In this chapter, you'll see:

- Templates
- Forms including fields and uploading files
- Helpers
- Layouts and partials

CHAPTER 22

# Action View

We've seen how the routing component determines which controller to use and how the controller chooses an action. We've seen how the controller and action between them decide what to render to the user. Normally, rendering takes place at the end of the action and involves a template. That's what this chapter is all about. Action View encapsulates all the functionality needed to render templates, most commonly generating HTML, XML, or JavaScript back to the user. As its name suggests, Action View is the view part of our MVC trilogy.

In this chapter, we'll start with templates, for which Rails provides a range of options. We will then cover a number of ways in which users provide input: forms, file uploads, and links. We'll complete this chapter by looking at a number of ways to reduce maintenance using helpers, layouts, and partials.

## Using Templates

When you write a view, you're writing a template: something that will get expanded to generate the final result. To understand how these templates work, we need to look at three areas:

- Where the templates go
- The environment they run in
- What goes inside them

### Where Templates Go

The render() method expects to find templates in the app/views directory of the current application. Within this directory, the convention is to have a separate subdirectory for the views of each controller. Our Depot application, for instance, includes products and store controllers. As a result, our application has templates in app/views/products and app/views/store. Each directory typically contains templates named after the actions in the corresponding controller.

You can also have templates that aren't named after actions. You render such templates from the controller using calls such as these:

```
render(action:   'fake_action_name')
render(template: 'controller/name')
render(file:     'dir/template')
```

The last of these allows you to store templates anywhere on your filesystem. This is useful if you want to share templates across applications.

## The Template Environment

Templates contain a mixture of fixed text and code. The code in the template adds dynamic content to the response. That code runs in an environment that gives it access to the information set up by the controller:

- All instance variables of the controller are also available in the template. This is how actions communicate data to the templates.

- The controller object's flash, headers, logger, params, request, response, and session are available as accessor methods in the view. Apart from the flash, view code probably shouldn't use these directly, because the responsibility for handling them should rest with the controller. However, we do find this useful when debugging. For example, the following html.erb template uses the debug() method to display the contents of the session, the details of the parameters, and the current response:

  ```
  <h4>Session</h4>  <%= debug(session) %>
  <h4>Params</h4>   <%= debug(params) %>
  <h4>Response</h4> <%= debug(response) %>
  ```

- The current controller object is accessible using the attribute named controller. This allows the template to call any public method in the controller (including the methods in ActionController::Base).

- The path to the base directory of the templates is stored in the attribute base_path.

## What Goes in a Template

Out of the box, Rails supports two types of templates:

- ERB templates are a mixture of content and embedded Ruby. They're typically used to generate HTML pages.

- Jbuilder[1] templates generate JSON responses.

1. https://github.com/rails/jbuilder

By far, the one that you'll be using the most will be ERB. In fact, you made extensive use of ERB templates in developing the Depot application.

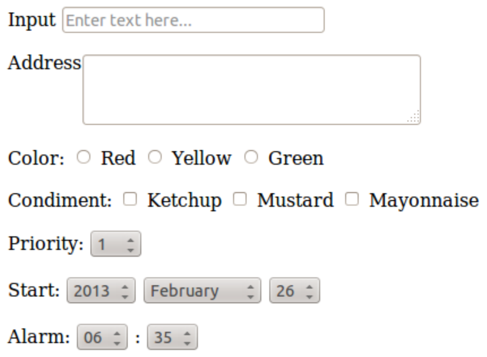
So far in this chapter, we've focused on producing output. In Chapter 21, Action Dispatch and Action Controller, on page 337, we focused on processing input. In a well-designed application, these two are not unrelated: the output we produce contains forms, links, and buttons that guide the end user to producing the next set of inputs. As you might expect by now, Rails provides a considerable amount of help in this area too.

## Generating Forms

HTML provides a number of elements, attributes, and attribute values that control how input is gathered. You certainly could hand-code your form directly into the template, but there's no need to.

In this section, we'll cover a number of *helpers* that Rails provides that assist with this process. In Using Helpers, on page 379, we'll show you how you can create your own helpers.

HTML provides a number of ways to collect data in forms. A few of the more common means are shown in the following screenshot. Note that the form itself isn't representative of any sort of typical use; in general, you'll use only a subset of these methods to collect data.



Let's look at the template that was used to produce that form:

rails7/views/app/views/form/input.html.erb

```
<%= form_for(:model) do |form| %>
<p>
  <%= form.label :input %>
  <%= form.text_field :input, :placeholder => 'Enter text here...' %>
</p>

<p>
  <%= form.label :address, :style => 'float: left' %>
```

```
 -      <%= form.text_area :address, :rows => 3, :cols => 40 %>
10    </p>
 -
 -    <p>
 -      <%= form.label :color %>:
 -      <%= form.radio_button :color, 'red' %>
15      <%= form.label :red %>
 -      <%= form.radio_button :color, 'yellow' %>
 -      <%= form.label :yellow %>
 -      <%= form.radio_button :color, 'green' %>
 -      <%= form.label :green %>
20    </p>
 -
 -    <p>
 -      <%= form.label 'condiment' %>:
 -      <%= form.check_box :ketchup %>
25      <%= form.label :ketchup %>
 -      <%= form.check_box :mustard %>
 -      <%= form.label :mustard %>
 -      <%= form.check_box :mayonnaise %>
 -      <%= form.label :mayonnaise %>
30    </p>
 -
 -    <p>
 -      <%= form.label :priority %>:
 -      <%= form.select :priority, (1..10) %>
35    </p>
 -
 -    <p>
 -      <%= form.label :start %>:
 -      <%= form.date_select :start %>
40    </p>
 -
 -    <p>
 -      <%= form.label :alarm %>:
 -      <%= form.time_select :alarm %>
45    </p>
 -    <% end %>
```

In that template, you'll see a number of labels, such as the one on line 3. You use labels to associate text with an input field for a specified attribute. The text of the label will default to the attribute name unless you specify it explicitly.

