In this chapter, you'll see:

- Representational State Transfer (REST)
- Defining how requests are routed to controllers
- Selecting a data representation
- Testing routes
- The controller environment
- Rendering and redirecting
- Sessions, flash, and callbacks

CHAPTER 21

Action Dispatch and Action Controller

Action Pack lies at the heart of Rails applications. It consists of three Ruby modules: ActionDispatch, ActionController, and ActionView. Action Dispatch routes requests to controllers. Action Controller converts requests into responses. Action View is used by Action Controller to format those responses.

As a concrete example, in the Depot application, we routed the root of the site (/) to the index() method of the StoreController. At the completion of that method, the template in app/views/store/index.html.erb was rendered. Each of these activities was orchestrated by modules in the Action Pack component.

Working together, these three submodules provide support for processing incoming requests and generating outgoing responses. In this chapter, we'll look at both Action Dispatch and Action Controller. In the next chapter, we'll cover Action View.

When we looked at Active Record, we saw it could be used as a freestanding library; we can use Active Record as part of a nonweb Ruby application. Action Pack is different. Although it's possible to use it directly as a framework, you probably won't. Instead, you'll take advantage of the tight integration offered by Rails. Components such as Action Controller, Action View, and Active Record handle the processing of requests, and the Rails environment knits them together into a coherent (and easy-to-use) whole. For that reason, we'll describe Action Controller in the context of Rails. Let's start by looking at how Rails applications handle requests. We'll then dive down into the details of routing and URL handling. We'll continue by looking at how you write code in a controller. Finally, we'll cover sessions, flash, and callbacks.

Dispatching Requests to Controllers

At its most basic, a web application accepts an incoming request from a browser, processes it, and sends a response.

A question immediately springs to mind: how does the application know what to do with the incoming request? A shopping cart application will receive requests to display a catalog, add items to a cart, create an order, and so on. How does it route these requests to the appropriate code?

It turns out that Rails provides two ways to define how to route a request: a comprehensive way that you'll use when you need to and a convenient way that you'll generally use whenever you can.

The comprehensive way lets you define a direct mapping of URLs to actions based on pattern matching, requirements, and conditions. The convenient way lets you define routes based on resources, such as the models that you define. And because the convenient way is built on the comprehensive way, you can freely mix and match the two approaches.

In both cases, Rails encodes information in the request URL and uses a subsystem called Action Dispatch to determine what should be done with that request. The actual process is flexible, but at the end of it Rails has determined the name of the *controller* that handles this particular request along with a list of any other request parameters. In the process, either one of these additional parameters or the HTTP method itself is used to identify the *action* to be invoked in the target controller.

Rails routes support the mapping between URLs and actions based on the contents of the URL and on the HTTP method used to invoke the request. We've seen how to do this on a URL-by-URL basis using anonymous or named routes. Rails also supports a higher-level way of creating groups of related routes. To understand the motivation for this, we need to take a little diversion into the world of representational state transfer (REST).

REST: Representational State Transfer

The ideas behind REST were formalized in Chapter 5 of Roy Fielding's 2000 PhD dissertation.[1] In a REST approach, servers communicate with clients using stateless connections. All the information about the state of the interaction between the two is encoded into the requests and responses between them. Long-term state is kept on the server as a set of identifiable *resources*.

1. http://www.ics.uci.edu/~fielding/pubs/dissertation/rest_arch_style.htm

Clients access these resources using a well-defined (and severely constrained) set of resource identifiers (URLs in our context). REST distinguishes the content of resources from the presentation of that content. REST is designed to support highly scalable computing while constraining application architectures to be decoupled by nature.

This description contains a lot of abstract stuff. What does REST mean in practice?

First, the formalities of a RESTful approach mean that network designers know when and where they can cache responses to requests. This enables load to be pushed out through the network, increasing performance and resilience while reducing latency.

Second, the constraints imposed by REST can lead to easier-to-write (and maintain) applications. RESTful applications don't worry about implementing remotely accessible services. Instead, they provide a regular (and straightforward) interface to a set of resources. Your application implements a way of listing, creating, editing, and deleting each resource, and your clients do the rest.

Let's make this more concrete. In REST, we use a basic set of verbs to operate on a rich set of nouns. If we're using HTTP, the verbs correspond to HTTP methods (`GET`, `PUT`, `PATCH`, `POST`, and `DELETE`, typically). The nouns are the resources in our application. We name those resources using URLs.

The Depot application that we produced contained a set of products. There are implicitly two resources here: first, the individual products, each of which constitutes a resource, and second, the collection of products.

To fetch a list of all the products, we could issue an HTTP `GET` request against this collection, say on the path `/products`. To fetch the contents of an individual resource, we have to identify it. The Rails way would be to give its primary key value (that is, its ID). Again we'd issue a `GET` request, this time against the URL `/products/1`.

To create a new product in our collection, we use an HTTP `POST` request directed at the `/products` path, with the post data containing the product to add. Yes, that's the same path we used to get a list of products. If you issue a `GET` to it, it responds with a list, and if you do a `POST` to it, it adds a new product to the collection.

Take this a step further. We've already seen you can retrieve the content of a product—you just issue a `GET` request against the path `/products/1`. To update that product, you'd issue an HTTP `PUT` request against the same URL. And, to delete it, you could issue an HTTP `DELETE` request, using the same URL.

Take this further. Maybe our system also tracks users. Again, we have a set of resources to deal with. REST tells us to use the same set of verbs (GET, POST, PATCH, PUT, and DELETE) against a similar-looking set of URLs (/users, /users/1, and so on).

Now we see some of the power of the constraints imposed by REST. We're already familiar with the way Rails constrains us to structure our applications a certain way. Now the REST philosophy tells us to structure the interface to our applications too. Suddenly our world gets a lot simpler.

Rails has direct support for this type of interface; it adds a kind of macro route facility, called *resources*. Let's take a look at how the config/routes.rb file might have looked back in Creating a Rails Application, on page 69:

```
Depot::Application.routes.draw do
  resources :products
end
```

The resources line caused seven new routes to be added to our application. Along the way, it assumed that the application will have a controller named ProductsController, containing seven actions with given names.

