to the `video_tag`:

```
<%= video_tag ["trailer.ogg", "movie.ogg"] %>
```

This will produce:

```
<video>
  <source src="/videos/trailer.ogg">
  <source src="/videos/movie.ogg">
</video>
```

Linking to Audio Files with the `audio_tag` The `audio_tag` helper builds an HTML5 `<audio>` tag to the specified file. By default, files are loaded from `public/audios`.

```
<%= audio_tag "music.mp3" %>
```

You can supply a path to the audio file if you like:

```
<%= audio_tag "music/first_song.mp3" %>
```

You can also supply a hash of additional options, such as `:id`, `:class`, etc.

Like the `video_tag`, the `audio_tag` has special options:

- `autoplay: true`, starts playing the audio on page load
- `controls: true`, provides browser supplied controls for the user to interact with the audio.
- `autobuffer: true`, the audio will pre load the file for the user on page load.

Understanding `yield`

Within the context of a layout, `yield` identifies a section where content from the view should be inserted. The simplest way to use this is to have a single `yield`, into which the entire contents of the view currently being rendered is inserted:

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

You can also create a layout with multiple yielding regions:

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

The main body of the view will always render into the unnamed `yield`. To render content into a named `yield`, you use the `content_for` method.

Using the `content_for` Method

The `content_for` method allows you to insert content into a named `yield` block in your layout. For example, this view would work with the layout that you just saw:

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

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

The result of rendering this page into the supplied layout would be this HTML:

```
<html>
  <head>
  <title>A simple page</title>
  </head>
  <body>
  <p>Hello, Rails!</p>
  </body>
</html>
```

The `content_for` method is very helpful when your layout contains distinct regions such as sidebars and footers that should get their own blocks of content inserted. It's also useful for inserting tags that load page-specific JavaScript or CSS files into the header of an otherwise generic layout.

Using Partials

Partial templates - usually just called "partials" - are another device for breaking the rendering process into more manageable chunks. With a partial, you can move the code for rendering a particular piece of a response to its own file.

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 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: as 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 looked like this:

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

<h1>Products</h1>

<p>Here are a few of our fine products:</p>
...

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

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

As seen in the previous sections of this guide, `yield` is a very powerful tool for cleaning up your layouts. Keep in mind that it's pure Ruby, so you can use it almost everywhere. For example, we can use it to DRY up form layout definitions for several similar resources:

- `users/index.html.erb`

```
<%= render "shared/search_filters", search: @q do |form| %>
  <p>
    Name contains: <%= form.text_field :name_contains %>
  </p>
<% end %>
```

- `roles/index.html.erb`

```
<%= render "shared/search_filters", search: @q do |form| %>
  <p>
    Title contains: <%= form.text_field :title_contains %>
  </p>
<% end %>
```

- `shared/_search_filters.html.erb`

```
<%= form_with model: search do |form| %>
  <h1>Search form:</h1>
  <fieldset>
    <%= yield form %>
  </fieldset>
  <p>
    <%= form.submit "Search" %>
  </p>
<% end %>
```

TIP: For content that is shared among all pages in your application, you can use partials directly from layouts.

Partial Layouts A partial can use its own layout file, just as a view can use a layout. For example, you might call a partial like this:

```
<%= render partial: "link_area", layout: "graybar" %>
```

This would look for a partial named `_link_area.html.erb` and render it using the layout `_graybar.html.erb`. Note that layouts for partials follow the same leading-underscore naming as regular partials, and are placed in the same folder with the partial that they belong to (not in the master `layouts` folder).

Also note that explicitly specifying `:partial` is required when passing additional options such as `:layout`.

Passing Local Variables You can also pass local variables into partials, making them even more powerful and flexible. For example, you can use this technique to reduce duplication between new and edit pages, while still keeping a bit of distinct content:

- `new.html.erb`

```
<h1>New zone</h1>
<%= render partial: "form", locals: {zone: @zone} %>
```

- `edit.html.erb`

```
<h1>Editing zone</h1>
<%= render partial: "form", locals: {zone: @zone} %>
```

- `_form.html.erb`

```
<%= form_with model: zone do |form| %>
  <p>
    <b>Zone name</b><br>
    <%= form.text_field :name %>
  </p>
  <p>
    <%= form.submit %>
  </p>
<% end %>
```

Although the same partial will be rendered into both views, Action View's submit helper will return "Create Zone" for the new action and "Update Zone" for the edit action.

To pass a local variable to a partial in only specific cases use the `local_assigns`.