You use the text_field() and text_area() helpers (on lines 4 and 9, respectively) to gather single-line and multiline input fields. You may specify a placeholder, which will be displayed inside the field until the user provides a value. Not every browser supports this function, but those that don't simply will display an empty box. Since this will degrade gracefully, there's no need for you to design to the least common denominator—make use of this feature, because those who can see it will benefit from it immediately.

Placeholders are one of the many small "fit and finish" features provided with HTML5, and once again, Rails is ready even if the browser your users have installed is not. You can use the search_field(), telephone_field(), url_field(), email_field(), number_field(), and range_field() helpers to prompt for a specific type of input. How the browser will make use of this information varies. Some may display the field slightly differently to more clearly identify its function. Safari on Mac, for example, will display search fields with rounded corners and will insert a little x for clearing the field once data entry begins. Some may provide added validation. For example, Opera will validate URL fields prior to submission. The iPad will even adjust the virtual onscreen keyboard to provide ready access to characters such as the @ sign when entering an email address.

Although the support for these functions varies by browser, those that don't provide extra support for these functions simply display a plain, unadorned input box. Once again, nothing is gained by waiting. If you have an input field that's expected to contain an email address, don't simply use text_field()—go ahead and start using email_field() now.

Lines 14, 24, and 34 demonstrate three different ways to provide a constrained set of options. Although the display may vary a bit from browser to browser, these approaches are all well supported across all browsers. The select() method is particularly flexible—it can be passed an Enumeration as shown here, an array of pairs of name-value pairs, or a Hash. A number of form options helpers[2] are available to produce such lists from various sources, including the database.

Finally, lines 39 and 44 show prompts for a date and time, respectively. As you might expect by now, Rails provides plenty of options here too.[3]

Not shown in this example are hidden_field() and password_field(). A hidden field is not displayed at all, but the value is passed back to the server. This may be useful as an alternative to storing transient data in sessions, enabling data from one request to be passed onto the next. Password fields are displayed, but the text entered in them is obscured.

2. http://api.rubyonrails.org/classes/ActionView/Helpers/FormOptionsHelper.html
3. http://api.rubyonrails.org/classes/ActionView/Helpers/DateHelper.html

This is more than an adequate starter set for most needs. Should you find that you have additional needs, you're likely to find a helper or gem is already available for you. A good place to start is with the Rails Guides.[4]

Meanwhile, let's explore how the data form's submit is processed.

## Processing Forms

In the figure on page 375 we can see how the various attributes in the model pass through the controller to the view, on to the HTML page, and back again into the model. The model object has attributes such as name, country, and password. The template uses helper methods to construct an HTML form to let the user edit the data in the model. Note how the form fields are named. The country attribute, for example, maps to an HTML input field with the name user[country].

When the user submits the form, the raw POST data is sent back to our application. Rails extracts the fields from the form and constructs the params hash. Simple values (such as the id field, extracted by routing from the form action) are stored directly in the hash. But if a parameter name has brackets in it, Rails assumes that it is part of more structured data and constructs a hash to hold the values. Inside this hash, the string inside the brackets acts as the key. This process can repeat if a parameter name has multiple sets of brackets in it.

| Form Parameters | Params |
|---|---|
| id=123 | { id: "123" } |
| user[name]=Dave | { user: { name: "Dave" }} |
| user[address][city]=Wien | { user: { address: { city: "Wien" }}} |

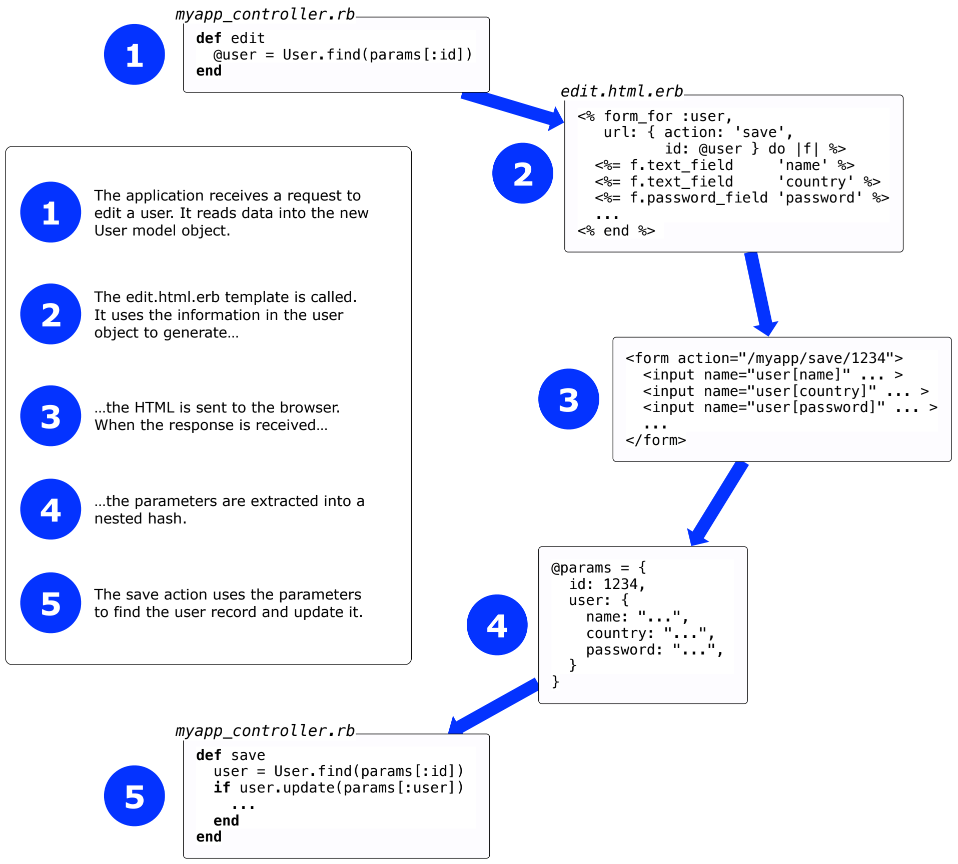
In the final part of the integrated whole, model objects can accept new attribute values from hashes, which allows us to say this:

```
user.update(user_params)
```