You can take a look at the routes that were generated for us. We do this by making use of the handy rails routes command.

```
      Prefix Verb   URI Pattern
                    Controller#Action
    products GET    /products(.:format)
                    {:action=>"index", :controller=>"products"}
             POST   /products(.:format)
                    {:action=>"create", :controller=>"products"}
 new_product GET    /products/new(.:format)
                    {:action=>"new", :controller=>"products"}
edit_product GET    /products/:id/edit(.:format)
                    {:action=>"edit", :controller=>"products"}
     product GET    /products/:id(.:format)
                    {:action=>"show", :controller=>"products"}
             PATCH  /products/:id(.:format)
                    {:action=>"update", :controller=>"products"}
             DELETE /products/:id(.:format)
                    {:action=>"destroy", :controller=>"products"}
```

All the routes defined are spelled out in a columnar format. The lines will generally wrap on your screen; in fact, they had to be broken into two lines per route to fit on this page. The columns are (optional) route name, HTTP method, route path, and (on a separate line on this page) route requirements.

Fields in parentheses are optional parts of the path. Field names preceded by a colon are for variables into which the matching part of the path is placed for later processing by the controller.

Now let's look at the seven controller actions that these routes reference. Although we created our routes to manage the products in our application, let's broaden this to talk about resources—after all, the same seven methods will be required for all resource-based routes:

index
: Returns a list of the resources.

create
: Creates a new resource from the data in the POST request, adding it to the collection.

new
: Constructs a new resource and passes it to the client. This resource will not have been saved on the server. You can think of the new action as creating an empty form for the client to fill in.

show
: Returns the contents of the resource identified by params[:id].

update
: Updates the contents of the resource identified by params[:id] with the data associated with the request.

edit
: Returns the contents of the resource identified by params[:id] in a form suitable for editing.

destroy
: Destroys the resource identified by params[:id].

You can see that these seven actions contain the four basic CRUD operations (create, read, update, and delete). They also contain an action to list resources and two auxiliary actions that return new and existing resources in a form suitable for editing on the client.

If for some reason you don't need or want all seven actions, you can limit the actions produced using :only or :except options on your resources:

```
resources :comments, except: [:update, :destroy]
```

Several of the routes are named routes enabling you to use helper functions such as products_url and edit_product_url(id:1).

Note that each route is defined with an optional format specifier. We'll cover formats in more detail in Selecting a Data Representation, on page 347.

Let's take a look at the controller code:

rails7/depot_a/app/controllers/products_controller.rb

```
class ProductsController < ApplicationController
  before_action :set_product, only: %i[ show edit update destroy ]

  # GET /products or /products.json
  def index
    @products = Product.all
  end

  # GET /products/1 or /products/1.json
  def show
  end

  # GET /products/new
  def new
    @product = Product.new
  end

  # GET /products/1/edit
  def edit
  end

  # POST /products or /products.json
  def create
    @product = Product.new(product_params)

    respond_to do |format|
      if @product.save
        format.html { redirect_to product_url(@product),
          notice: "Product was successfully created." }
        format.json { render :show, status: :created,
          location: @product }
      else
        format.html { render :new,
          status: :unprocessable_entity }
        format.json { render json: @product.errors,
          status: :unprocessable_entity }
      end
    end
  end

  # PATCH/PUT /products/1 or /products/1.json
  def update
    respond_to do |format|
      if @product.update(product_params)
        format.html { redirect_to product_url(@product),
          notice: "Product was successfully updated." }
        format.json { render :show, status: :ok, location: @product }
      else
```

```
        format.html { render :edit,
          status: :unprocessable_entity }
        format.json { render json: @product.errors,
          status: :unprocessable_entity }
      end
    end
  end

  # DELETE /products/1 or /products/1.json
  def destroy
    @product.destroy

    respond_to do |format|
      format.html { redirect_to products_url,
          notice: "Product was successfully destroyed." }
      format.json { head :no_content }
    end
  end

  private
    # Use callbacks to share common setup or constraints between actions.
    def set_product
      @product = Product.find(params[:id])
    end

    # Only allow a list of trusted parameters through.
    def product_params
      params.require(:product).
        permit(:title, :description, :image_url, :price)
    end
end
```

Notice how we have one action for each of the RESTful actions. The comment before each shows the format of the URL that invokes it.

Notice also that many of the actions contain a `respond_to()` block. As we saw in Chapter 11, Task F: Hotwiring the Storefront, on page 143, Rails uses this to determine the type of content to send in a response. The scaffold generator automatically creates code that will respond appropriately to requests for HTML or JSON content. We'll play with that in a little while.

The views created by the generator are fairly straightforward. The only tricky thing is the need to use the correct HTTP method to send requests to the server.

For example, the view for the `index` action looks like this:

rails7/depot_a/app/views/products/index.html.erb

```
<div class="w-full">
  <% if notice.present? %>
    <p class="py-2 px-3 bg-green-50 mb-5 text-green-500 font-medium
              rounded-lg inline-block" id="notice">
      <%= notice %>
    </p>
  <% end %>

  <div class="flex justify-between items-center pb-8">
    <h1 class="mx-auto text-lg font-bold text-4xl">Products</h1>
  </div>

  <table id="products" class="mx-auto">
    <tfoot>
      <tr>
        <td colspan="3">
          <div class="mt-8">
            <%= link_to 'New product',
                        new_product_path,
                        class: "inline rounded-lg py-3 px-5 bg-green-600
                                text-white block font-medium" %>
          </div>
        </td>
      </tr>
    </tfoot>

    <tbody>
      <% @products.each do |product| %>
        <tr class="<%= cycle('bg-green-50', 'bg-white') %>">

          <td class="px-2 py-3">
            <%= image_tag(product.image_url, class: 'w-40') %>
          </td>

          <td>
            <h1 class="text-xl font-bold mb-3"><%= product.title %></h1>
            <p>
              <%= truncate(strip_tags(product.description),
                           length: 80) %>
            </p>
          </td>

          <td class="px-3">
            <ul>
              <li>
                <%= link_to 'Show',
                            product,
                            class: 'hover:underline' %>
              </li>
```

```
            <li>
              <%= link_to 'Edit',
                          edit_product_path(product),
                          class: 'hover:underline' %>
            </li>

            <li>
              <%= link_to 'Destroy',
                          product,
                          class: 'hover:underline',
                          data: { turbo_method: :delete,
                                  turbo_confirm: "Are you sure?" } %>
            </li>
          </ul>
        </td>
      </tr>
    <% end %>
  </tbody>
</table>
</div>
```

The links to the actions that edit a product and add a new product should both use regular GET methods, so a standard link_to works fine. However, the request to destroy a product must issue an HTTP DELETE, so the call includes the method: :delete option to link_to.