- `index.html.erb`

```
<%= render user.articles %>
```

- `show.html.erb`

```
<%= render article, full: true %>
```

- `_article.html.erb`

```
<h2><%= article.title %></h2>

<% if local_assigns[:full] %>
  <%= simple_format article.body %>
<% else %>
  <%= truncate article.body %>
<% end %>
```

This way it is possible to use the partial without the need to declare all local variables.

Every partial also has a local variable with the same name as the partial (minus the leading underscore). You can pass an object in to this local variable via the `:object` option:

```
<%= render partial: "customer", object: @new_customer %>
```

Within the `customer` partial, the `customer` variable will refer to `@new_customer` from the parent view.

If you have an instance of a model to render into a partial, you can use a shorthand syntax:

```
<%= render @customer %>
```

Assuming that the `@customer` instance variable contains an instance of the `Customer` model, this will use `_customer.html.erb` to render it and will pass the local variable `customer` into the partial which will refer to the `@customer` instance variable in the parent view.

Rendering Collections Partials are very useful in rendering collections. When you pass a collection to a partial via the `:collection` option, the partial will be inserted once for each member in the collection:

- `index.html.erb`

  ```
  <h1>Products</h1>
  <%= render partial: "product", collection: @products %>
  ```

- `_product.html.erb`

  ```
  <p>Product Name: <%= product.name %></p>
  ```

When a partial is called with a pluralized collection, then 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 the `_product` partial, you can refer to `product` to get the instance that is being rendered.

There is also a shorthand for this. Assuming `@products` is a collection of `Product` instances, you can simply write this in the `index.html.erb` to produce the same result:

```
<h1>Products</h1>
<%= render @products %>
```

Rails determines the name of the partial to use by looking at the model name in the collection. In fact, you can even create a heterogeneous collection and render it this way, and Rails will choose the proper partial for each member of the collection:

- `index.html.erb`

  ```
  <h1>Contacts</h1>
  <%= render [customer1, employee1, customer2, employee2] %>
  ```

- `customers/_customer.html.erb`

  ```
  <p>Customer: <%= customer.name %></p>
  ```

- `employees/_employee.html.erb`

  ```
  <p>Employee: <%= employee.name %></p>
  ```

In this case, Rails will use the customer or employee partials as appropriate for each member of the collection.

In the event that the collection is empty, `render` will return nil, so it should be fairly simple to provide alternative content.

```
<h1>Products</h1>
<%= render(@products) || "There are no products available." %>
```

Local Variables To use a custom local variable name within the partial, specify the `:as` option in the call to the partial:

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

With this change, you can access an instance of the `@products` collection as the `item` local variable within the partial.

You can also pass in arbitrary local variables to any partial you are rendering with the `locals: {}` option:

```
<%= render partial: "product", collection: @products,
           as: :item, locals: {title: "Products Page"} %>
```

In this case, the partial will have access to a local variable `title` with the value "Products Page".

TIP: Rails also makes a counter variable available within a partial called by the collection, named after the title of the partial followed by `_counter`. For example, when rendering a collection `@products` the partial `_product.html.erb` can access the variable `product_counter` which indexes the number of times it has been rendered within the enclosing view. Note that it also applies for when the partial name was changed by using the `as:` option. For example, the counter variable for the code above would be `item_counter`.

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

Spacer Templates

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

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

Collection Partial Layouts When rendering collections it is also possible to use the `:layout` option:

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

The layout will be rendered together with the partial for each item in the collection. The current object and object_counter variables will be available in the layout as well, the same way they are within the partial.

Using Nested Layouts

You may find that your application requires a layout that differs slightly from your regular application layout to support one particular controller. Rather than repeating the main layout and editing it, you can accomplish this by using nested layouts (sometimes called sub-templates). Here's an example:

Suppose you have the following `ApplicationController` layout:

- `app/views/layouts/application.html.erb`

```
<html>
<head>
  <title><%= @page_title or "Page Title" %></title>
  <%= stylesheet_link_tag "layout" %>
  <style><%= yield :stylesheets %></style>
</head>
<body>
  <div id="top_menu">Top menu items here</div>
  <div id="menu">Menu items here</div>
  <div id="content"><%= content_for?(:content) ? yield(:content) : yield %></div>
</body>
</html>
```

On pages generated by `NewsController`, you want to hide the top menu and add a right menu:

- `app/views/layouts/news.html.erb`

```
<% 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" %>
```

That's it. The News views will use the new layout, hiding the top menu and adding a new right menu inside the "content" div.

There are several ways of getting similar results with different sub-templating schemes using this technique. Note that there is no limit in nesting levels. One can use the `ActionView::render` method via `render template: 'layouts/news'`

to base a new layout on the News layout. If you are sure you will not subtemplate the `News` layout, you can replace the `content_for?(:news_content) ? yield(:news_content) : yield` with simply `yield`.