Rails integration goes deeper than this. Looking at the .html.erb file in the preceding figure, we can see that the template uses a set of helper methods to create the form's HTML; these are methods such as form_with() and text_field().

Before moving on, it's worth noting that params may be used for more than text. Entire files can be uploaded. We'll cover that next.

4. http://guides.rubyonrails.org/form_helpers.html



1 The application receives a request to edit a user. It reads data into the new User model object.

2 The edit.html.erb template is called. It uses the information in the user object to generate…

3 …the HTML is sent to the browser. When the response is received…

4 …the parameters are extracted into a nested hash.

5 The save action uses the parameters to find the user record and update it.

*myapp_controller.rb*
```
def edit
  @user = User.find(params[:id])
end
```

*edit.html.erb*
```
<% form_for :user,
   url: { action: 'save',
          id: @user } do |f| %>
  <%= f.text_field     'name' %>
  <%= f.text_field     'country' %>
  <%= f.password_field 'password' %>
  ...
<% end %>
```

```
<form action="/myapp/save/1234">
  <input name="user[name]" ... >
  <input name="user[country]" ... >
  <input name="user[password]" ... >
  ...
</form>
```

```
@params = {
  id: 1234,
  user: {
    name: "...",
    country: "...",
    password: "...",
  }
}
```

*myapp_controller.rb*
```
def save
  user = User.find(params[:id])
  if user.update(params[:user])
    ...
  end
end
```

## Uploading Files to Rails Applications

Your application may allow users to upload files. For example, a bug-reporting system might let users attach log files and code samples to a problem ticket, or a blogging application could let its users upload a small image to appear next to their articles.

In HTTP, files are uploaded as a *multipart/form-data* POST message. As the name suggests, forms are used to generate this type of message. Within that form, you'll use <input> tags with type="file". When rendered by a browser, this allows the user to select a file by name. When the form is subsequently submitted, the file or files will be sent back along with the rest of the form data.

To illustrate the file upload process, we'll show some code that allows a user to upload an image and display that image alongside a comment. To do this, we first need a pictures table to store the data:

rails7/e1/views/db/migrate/20170425000004_create_pictures.rb

```ruby
class CreatePictures < ActiveRecord::Migration
  def change
    create_table :pictures do |t|
      t.string :comment
      t.string :name
      t.string :content_type
      # If using MySQL, blobs default to 64k, so we have to give
      # an explicit size to extend them
      t.binary :data, :limit => 1.megabyte
    end
  end
end
```

We'll create a somewhat artificial upload controller just to demonstrate the process. The get action is pretty conventional; it simply creates a new picture object and renders a form:

rails7/e1/views/app/controllers/upload_controller.rb

```ruby
class UploadController < ApplicationController
  def get
    @picture = Picture.new
  end
  # . . .
  private
    # Never trust parameters from the scary internet, only allow the white
    # list through.
    def picture_params
      params.require(:picture).permit(:comment, :uploaded_picture)
    end
end
```

The get template contains the form that uploads the picture (along with a comment). Note how we override the encoding type to allow data to be sent back with the response:

rails7/e1/views/app/views/upload/get.html.erb

```erb
<%= form_for(:picture,
             url: {action: 'save'},
             html: {multipart: true}) do |form| %>

    Comment:             <%= form.text_field("comment") %><br/>
    Upload your picture: <%= form.file_field("uploaded_picture") %><br/>

    <%= submit_tag("Upload file") %>
<% end %>
```

The form has one other subtlety. The picture uploads into an attribute called uploaded_picture. However, the database table doesn't contain a column of that name. That means that there must be some magic happening in the model:

rails7/e1/views/app/models/picture.rb
```
class Picture < ActiveRecord::Base

  validates_format_of :content_type,
                      with: /\Aimage/,
                      message: "must be a picture"

  def uploaded_picture=(picture_field)
    self.name         = base_part_of(picture_field.original_filename)
    self.content_type = picture_field.content_type.chomp
    self.data         = picture_field.read
  end

  def base_part_of(file_name)
    File.basename(file_name).gsub(/[^\w._-]/, '')
  end
end
```

We define an accessor called uploaded_picture=() to receive the file uploaded by the form. The object returned by the form is an interesting hybrid. It's file-like, so we can read its contents with the read() method; that's how we get the image data into the data column. It also has the attributes content_type and original_filename, which let us get at the uploaded file's metadata. Accessor methods pick all this apart, resulting in a single object stored as separate attributes in the database.