Adding Additional Actions

Rails resources provide you with an initial set of actions, but you don't need to stop there. For example, if you want to add an interface to allow people to fetch a list of people who bought any given product, you can add an extension to the resources call:

```
Depot::Application.routes.draw do
  resources :products do
    get :who_bought, on: :member
  end
end
```

That syntax is straightforward. It says "We want to add a new action named who_bought, invoked via an HTTP GET. It applies to each member of the collection of products."

Instead of specifying :member, if we instead specified :collection, then the route would apply to the collection as a whole. This is often used for scoping; for example, you may have collections of products on clearance or products that have been discontinued.

Nested Resources

Often our resources themselves contain additional collections of resources. For example, we may want to allow folks to review our products. Each review would be a resource, and collections of reviews would be associated with each product resource. Rails provides a convenient and intuitive way of declaring the routes for this type of situation:

```
resources :products do
  resources :reviews
end
```

This defines the top-level set of product routes and additionally creates a set of subroutes for reviews. Because the review resources appear inside the products block, a review resource *must* be qualified by a product resource. This means that the path to a review must always be prefixed by the path to a particular product. To fetch the review with ID 4 for the product with an ID of 99, you'd use a path of /products/99/reviews/4.

The named route for /products/:product_id/reviews/:id is product_review, not simply review. This naming simply reflects the nesting of these resources.

As always, you can see the full set of routes generated by our configuration by using the rails routes command.

Routing Concerns

So far, we've been dealing with a fairly small set of resources. On a larger system there may be types of objects for which a review may be appropriate or to which a who_bought action might reasonably be applied. Instead of repeating these instructions for each resource, consider refactoring your routes using concerns to capture the common behavior.

```
concern :reviewable do
  resources :reviews
end

resources :products, concern: :reviewable
resources :users, concern: :reviewable
```

The preceding definition of the products resource is equivalent to the one in the previous section.

Shallow Route Nesting

At times, nested resources can produce cumbersome URLs. A solution to this is to use shallow route nesting:

```
resources :products, shallow: true do
  resources :reviews
end
```

This will enable the recognition of the following routes:

```
/products/1         => product_path(1)
/products/1/reviews => product_reviews_index_path(1)
/reviews/2          => reviews_path(2)
```

Try the rails routes command to see the full mapping.

Selecting a Data Representation

One of the goals of a REST architecture is to decouple data from its representation. If a human uses the URL path /products to fetch products, they should see nicely formatted HTML. If an application asks for the same URL, it could elect to receive the results in a code-friendly format (YAML, JSON, or XML, perhaps).

We've already seen how Rails can use the HTTP Accept header in a respond_to block in the controller. However, it isn't always easy (and sometimes it's plain impossible) to set the Accept header. To deal with this, Rails allows you to pass the format of response you'd like as part of the URL. As you've seen, Rails accomplishes this by including a field called :format in your route definitions. To do this, set a :format parameter in your routes to the file extension of the MIME type you'd like returned:

```
GET    /products(.:format)
       {:action=>"index", :controller=>"products"}
```

Because a full stop (period) is a separator character in route definitions, :format is treated as just another field. Because we give it a nil default value, it's an optional field.

Having done this, we can use a respond_to() block in our controllers to select our response type depending on the requested format:

```
def show
  respond_to do |format|
    format.html
    format.json { render json: @product.to_json }
  end
end
```

Given this, a request to /store/show/1 or /store/show/1.html will return HTML content, while /store/show/1.xml will return XML, and /store/show/1.json will return JSON. You can also pass the format in as an HTTP request parameter:

```
GET HTTP://pragprog.com/store/show/123?format=xml
```

Although the idea of having a single controller that responds with different content types seems appealing, the reality is tricky. In particular, it turns out that error handling can be tough. Although it's acceptable on error to redirect a user to a form, showing them a nice flash message, you have to adopt a different strategy when you serve XML. Consider your application architecture carefully before deciding to bundle all your processing into single controllers.

Rails makes it straightforward to develop an application that's based on resource-based routing. Many claim it greatly simplifies the coding of their applications. However, it isn't always appropriate. Don't feel compelled to use it if you can't find a way of making it work. And you can always mix and match. Some controllers can be resource based, and others can be based on actions. Some controllers can even be resource based with a few extra actions.

Processing of Requests

In the previous section, we worked out how Action Dispatch routes an incoming request to the appropriate code in your application. Now let's see what happens inside that code.

Action Methods

When a controller object processes a request, it looks for a public instance method with the same name as the incoming action. If it finds one, that method is invoked. If it doesn't find one and the controller implements `method_missing()`, that method is called, passing in the action name as the first parameter and an empty argument list as the second. If no method can be called, the controller looks for a template named after the current controller and action. If found, this template is rendered directly. If none of these things happens, an `AbstractController::ActionNotFound` error is generated.