Note that we also add a validation to check that the content type is of the form image/*xxx*. We don't want someone uploading JavaScript.

The save action in the controller is totally conventional:

rails7/e1/views/app/controllers/upload_controller.rb
```
def save
  @picture = Picture.new(picture_params)
  if @picture.save
    redirect_to(action: 'show', id: @picture.id)
  else
    render(action: :get)
  end
end
```

Now that we have an image in the database, how do we display it? One way is to give it its own URL and link to that URL from an image tag. For example, we could use a URL such as upload/picture/123 to return the image for picture 123. This would use send_data() to return the image to the browser. Note how we set the content type and filename—this lets browsers interpret the data and supplies a default name should the user choose to save the image:

rails7/e1/views/app/controllers/upload_controller.rb

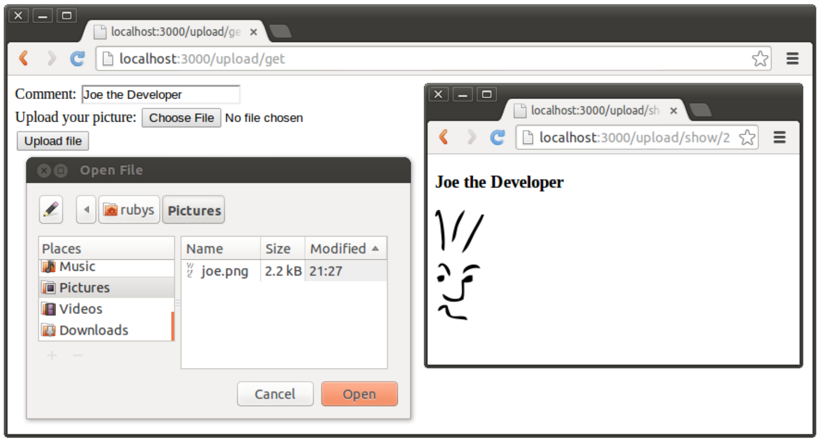
```ruby
def picture
  @picture = Picture.find(params[:id])
  send_data(@picture.data,
            filename: @picture.name,
            type: @picture.content_type,
            disposition: "inline")
end
```

Finally, we can implement the show action, which displays the comment and the image. The action simply loads the picture model object:

rails7/e1/views/app/controllers/upload_controller.rb

```ruby
def show
  @picture = Picture.find(params[:id])
end
```

In the template, the image tag links back to the action that returns the picture content. In the following screenshot, we can see the get and show actions.



rails7/e1/views/app/views/upload/show.html.erb

```erb
<h3><%= @picture.comment %></h3>

<img src="<%= url_for(:action => 'picture', :id => @picture.id) %>"/>
```

If you'd like an easier way of dealing with uploading and storing images, take a look at Active Storage,[5] which we used in Chapter 16, Task K: Receive Emails and Respond with Rich Text, on page 247.

Forms and uploads are just two examples of helpers that Rails provides. Next we'll show you how you can provide your own helpers and introduce you to a number of other helpers that come with Rails.

5. https://edgeguides.rubyonrails.org/active_storage_overview.html

## Using Helpers

Earlier we said it's OK to put code in templates. Now we're going to modify that statement. It's perfectly acceptable to put *some* code in templates—that's what makes them dynamic. However, it's poor style to put too much code in templates.

Three main reasons for this stand out. First, the more code you put in the view side of your application, the easier it is to let discipline slip and start adding application-level functionality to the template code. This is definitely poor form; you want to put application stuff in the controller and model layers so that it's available everywhere. This will pay off when you add new ways of viewing the application.

The second reason is that html.erb is basically HTML. When you edit it, you're editing an HTML file. If you have the luxury of having professional designers create your layouts, they'll want to work with HTML. Putting a bunch of Ruby code in there just makes it hard to work with.

The final reason is that code embedded in views is hard to test, whereas code split out into helper modules can be isolated and tested as individual units.

Rails provides a nice compromise in the form of helpers. A *helper* is simply a module containing methods that assist a view. Helper methods are output-centric. They exist to generate HTML (or XML, or JavaScript)—a helper extends the behavior of a template.

### Your Own Helpers

By default, each controller gets its own helper module. Additionally, there's an application-wide helper named application_helper.rb. It won't be surprising to learn that Rails makes certain assumptions to help link the helpers into the controller and its views. While all view helpers are available to all controllers, it's often good practice to organize helpers. Helpers that are unique to the views associated with the ProductController tend to be placed in a helper module called ProductHelper in the file product_helper.rb in the app/helpers directory. You don't have to remember all these details—the rails generate controller script creates a stub helper module automatically.

We can use helpers to clean up the application layout a bit. Currently we have the following:

```
<h3><%= @page_title || "Pragmatic Store" %></h3>
```

Let's move the code that works out the page title into a helper method. Because we're in the store controller, we edit the store_helper.rb file in app/helpers:

```
module StoreHelper
  def page_title
    @page_title || "Pragmatic Store"
  end
end
```

Now the view code simply calls the helper method:

```
<h3><%= page_title %></h3>
```

(We might want to eliminate even more duplication by moving the rendering of the entire title into a separate partial template, shared by all the controller's views, but we don't talk about partial templates until Partial-Page Templates, on page 390.)