Controller Environment

The controller sets up the environment for actions (and, by extension, for the views that they invoke). Many of these methods provide direct access to information contained in the URL or request:

action_name
: The name of the action currently being processed.

cookies
: The cookies associated with the request. Setting values into this object stores cookies on the browser when the response is sent. Rails support for sessions is based on cookies. We discuss sessions in Rails Sessions, on page 360.

headers

A hash of HTTP headers that will be used in the response. By default, Cache-Control is set to no-cache. You might want to set Content-Type headers for special-purpose applications. Note that you shouldn't set cookie values in the header directly—use the cookie API to do this.

params

A hash-like object containing request parameters (along with pseudoparameters generated during routing). It's hash-like because you can index entries using either a symbol or a string—params[:id] and params['id'] return the same value. Idiomatic Rails applications use the symbol form.

request

The incoming request object. It includes these attributes:

- request_method returns the request method, one of :delete, :get, :head, :post, or :put.
- method returns the same value as request_method except for :head, which it returns as :get because these two are functionally equivalent from an application point of view.
- delete?, get?, head?, post?, and put? return true or false based on the request method.
- xml_http_request? and xhr? return true if this request was issued by one of the Ajax helpers. Note that this parameter is independent of the method parameter.
- url(), which returns the full URL used for the request.
- protocol(), host(), port(), path(), and query_string(), which return components of the URL used for the request, based on the following pattern: protocol://host:port/path?query_string.
- domain(), which returns the last two components of the domain name of the request.
- host_with_port(), which is a host:port string for the request.
- port_string(), which is a :port string for the request if the port isn't the default port (80 for HTTP, 443 for HTTPS).
- ssl?(), which is true if this is an SSL request; in other words, the request was made with the HTTPS protocol.
- remote_ip(), which returns the remote IP address as a string. The string may have more than one address in it if the client is behind a proxy.

- env(), the environment of the request. You can use this to access values set by the browser, such as this:

```
request.env['HTTP_ACCEPT_LANGUAGE']
```

- accepts(), which is an array with Mime::Type objects that represent the MIME types in the Accept header.
- format(), which is computed based on the value of the Accept header, with Mime[:HTML] as a fallback.
- content_type(), which is the MIME type for the request. This is useful for put and post requests.
- headers(), which is the complete set of HTTP headers.
- body(), which is the request body as an I/O stream.
- content_length(), which is the number of bytes purported to be in the body.

Rails leverages a gem named Rack to provide much of this functionality. See the documentation of Rack::Request for full details.

response

The response object, filled in during the handling of the request. Normally, this object is managed for you by Rails. As we'll see when we look at callbacks in Callbacks, on page 366, we sometimes access the internals for specialized processing.

session

A hash-like object representing the current session data. We describe this in Rails Sessions, on page 360.

In addition, a logger is available throughout Action Pack.

Responding to the User

Part of the controller's job is to respond to the user, which is done in four ways:

- The most common way is to render a template. In terms of the MVC paradigm, the template is the view, taking information provided by the controller and using it to generate a response to the browser.
- The controller can return a string directly to the browser without invoking a view. This is fairly rare but can be used to send error notifications.
- The controller can return nothing to the browser. This is sometimes used when responding to an Ajax request. In all cases, however, the controller returns a set of HTTP headers because some kind of response is expected.

- The controller can send other data to the client (something other than HTML). This is typically a download of some kind (perhaps a PDF document or a file's contents).

A controller always responds to the user exactly one time per request. This means you should have just one call to a render(), redirect_to(), or send_*xxx*() method in the processing of any request. (A DoubleRenderError exception is thrown on the second render.)

Because the controller must respond exactly once, it checks to see whether a response has been generated just before it finishes handling a request. If not, the controller looks for a template named after the controller and action and automatically renders it. This is the most common way that rendering takes place. You may have noticed that in most of the actions in our shopping cart tutorial we never explicitly rendered anything. Instead, our action methods set up the context for the view and return. The controller notices that no rendering has taken place and automatically invokes the appropriate template.

You can have multiple templates with the same name but with different extensions (for example, .html.erb, .xml.builder, and .js.erb). If you don't specify an extension in a render request, Rails assumes html.erb.

Rendering Templates

A *template* is a file that defines the content of a response for our application. Rails supports three template formats out of the box: *erb*, which is embedded Ruby code (typically with HTML); *builder*, a more programmatic way of constructing XML content; and *RJS*, which generates JavaScript. We'll talk about the contents of these files starting in Using Templates, on page 369.

By convention, the template for *action* of *controller* will be in the file app/views/controller/action.type.xxx (where *type* is the file type, such as html, atom, or js; and *xxx* is one of erb, builder, or scss). The app/views part of the name is the default. You can override this for an entire application by setting this:

```
ActionController.prepend_view_path dir_path
```

The render() method is the heart of all rendering in Rails. It takes a hash of options that tell it what to render and how to render it.

It's tempting to write code in our controllers that looks like this:

```
# DO NOT DO THIS
def update
  @user = User.find(params[:id])
  if @user.update(user_params)
```

```
    render action: show
  end
  render template: "fix_user_errors"
end
```

It seems somehow natural that the act of calling render (and redirect_to) should somehow terminate the processing of an action. This isn't the case. The previous code will generate an error (because render is called twice) in the case where update succeeds.

Let's look at the render options used in the controller here (we'll look separately at rendering in the view starting in Partial-Page Templates, on page 390):

render()

With no overriding parameter, the render() method renders the default template for the current controller and action. The following code will render the template app/views/blog/index.html.erb:

```
class BlogController < ApplicationController
  def index
    render
  end
end
```

So will the following (as the default behavior of a controller is to call render() if the action doesn't):

```
class BlogController < ApplicationController
  def index
  end
end
```

And so will this (because the controller will call a template directly if no action method is defined):

```
class BlogController < ApplicationController
end
```

render(text: string)

Sends the given string to the client. No template interpretation or HTML escaping is performed.

```
class HappyController < ApplicationController
  def index
    render(text: "Hello there!")
  end
end
```

render(inline: string, [type: "erb"|"builder"|"scss"], [locals: hash])

Interprets *string* as the source to a template of the given type, rendering the results back to the client. You can use the :locals hash to set the values of local variables in the template.