## Helpers for Formatting and Linking

Rails comes with a bunch of built-in helper methods, available to all views. Here, we'll touch on the highlights, but you'll probably want to look at the Action View RDoc for the specifics—there's a lot of functionality in there.

Aside from the general convenience these helpers provide, many of them also handle internationalization and localization. In Chapter 15, Task J: Internationalization, on page 225, we translated much of the application. Many of the helpers we used handled that for us, such as number_to_currency(). It's always a good practice to use Rails helpers where they're appropriate, even if it seems just as easy to hard-code the output you want.

### Formatting Helpers

One set of helper methods deals with dates, numbers, and text:

```
<%= distance_of_time_in_words(Time.now, Time.local(2016, 12, 25)) %>
```

4 months

```
<%= distance_of_time_in_words(Time.now, Time.now + 33, include_seconds: false) %>
```

1 minute

```
<%= distance_of_time_in_words(Time.now, Time.now + 33, include_seconds: true) %>
```

Half a minute

```
<%= time_ago_in_words(Time.local(2012, 12, 25)) %>
```

7 months

```
<%= number_to_currency(123.45) %>
```

$123.45

*<%= number_to_currency(234.56, unit: "CAN$", precision: 0) %>*
: CAN$235

*<%= number_to_human_size(123_456) %>*
: 120.6 KB

*<%= number_to_percentage(66.66666) %>*
: 66.667%

*<%= number_to_percentage(66.66666, precision: 1) %>*
: 66.7%

*<%= number_to_phone(2125551212) %>*
: 212-555-1212

*<%= number_to_phone(2125551212, area_code: true, delimiter: " ") %>*
: (212) 555 1212

*<%= number_with_delimiter(12345678) %>*
: 12,345,678

*<%= number_with_delimiter(12345678, delimiter: "_") %>*
: 12_345_678

*<%= number_with_precision(50.0/3, precision: 2) %>*
: 16.67

The `debug()` method dumps out its parameter using YAML and escapes the result so it can be displayed in an HTML page. This can help when trying to look at the values in model objects or request parameters:

```
<%= debug(params) %>
```

```
--- !ruby/hash:HashWithIndifferentAccess
name: Dave
language: Ruby
action: objects
controller: test
```

Yet another set of helpers deals with text, using methods to truncate strings and highlight words in a string:

*<%= simple_format(@trees) %>*
: Formats a string, honoring line and paragraph breaks. You could give it the plain text of the Joyce Kilmer poem *Trees*,[6] and it would add the HTML to format it as follows.

6. https://www.poetryfoundation.org/poetrymagazine/poems/12744/trees

<p> I think that I shall never see <br />A poem lovely as a tree.</p> <p>A tree whose hungry mouth is prest <br />Against the sweet earth's flowing breast; </p>

*<%= excerpt(@trees, "lovely", 8) %>*

...A poem lovely as a tre...

*<%= highlight(@trees, "tree") %>*

I think that I shall never see A poem lovely as a <strong class="highlight">tree</strong>. A <strong class="highlight">tree</strong> whose hungry mouth is prest Against the sweet earth's flowing breast;

*<%= truncate(@trees, length: 20) %>*

I think that I sh...

There's a method to pluralize nouns:

*<%= pluralize(1, "person") %> but <%= pluralize(2, "person") %>*

1 person but 2 people

If you'd like to do what the fancy websites do and automatically hyperlink URLs and email addresses, there are helpers to do that. Another one strips hyperlinks from text.

Back in Iteration A2 on page 80, we saw how the cycle() helper can be used to return the successive values from a sequence each time it's called, repeating the sequence as necessary. This is often used to create alternating styles for the rows in a table or list. The current_cycle() and reset_cycle() methods are also available.

Finally, if you're writing something like a blog site or you're allowing users to add comments to your store, you could offer them the ability to create their text in Markdown (BlueCloth)[7] or Textile (RedCloth)[8] format. These are formatters that take text written in human-friendly markup and convert it into HTML.

## Linking to Other Pages and Resources

The ActionView::Helpers::AssetTagHelper and ActionView::Helpers::UrlHelper modules contain a number of methods that let you reference resources external to the current template. Of these, the most commonly used is link_to(), which creates a hyperlink to another action in your application:

```
<%= link_to "Add Comment", new_comments_path %>
```

7. https://github.com/rtomayko/rdiscount
8. http://redcloth.org/

The first parameter to link_to() is the text displayed for the link. The next is a string or hash specifying the link's target.

An optional third parameter provides HTML attributes for the generated link:

```
<%= link_to "Delete", product_path(@product),
    { class: "dangerous", method: 'delete' }
%>
```

This third parameter also supports two additional options that modify the behavior of the link. Each requires JavaScript to be enabled in the browser.

The :method option is a hack—it allows you to make the link look to the application as if the request were created by a POST, PUT, PATCH, or DELETE, rather than the normal GET method. This is done by creating a chunk of JavaScript that submits the request when the link is clicked—if JavaScript is disabled in the browser, a GET will be generated.