The following code adds method_missing() to a controller if the application is running in development mode. If the controller is called with an invalid action, this renders an inline template to display the action's name and a formatted version of the request parameters:

```
class SomeController < ApplicationController

  if RAILS_ENV == "development"
    def method_missing(name, *args)
      render(inline: %{
        <h2>Unknown action: #{name}</h2>
        Here are the request parameters:<br/>
        <%= debug(params) %> })
    end
  end
end
```

render(action: action_name)

Renders the template for a given action in this controller. Sometimes folks use the :action form of render() when they should use redirects. See the discussion starting in Redirects, on page 356, for why this is a bad idea.

```
def display_cart
  if @cart.empty?
    render(action: :index)
  else
    # ...
  end
end
```

Note that calling render(:action...) does not call the action method; it simply displays the template. If the template needs instance variables, these must be set up by the method that calls the render() method.

Let's repeat this, because this is a mistake that beginners often make: calling render(:action...) does not invoke the action method. It simply renders that action's default template.

render(template: name, [locals: hash])

Renders a template and arranges for the resulting text to be sent back to the client. The :template value must contain both the controller and action parts of the new name, separated by a forward slash. The following code will render the template app/views/blog/short_list:

```
class BlogController < ApplicationController
  def index
    render(template: "blog/short_list")
  end
end
```

render(file: path)

Renders a view that may be entirely outside of your application (perhaps one shared with another Rails application). By default, the file is rendered without using the current layout. This can be overridden with layout: true.

render(partial: name, ...)

Renders a partial template. We talk about partial templates in depth in Partial-Page Templates, on page 390.

render(nothing: true)

Returns nothing—sends an empty body to the browser.

render(xml: stuff)

Renders *stuff* as text, forcing the content type to be application/xml.

render(json: stuff, [callback: hash])

Renders *stuff* as JSON, forcing the content type to be application/json. Specifying :callback will cause the result to be wrapped in a call to the named callback function.

render(:update) do |page| ... end

Renders the block as an RJS template, passing in the page object.

```
render(:update) do |page|
  page[:cart].replace_html partial: 'cart', object: @cart
  page[:cart].visual_effect :blind_down if @cart.total_items == 1
end
```

All forms of render() take optional :status, :layout, and :content_type parameters. The :status parameter provides the value used in the status header in the HTTP response. It defaults to "200 OK". Do not use render() with a 3xx status to do redirects; Rails has a redirect() method for this purpose.

The :layout parameter determines whether the result of the rendering will be wrapped by a layout. (We first came across layouts in Iteration C2: Adding a Page Layout, on page 105. We'll look at them in depth starting in Reducing Maintenance with Layouts and Partials, on page 385.) If the parameter is false, no layout will be applied. If set to nil or true, a layout will be applied only if there's one associated with the current action. If the :layout parameter has a string as a value, it'll be taken as the name of the layout to use when rendering. A layout is never applied when the :nothing option is in effect.

The :content_type parameter lets you specify a value that will be passed to the browser in the Content-Type HTTP header.

Sometimes it's useful to be able to capture what would otherwise be sent to the browser in a string. The render_to_string() method takes the same parameters as render() but returns the result of rendering as a string—the rendering is not stored in the response object and so won't be sent to the user unless you take some additional steps.

Calling render_to_string doesn't count as a real render. You can invoke the real render method later without getting a DoubleRender error.

Sending Files and Other Data

We've looked at rendering templates and sending strings in the controller. The third type of response is to send data (typically, but not necessarily, file contents) to the client.

```
send_data(data, options…)
```

This sends a data stream to the client. Typically the browser will use a combination of the content type and the disposition, both set in the options, to determine what to do with this data.

```
def sales_graph
png_data = Sales.plot_for(Date.today.month)
send_data(png_data, type: "image/png", disposition: "inline")
end
```

The options are as follows:

:disposition (string)
: Suggests to the browser that the file should be displayed inline (option inline) or downloaded and saved (option attachment, the default).

:filename string
: A suggestion to the browser of the default filename to use when saving this data.

:status (string)
: The status code (defaults to "200 OK").

:type (string)
: The content type, defaulting to application/octet-stream.

:url_based_filename boolean
: If true and :filename is not set, this option prevents Rails from providing the basename of the file in the Content-Disposition header. Specifying the

basename of the file is necessary to make some browsers handle i18n filenames correctly.

A related method is send_file, which sends the contents of a file to the client.

```
send_file(path, options…)
```

This sends the given file to the client. The method sets the Content-Length, Content-Type, Content-Disposition, and Content-Transfer-Encoding headers.

:buffer_size (number)
: The amount sent to the browser in each write if streaming is enabled (:stream is true).

:disposition (string)
: Suggests to the browser that the file should be displayed inline (option inline) or downloaded and saved (option attachment, the default).

:filename (string)
: A suggestion to the browser of the default filename to use when saving the file. If not set, defaults to the filename part of *path*.

:status string
: The status code (defaults to "200 OK").

:stream (true or false)
: If false, the entire file is read into server memory and sent to the client. Otherwise, the file is read and written to the client in :buffer_size chunks.

:type (string)
: The content type, defaulting to application/octet-stream.

You can set additional headers for either send_ method by using the headers attribute in the controller:

```
def send_secret_file
  send_file("/files/secret_list")
  headers["Content-Description"] = "Top secret"
end
```

We show how to upload files starting in Uploading Files to Rails Applications, on page 375.

Redirects

An HTTP redirect is sent from a server to a client in response to a request. In effect, it says, “I’m done processing this request, and you should go here to see the results.” The redirect response includes a URL that the client should try next along with some status information saying whether this redirection

is permanent (status code 301) or temporary (307). Redirects are sometimes used when web pages are reorganized; clients accessing pages in the old locations will get referred to the page's new home. More commonly, Rails applications use redirects to pass the processing of a request off to some other action.

Redirects are handled behind the scenes by web browsers. Normally, the only way you'll know that you've been redirected is a slight delay and the fact that the URL of the page you're viewing will have changed from the one you requested. This last point is important—as far as the browser is concerned, a redirect from a server acts pretty much the same as having an end user enter the new destination URL manually.

Redirects turn out to be important when writing well-behaved web applications. Let's look at a basic blogging application that supports comment posting. After a user has posted a comment, our application should redisplay the article, presumably with the new comment at the end.

It's tempting to code this using logic such as the following:

```
class BlogController
  def display
    @article = Article.find(params[:id])
  end

  def add_comment
    @article = Article.find(params[:id])
    comment  = Comment.new(params[:comment])
    @article.comments << comment
    if @article.save
      flash[:note] = "Thank you for your valuable comment"
    else
      flash[:note] = "We threw your worthless comment away"
    end
    # DON'T DO THIS
    render(action: 'display')
  end
end
```

The intent here was clearly to display the article after a comment has been posted. To do this, the developer ended the add_comment() method with a call to render(action:'display'). This renders the display view, showing the updated article to the end user. But think of this from the browser's point of view. It sends a URL ending in blog/add_comment and gets back an index listing. As far as the browser is concerned, the current URL is still the one that ends in blog/add_comment. This means that if the user hits Refresh or Reload (perhaps to see whether anyone else has posted a comment), the add_comment URL will be sent again to the

application. The user intended to refresh the display, but the application sees a request to add another comment. In a blog application, this kind of unintentional double entry is inconvenient. In an online store, it can get expensive.

In these circumstances, the correct way to show the added comment in the index listing is to redirect the browser to the display action. We do this using the Rails redirect_to() method. If the user subsequently hits Refresh, it will simply reinvoke the display action and not add another comment.

```
def add_comment
  @article = Article.find(params[:id])
  comment = Comment.new(params[:comment])
  @article.comments << comment
  if @article.save
    flash[:note] = "Thank you for your valuable comment"
  else
    flash[:note] = "We threw your worthless comment away"
  end
➤ redirect_to(action: 'display')
end
```

Rails has a lightweight yet powerful redirection mechanism. It can redirect to an action in a given controller (passing parameters), to a URL (on or off the current server), or to the previous page.

Let's look at these three forms in turn:

redirect_to(action: ..., options...) Sends a temporary redirection to the browser based on the values in the options hash. The target URL is generated using url_for(), so this form of redirect_to() has all the smarts of Rails routing code behind it.

redirect_to(path) Redirects to the given path. If the path doesn't start with a protocol (such as http://), the protocol and port of the current request will be prepended. This method does not perform any rewriting on the URL, so it shouldn't be used to create paths that are intended to link to actions in the application (unless you generate the path using url_for or a named route URL generator).

```
def save
  order = Order.new(params[:order])
  if order.save
    redirect_to action: "display"
  else
    session[:error_count] ||= 0
    session[:error_count] += 1
    if session[:error_count] < 4
      self.notice = "Please try again"
```

```
      else
        # Give up -- user is clearly struggling
        redirect_to("/help/order_entry.html")
      end
    end
  end
```

redirect_to(:back) Redirects to the URL given by the HTTP_REFERER header in the current request.

```
def save_details
  unless params[:are_you_sure] == 'Y'
    redirect_to(:back)
  else
    # ...
  end
end
```

By default all redirections are flagged as temporary (they'll affect only the current request). When redirecting to a URL, it's possible you might want to make the redirection permanent. In that case, set the status in the response header accordingly:

```
headers["Status"] = "301 Moved Permanently"
redirect_to("http://my.new.home")
```

Because redirect methods send responses to the browser, the same rules apply as for the rendering methods—you can issue only one per request.

So far, we've been looking at requests and responses in isolation. Rails also provides a number of mechanisms that span requests.

Objects and Operations That Span Requests

While the bulk of the state that persists across requests belongs in the database and is accessed via Active Record, some other bits of state have different life spans and need to be managed differently. In the Depot application, while the Cart itself was stored in the database, knowledge of which cart is the current cart was managed by sessions. Flash notices were used to communicate messages such as "Can't delete the last user" to the next request after a redirect. And callbacks were used to extract locale data from the URLs themselves.

In this section, we'll explore each of these mechanisms in turn.

Rails Sessions

A Rails session is a hash-like structure that persists across requests. Unlike raw cookies, sessions can hold any objects (as long as those objects can be marshaled), which makes them ideal for holding state information in web applications. For example, in our store application, we used a session to hold the shopping cart object between requests. The `Cart` object could be used in our application just like any other object. But Rails arranged things such that the cart was saved at the end of handling each request and, more important, that the correct cart for an incoming request was restored when Rails started to handle that request. Using sessions, we can pretend that our application stays around between requests.

And that leads to an interesting question: exactly where does this data stay around between requests? One choice is for the server to send it down to the client as a cookie. This is the default for Rails. It places limitations on the size and increases the bandwidth but means that there's less for the server to manage and clean up. Note that the contents are (by default) encrypted, which means that users can neither see nor tamper with the contents.

The other option is to store the data on the server. It requires more work to set up and is rarely necessary. First, Rails has to keep track of sessions. It does this by creating (by default) a 32-hex character key (which means there are 16^{32} possible combinations). This key is called the *session ID*, and it's effectively random. Rails arranges to store this session ID as a cookie (with the key `_session_id`) on the user's browser. Because subsequent requests come into the application from this browser, Rails can recover the session ID.

Second, Rails keeps a persistent store of session data on the server, indexed by the session ID. When a request comes in, Rails looks up the data store using the session ID. The data that it finds there is a serialized Ruby object. It deserializes this and stores the result in the controller's `session` attribute, where the data is available to our application code. The application can add to and modify this data to its heart's content. When it finishes processing each request, Rails writes the session data back into the data store. There it sits until the next request from this browser comes along.

What should you store in a session? You can store anything you want, subject to a few restrictions and caveats:

- Some restrictions apply on what kinds of object you can store in a session. The details depend on the storage mechanism you choose (which we'll look at shortly). In the general case, objects in a session

must be serializable (using Ruby's Marshal functions). This means, for example, that you can't store an I/O object in a session.

- If you store any Rails model objects in a session, you'll have to add model declarations for them. This causes Rails to preload the model class so that its definition is available when Ruby comes to deserialize it from the session store. If the use of the session is restricted to just one controller, this declaration can go at the top of that controller.

```
class BlogController < ApplicationController
  model :user_preferences
  # . . .
```

  However, if the session might get read by another controller (which is likely in any application with multiple controllers), you'll probably want to add the declaration to application_controller.rb in app/controllers.