The :data parameter allows you to set custom data attributes. The most commonly used one is the :confirm option, which takes a short message. If present, an unobtrusive JavaScript driver will display the message and get the user's confirmation before the link is followed:

```
<%= link_to "Delete", product_path(@product),
                      method: :delete,
                      data: { confirm: 'Are you sure?' }
%>
```

The button_to() method works the same as link_to() but generates a button in a self-contained form rather than a straight hyperlink. This is the preferred method of linking to actions that have side effects. However, these buttons live in their own forms, which imposes a couple of restrictions: they cannot appear inline, and they cannot appear inside other forms.

Rails has conditional linking methods that generate hyperlinks if some condition is met or just return the link text otherwise. link_to_if() and link_to_unless() take a condition parameter, followed by the regular parameters to link_to. If the condition is true (for link_to_if) or false (for link_to_unless), a regular link will be created using the remaining parameters. If not, the name will be added as plain text (with no hyperlink).

The link_to_unless_current() helper creates menus in sidebars where the current page name is shown as plain text and the other entries are hyperlinks:

```
<ul>
<% %w{ create list edit save logout }.each do |action| %>
   <li>
     <%= link_to_unless_current(action.capitalize, action: action) %>
   </li>
<% end %>
</ul>
```

The link_to_unless_current() helper may also be passed a block that's evaluated only if the current action is the action given, effectively providing an alternative to the link. There's also a current_page() helper method that simply tests whether the current page was generated by the given options.

As with url_for(), link_to() and friends also support absolute URLs:

```
<%= link_to("Help", "http://my.site/help/index.html") %>
```

The image_tag() helper creates <img> tags. Optional :size parameters (of the form *widthxheight*) or separate width and height parameters define the size of the image:

```
<%= image_tag("/assets/dave.png", class: "bevel", size: "80x120") %>
<%= image_tag("/assets/andy.png", class: "bevel",
              width: "80", height: "120") %>
```

If you don't give an :alt option, Rails synthesizes one for you using the image's filename. If the image path doesn't start with a / character, Rails assumes that it lives under the app/assets/images directory.

You can make images into links by combining link_to() and image_tag():

```
<%= link_to(image_tag("delete.png", size: "50x22"),
            product_path(@product),
            data: { confirm: "Are you sure?" },
            method: :delete)
%>
```

The mail_to() helper creates a mailto: hyperlink that, when clicked, normally loads the client's email application. It takes an email address, the name of the link, and a set of HTML options. Within these options, you can also use :bcc, :cc, :body, and :subject to initialize the corresponding email fields. Finally, the magic option encode: "javascript" uses client-side JavaScript to obscure the generated link, making it harder for spiders to harvest email addresses from your site. Unfortunately, it also means your users won't see the email link if they have JavaScript disabled in their browsers.

```
<%= mail_to("support@pragprog.com", "Contact Support",
            subject: "Support question from #{@user.name}",
            encode:  "javascript") %>
```

As a weaker form of obfuscation, you can use the :replace_at and :replace_dot options to replace the at sign and dots in the displayed name with other strings. This is unlikely to fool harvesters.

The AssetTagHelper module also includes helpers that make it easy to link to style sheets and JavaScript code from your pages and to create autodiscovery Atom feed links. We created links in the layouts for the Depot application using the stylesheet_link_tag() and javascript_importmap_tags() methods in the head:

**rails7/depot_r/app/views/layouts/application.html.erb**

```
<!DOCTYPE html>
<html>
  <head>
    <title>Pragprog Books Online Store</title>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>
    <%= stylesheet_link_tag "inter-font", "data-turbo-track": "reload" %>
    <%= stylesheet_link_tag "tailwind", "data-turbo-track": "reload" %>

    <%= stylesheet_link_tag "application", "data-turbo-track": "reload" %>
    <%= javascript_importmap_tags %>
  </head>
```

The javascript_importmap_tags() method produces a list JavaScript filenames (assumed to live in app/javascript) which enables these resources to be imported by your application.

By default, image and style sheet assets are assumed to live in the images and stylesheets directories relative to the application's assets directory. If the path given to an asset tag method starts with a forward slash, then the path is assumed to be absolute and no prefix is applied. Sometimes it makes sense to move this static content onto a separate box or to different locations on the current box. Do this by setting the configuration variable asset_host:

```
config.action_controller.asset_host = "http://media.my.url/assets"
```

Although this list of helpers may seem to be comprehensive, Rails provides many more; new helpers are introduced with each release, and a select few are retired or moved off into a plugin where they can be evolved at a different pace than Rails.

## Reducing Maintenance with Layouts and Partials

So far in this chapter we've looked at templates as isolated chunks of code and HTML. But one of the driving ideas behind Rails is honoring the DRY principle and eliminating the need for duplication. The average website, though, has lots of duplication:

- Many pages share the same tops, tails, and sidebars.
- Multiple pages may contain the same snippets of rendered HTML (a blog site, for example, may display an article in multiple places).
- The same functionality may appear in multiple places. Many sites have a standard search component or a polling component that appears in most of the sites' sidebars.

Rails provides both layouts and partials that reduce the need for duplication in these three situations.

## Layouts

Rails allows you to render pages that are nested inside other rendered pages. Typically this feature is used to put the content from an action within a standard site-wide page frame (title, footer, and sidebar). In fact, if you've been using the generate script to create scaffold-based applications, then you've been using these layouts all along.

When Rails honors a request to render a template from within a controller, it actually renders two templates. Obviously, it renders the one you ask for (or the default template named after the action if you don't explicitly render anything). But Rails also tries to find and render a layout template (we'll talk about how it finds the layout in a second). If it finds the layout, it inserts the action-specific output into the HTML produced by the layout.

Let's look at a layout template:

```
<html>
  <head>
    <title>Form: <%= controller.action_name %></title>
    <%= stylesheet_link_tag 'scaffold' %>
  </head>
  <body>

    <%= yield :layout %>

  </body>
</html>
```

The layout sets out a standard HTML page, with the head and body sections. It uses the current action name as the page title and includes a CSS file. In the body is a call to yield. This is where the magic takes place. When the template for the action was rendered, Rails stored its content, labeling it :layout.

Inside the layout template, calling yield retrieves this text. In fact, :layout is the default content returned when rendering, so you can write yield instead of yield :layout. We personally prefer the slightly more explicit version.

Suppose the my_action.html.erb template contained this:

```
<h1><%= @msg %></h1>
```

And also suppose the controller set @msg to Hello, World!. Then the browser would see the following HTML:

```
<html>
  <head>
    <title>Form: my_action</title>
    <link href="/stylesheets/scaffold.css" media="screen"
          rel="Stylesheet" type="text/css" />
  </head>
  <body>

    <h1>Hello, World!</h1>

  </body>
</html>
```

## Locating Layout Files

As you've probably come to expect, Rails does a good job of providing defaults for layout file locations, but you can override the defaults if you need something different.

Layouts are controller-specific. If the current request is being handled by a controller called *store*, Rails will by default look for a layout called store (with the usual .html.erb or .xml.builder extension) in the app/views/layouts directory. If you create a layout called application in the layouts directory, it will be applied to all controllers that don't otherwise have a layout defined for them.

You can override this using the layout declaration inside a controller. The most basic invocation is to pass it the name of a layout as a string. The following declaration will make the template in the file standard.html.erb or standard.xml.builder the layout for all actions in the store controller.

The layout file will be looked for in the app/views/layouts directory:

```
class StoreController < ApplicationController

  layout "standard"

  # ...
end
```

You can qualify which actions will have the layout applied to them using the :only and :except qualifiers:

```
class StoreController < ApplicationController

  layout "standard", except: [ :rss, :atom ]

  # ...
end
```

Specifying a layout of nil turns off layouts for a controller.

Sometimes you need to change the appearance of a set of pages at runtime. For example, a blogging site might offer a different-looking side menu if the user is logged in, or a store site might have different-looking pages if the site is down for maintenance. Rails supports this need with dynamic layouts. If the parameter to the layout declaration is a symbol, it's taken to be the name of a controller instance method that returns the name of the layout to be used:

```
class StoreController < ApplicationController

  layout :determine_layout
  # ...
  private

  def determine_layout
    if Store.is_closed?
      "store_down"
    else
      "standard"
    end
  end
end
```

Subclasses of a controller use the parent's layout unless they override it using the layout directive. Finally, individual actions can choose to render using a specific layout (or with no layout at all) by passing render() the :layout option:

```
def rss
  render(layout: false)   # never use a layout
end
def checkout
  render(layout: "layouts/simple")
end
```

## Passing Data to Layouts

Layouts have access to all the same data that's available to conventional templates. In addition, any instance variables set in the normal template will be available in the layout (because the regular template is rendered before the layout is invoked). This might be used to parameterize headings or menus in the layout. For example, the layout might contain this:

```
<html>
  <head>
    <title><%= @title %></title>
    <%= stylesheet_link_tag 'scaffold' %>
  </head>
  <body>
    <h1><%= @title %></h1>
    <%= yield :layout %>
  </body>
</html>
```

An individual template could set the title by assigning to the @title variable:

```
<% @title = "My Wonderful Life" %>
<p>
  Dear Diary:
</p>
<p>
  Yesterday I had pizza for dinner. It was nice.
</p>
```

We can take this further. The same mechanism that lets us use yield :layout to embed the rendering of a template into the layout also lets you generate arbitrary content in a template, which can then be embedded into any template.

For example, different templates may need to add their own template-specific items to the standard page sidebar. We'll use the content_for mechanism in those templates to define content and then use yield in the layout to embed this content into the sidebar.

In each regular template, use a content_for to give a name to the content rendered inside a block. This content will be stored inside Rails and will not contribute to the output generated by the template:

```
<h1>Regular Template</h1>

<% content_for(:sidebar) do %>
  <ul>
    <li>this text will be rendered</li>
    <li>and saved for later</li>
    <li>it may contain <%= "dynamic" %> stuff</li>
  </ul>
<% end %>
<p>
  Here's the regular stuff that will appear on
  the page rendered by this template.
</p>
```

Then, in the layout, use yield :sidebar to include this block in the page's sidebar:

```
<!DOCTYPE .... >
<html>
  <body>
    <div class="sidebar">
      <p>
        Regular sidebar stuff
      </p>
      <div class="page-specific-sidebar">
➤       <%= yield :sidebar %>
      </div>
    </div>
  </body>
</html>
```

This same technique can be used to add page-specific JavaScript functions into the <head> section of a layout, create specialized menu bars, and so on.

## Partial-Page Templates

Web applications commonly display information about the same application object or objects on multiple pages. A shopping cart might display an order line item on the shopping cart page and again on the order summary page. A blog application might display the contents of an article on the main index page and again at the top of a page soliciting comments. Typically this would involve copying snippets of code between the different template pages.