- You probably don't want to store massive objects in session data—put them in the database and reference them from the session. This is particularly true for cookie-based sessions, where the overall limit is 4 KB.

- You probably don't want to store volatile objects in session data. For example, you might want to keep a tally of the number of articles in a blog and store that in the session for performance reasons. But if you do that, the count won't get updated if some other user adds an article.

  It's tempting to store objects representing the currently logged-in user in session data. This might not be wise if your application needs to be able to invalidate users. Even if a user is disabled in the database, their session data will still reflect a valid status.

  Store volatile data in the database, and reference it from the session instead.

- You probably don't want to store critical information solely in session data. For example, if your application generates an order confirmation number in one request and stores it in session data so that it can be saved to the database when the next request is handled, you risk losing that number if the user deletes the cookie from their browser. Critical information needs to be in the database.

One more caveat—and it's a big one. If you store an object in session data, then the next time you come back to that browser, your application will end up retrieving that object. However, if in the meantime you've updated your application, the object in session data may not agree with the definition of

that object's class in your application, and the application will fail while processing the request. You have three options here. One is to store the object in the database using conventional models and keep just the ID of the row in the session. Model objects are far more forgiving of schema changes than the Ruby marshaling library. The second option is to manually delete all the session data stored on your server whenever you change the definition of a class stored in that data.

The third option is slightly more complex. If you add a version number to your session keys and change that number whenever you update the stored data, you'll only ever load data that corresponds with the current version of the application. You can potentially version the classes whose objects are stored in the session and use the appropriate classes depending on the session keys associated with each request. This last idea can be a lot of work, so you'll need to decide whether it's worth the effort.

Because the session store is hash-like, you can save multiple objects in it, each with its own key.

There's no need to also disable sessions for particular actions. Because sessions are lazily loaded, simply don't reference a session in any action in which you don't need a session.

Session Storage

Rails has a number of options when it comes to storing your session data. Each has good and bad points. We'll start by listing the options and then compare them at the end.

The `session_store` attribute of `ActionController::Base` determines the session storage mechanism—set this attribute to a class that implements the storage strategy. This class must be defined in the `ActiveSupport::Cache::Store` module. You use symbols to name the session storage strategy; the symbol is converted into a CamelCase class name.

`session_store = :cookie_store`

This is the default session storage mechanism used by Rails, starting with version 2.0. This format represents objects in their marshaled form, which allows any serializable data to be stored in sessions but is limited to 4 KB total. This is the option we used in the Depot application.

session_store = :active_record_store

You can use the activerecord-session_store gem[2] to store your session data in your application's database using ActiveRecordStore.

session_store = :drb_store

DRb is a protocol that allows Ruby processes to share objects over a network connection. Using the DRbStore database manager, Rails stores session data on a DRb server (which you manage outside the web application). Multiple instances of your application, potentially running on distributed servers, can access the same DRb store. DRb uses Marshal to serialize objects.

session_store = :mem_cache_store

memcached is a freely available, distributed object caching system maintained by Dormando.[3] memcached is more complex to use than the other alternatives and is probably interesting only if you're already using it for other reasons at your site.

session_store = :memory_store

This option stores the session data locally in the application's memory. Because no serialization is involved, any object can be stored in an in-memory session. As we'll see in a minute, this generally isn't a good idea for Rails applications.

session_store = :file_store

Session data is stored in flat files. It's pretty much useless for Rails applications because the contents must be strings. This mechanism supports the additional configuration options :prefix, :suffix, and :tmpdir.

Comparing Session Storage Options

With all these session options to choose from, which should you use in your application? As always, the answer is "it depends."

When it comes to performance, there are few absolutes, and everyone's context is different. Your hardware, network latencies, database choices, and possibly even the weather will impact how all the components of session storage interact. Our best advice is to start with the simplest workable solution and then monitor it. If it starts to slow you down, find out why before jumping out of the frying pan.

2. https://github.com/rails/activerecord-session_store#installation
3. http://memcached.org/

If you have a high-volume site, keeping the size of the session data small and going with cookie_store is the way to go.

If we rule out memory store as being too simplistic, file store as too restrictive, and memcached as overkill, the server-side choices boil down to CookieStore, Active Record store, and DRb-based storage. Should you need to store more in a session than you can with cookies, we recommend you start with an Active Record solution. If, as your application grows, you find this becoming a bottleneck, you can migrate to a DRb-based solution.

Session Expiry and Cleanup

One problem with all the server-side session storage solutions is that each new session adds something to the session store. This means you'll eventually need to do some housekeeping or you'll run out of server resources.

Another reason to tidy up sessions is that many applications don't want a session to last forever. Once a user has logged in from a particular browser, the application might want to enforce a rule that the user stays logged in only as long as they're active; when they log out or some fixed time after they last use the application, their session should be terminated.

You can sometimes achieve this effect by expiring the cookie holding the session ID. But this is open to end-user abuse. Worse, it's hard to synchronize the expiry of a cookie on the browser with the tidying up of the session data on the server.

We therefore suggest you expire sessions by simply removing their server-side session data. Should a browser request subsequently arrive containing a session ID for data that's been deleted, the application will receive no session data; the session will effectively not be there.

Implementing this expiration depends on the storage mechanism being used.

For Active Record–based session storage, use the updated_at columns in the sessions table. You can delete all sessions that have not been modified in the last hour (ignoring daylight saving time changes) by having your sweeper task issue SQL such as this:

```
delete from sessions
 where now() - updated_at > 3600;
```

For DRb-based solutions, expiry takes place within the DRb server process. You'll probably want to record timestamps alongside the entries in the session data hash. You can run a separate thread (or even a separate process) that periodically deletes the entries in this hash.

In all cases, your application can help this process by calling reset_session() to delete sessions when they're no longer needed (for example, when a user logs out).

Flash: Communicating Between Actions

When we use redirect_to() to transfer control to another action, the browser generates a separate request to invoke that action. That request will be handled by our application in a fresh instance of a controller object—instance variables that were set in the original action aren't available to the code handling the redirected action. But sometimes we need to communicate between these two instances. We can do this using a facility called the *flash*.