Rails, however, eliminates this duplication with the *partial-page templates* (more frequently called *partials*). You can think of a partial as a kind of subroutine. You invoke it one or more times from within another template, potentially passing it objects to render as parameters. When the partial template finishes rendering, it returns control to the calling template.

Internally, a partial template looks like any other template. Externally, there's a slight difference. The name of the file containing the template code must start with an underscore character, differentiating the source of partial templates from their more complete brothers and sisters.

For example, the partial to render a blog entry might be stored in the file _article.html.erb in the normal views directory, app/views/blog:

```
<div class="article">
  <div class="articleheader">
    <h3><%= article.title %></h3>
  </div>
  <div class="articlebody">
    <%= article.body %>
  </div>
</div>
```

Other templates use the render(partial:) method to invoke this:

```
<%= render(partial: "article", object: @an_article) %>
<h3>Add Comment</h3>
. . .
```

The :partial parameter to render() is the name of the template to render (but without the leading underscore). This name must be both a valid filename and a valid Ruby identifier (so a-b and 20042501 are not valid names for partials). The :object parameter identifies an object to be passed into the partial. This object will be available within the template via a local variable with the same name as the template. In this example, the @an_article object will be passed to the template, and the template can access it using the local variable article. That's why we could write things such as article.title in the partial.

You can set additional local variables in the template by passing render() a :locals parameter. This takes a hash where the entries represent the names and values of the local variables to set:

```
render(partial: 'article',
       object:  @an_article,
       locals:  { authorized_by: session[:user_name],
                  from_ip:       request.remote_ip })
```

### Partials and Collections

Applications commonly need to display collections of formatted entries. A blog might show a series of articles, each with text, author, date, and so on. A store might display entries in a catalog, where each has an image, a description, and a price.

The :collection parameter to render() works in conjunction with the :partial parameter. The :partial parameter lets us use a partial to define the format of an individual entry, and the :collection parameter applies this template to each member of the collection.

To display a list of article model objects using our previously defined _article.html.erb partial, we could write this:

```
<%= render(partial: "article", collection: @article_list) %>
```

Inside the partial, the local variable article will be set to the current article from the collection—the variable is named after the template. In addition, the variable article_counter will have its value set to the index of the current article in the collection.

The optional :spacer_template parameter lets you specify a template that will be rendered between each of the elements in the collection. For example, a view might contain the following:

rails7/e1/views/app/views/partial/_list.html.erb
```
<%= render(partial:         "animal",
           collection:      %w{ ant bee cat dog elk },
           spacer_template: "spacer")
%>
```

This uses _animal.html.erb to render each animal in the given list, rendering the partial _spacer.html.erb between each. Say _animal.html.erb contains this:

rails7/e1/views/app/views/partial/_animal.html.erb
```
<p>The animal is <%= animal %></p>
```

And _spacer.html.erb contains this:

rails7/e1/views/app/views/partial/_spacer.html.erb
```
<hr />
```

Your users would see a list of animal names with a line between each.

## Shared Templates

If the first option or :partial parameter to a render call is a String with no slashes, Rails assumes that the target template is in the current controller's view directory. However, if the name contains one or more / characters, Rails assumes that the part up to the last slash is a directory name and the rest is the template name. The directory is assumed to be under app/views. This makes it easy to share partials and subtemplates across controllers.

The convention among Rails applications is to store these shared partials in a subdirectory of app/views called shared. Render shared partials using statements such as these:

```
<%= render("shared/header", locals: {title: @article.title}) %>
<%= render(partial: "shared/post", object: @article) %>
. . .
```

In this previous example, the @article object will be assigned to the local variable post within the template.

## Partials with Layouts

Partials can be rendered with a layout, and you can apply a layout to a block within any template:

```
<%= render partial: "user", layout: "administrator" %>

<%= render layout: "administrator" do %>
  # ...
<% end %>
```

Partial layouts are to be found directly in the app/views directory associated with the controller along with the customary underbar prefix, such as app/views/users/_administrator.html.erb.

### Partials and Controllers

It isn't just view templates that use partials. Controllers also get in on the act. Partials give controllers the ability to generate fragments from a page using the same partial template as the view. This is particularly important when you're using Ajax support to update just part of a page from the controller—use partials, and you know your formatting for the table row or line item that you're updating will be compatible with that used to generate its brethren initially.

Taken together, partials and layouts provide an effective way to make sure that the user interface portion of your application is maintainable. But being maintainable is only part of the story; doing so in a way that also performs well is also crucial.

## What We Just Did

Views are the public face of Rails applications, and we've seen that Rails delivers extensive support for what you need to build robust and maintainable user and application programming interfaces.

We started with templates, of which Rails provides built-in support for three types: ERB, Builder, and SCSS. Templates make it easy for us to provide HTML, JSON, XML, CSS, and JavaScript responses to any request. We'll discuss adding another option in Creating HTML Templates with Slim, on page 418.

We dove into forms, which are the primary means by which users will interact with your application. Along the way, we covered uploading files.

We continued with helpers, which enable us to factor out complex application logic to allow our views to focus on presentation aspects. We explored a number of helpers that Rails provides, ranging from basic formatting to hypertext links, which are the final way in which users interact with HTML pages.

We completed our tour of Action View by covering two related ways of factoring out large chunks of content for reuse. We use layouts to factor out the outermost layers of a view and provide a common look and feel. We use partials to factor out common inner components, such as a single form or table.

That covers how a user with a browser will access our Rails application. Next up: covering how we define and maintain the schema of the database our application will use to store data.