The flash is a temporary scratchpad for values. It's organized like a hash and stored in the session data, so you can store values associated with keys and later retrieve them. It has one special property. By default, values stored into the flash during the processing of a request will be available during the processing of the immediately following request. Once that second request has been processed, those values are removed from the flash.

Probably the most common use of the flash is to pass error and informational strings from one action to the next. The intent here is that the first action notices some condition, creates a message describing that condition, and redirects to a separate action. By storing the message in the flash, the second action is able to access the message text and use it in a view. An example of such usage can be found in Iteration E1 on page 134.

It's sometimes convenient to use the flash as a way of passing messages into a template in the current action. For example, our display() method might want to output a cheery banner if there isn't another, more pressing note. It doesn't need that message to be passed to the next action—it's for use in the current request only. To do this, it could use flash.now, which updates the flash but doesn't add to the session data.

While flash.now creates a transient flash entry, flash.keep does the opposite, making entries that are currently in the flash stick around for another request cycle. If you pass no parameters to flash.keep, then all the flash contents are preserved.

Flashes can store more than just text messages—you can use them to pass all kinds of information between actions. Obviously, for longer-term information you'd want to use the session (probably in conjunction with your database) to store the data, but the flash is great if you want to pass parameters from one request to the next.

Because the flash data is stored in the session, all the usual rules apply. In particular, every object must be serializable. We strongly recommend passing only basic objects like Strings or Hashes in the flash.

Callbacks

Callbacks enable you to write code in your controllers that wrap the processing performed by actions—you can write a chunk of code once and have it be called before or after any number of actions in your controller (or your controller's subclasses). This turns out to be a powerful facility. Using callbacks, we can implement authentication schemes, logging, response compression, and even response customization.

Rails supports three types of callbacks: before, after, and around. Such callbacks are called just prior to and/or just after the execution of actions. Depending on how you define them, they either run as methods inside the controller or are passed to the controller object when they are run. Either way, they get access to details of the request and response objects, along with the other controller attributes.

Before and After Callbacks

As their names suggest, before and after callbacks are invoked before or after an action. Rails maintains two chains of callbacks for each controller. When a controller is about to run an action, it executes all the callbacks on the before chain. It executes the action before running the callbacks on the after chain.

Callbacks can be passive, monitoring activity performed by a controller. They can also take a more active part in request handling. If a before action callback returns `false`, then processing of the callback chain terminates and the action is not run. A callback may also render output or redirect requests, in which case the original action never gets invoked.

We saw an example of using callbacks for authorization in the administration part of our store example in Iteration I3: Limiting Access, on page 217. We defined an authorization method that redirected to a login screen if the current session didn't have a logged-in user. We then made this method a before action callback for all the actions in the administration controller.

Callback declarations also accept blocks and the names of classes. If a block is specified, it'll be called with the current controller as a parameter. If a class is given, its `filter()` class method will be called with the controller as a parameter.

By default, callbacks apply to all actions in a controller (and any subclasses of that controller). You can modify this with the `:only` option, which takes one

or more actions on which the callback is invoked, and the :except option, which lists actions to be excluded from callback.

The before_action and after_action declarations append to the controller's chain of callbacks. Use the variants prepend_before_action() and prepend_after_action() to put callbacks at the front of the chain.

After callbacks can be used to modify the outbound response, changing the headers and content if required. Some applications use this technique to perform global replacements in the content generated by the controller's templates (for example, by substituting a customer's name for the string <customer/> in the response body). Another use might be compressing the response if the user's browser supports it.

Around callbacks wrap the execution of actions. You can write an around callback in two different styles. In the first, the callback is a single chunk of code. That code is called before the action is executed. If the callback code invokes yield, the action is executed. When the action completes, the callback code continues executing.

Thus, the code before the yield is like a before action callback, and the code after is the after action callback. If the callback code never invokes yield, the action isn't run—this way you can achieve the same result as a before action callback returning false.

The benefit of around callbacks is that they can retain context across the invocation of the action.

As well as passing around_action the name of a method, you can pass it a block or a filter class.

If you use a block as a callback, it'll be passed two parameters: the controller object and a proxy for the action. Use call() on this second parameter to invoke the original action.

A second form allows you to pass an object as a callback. This object should implement a method called filter(). This method will be passed the controller object. It yields to invoke the action.

Like before and after callbacks, around callbacks take :only and :except parameters.

Around callbacks are (by default) added to the callback chain differently: the first around action callback added executes first. Subsequently added around callbacks will be nested within existing around callbacks.

Callback Inheritance

If you subclass a controller containing callbacks, the callbacks will be run on the child objects as well as in the parent. But callbacks defined in the children won't run in the parent.

If you don't want a particular callback to run in a child controller, you can override the default processing with the skip_before_action and skip_after_action declarations. These accept the :only and :except parameters.

You can use skip_action to skip any action callback (before, after, and around). However, it works only for callbacks that were specified as the (symbol) name of a method.

We made use of skip_before_action in Iteration I3: Limiting Access, on page 217.

What We Just Did

We learned how Action Dispatch and Action Controller cooperate to enable our server to respond to requests. The importance of this can't be emphasized enough. In nearly every application, this is the primary place where the creativity of your application is expressed. While Active Record and Action View are hardly passive, our routes and our controllers are where the action is.

We started this chapter by covering the concept of REST, which was the inspiration for the way in which Rails approaches the routing of requests. We saw how this provided seven basic actions as a starting point and how to add more actions. We also saw how to select a data representation (for example, JSON or XML). And we covered how to test routes.

We then covered the environment that Action Controller provides for your actions as well as the methods it provides for rendering and redirecting. Finally, we covered sessions, flash, and callbacks, each of which is available for use in your application's controllers.

Along the way, we showed how these concepts were used in the Depot application. Now that you've seen each in use and have been exposed to the theory behind each, how you combine and use these concepts is limited only by your own creativity.

In the next chapter, we'll cover the remaining component of Action Pack, namely, Action View, which handles the rendering of results.