Action Controller Overview

In this guide you will learn how controllers work and how they fit into the request cycle in your application. After reading this guide, you will be able to:

Follow the flow of a request through a controller

Understand why and how to store data in the session or cookies

Work with filters to execute code during request processing

Use Action Controller's built-in HTTP authentication

Stream data directly to the user's browser

Filter sensitive parameters so they do not appear in the application's log

Deal with exceptions that may be raised during request processing

Chapters

1 What Does a Controller Do?

Action Controller is the C in MVC. After routing has determined which controller to use for a request, your controller is responsible for making sense of the request and producing the appropriate output. Luckily, Action Controller does most of the groundwork for you and uses smart conventions to make this as straightforward as possible.

For most conventional **RESTful** applications, the controller will receive the request (this is invisible to you as the developer), fetch or save data from a model and use a view to create HTML output. If your controller needs to do things a little differently, that's not a problem, this is just the most common way for a controller to work.

A controller can thus be thought of as a middle man between models and views. It makes the model data available to the view so it can display that data to the user, and it saves or updates data from the user to the model.

For more details on the routing process, see **Rails Routing from the Outside In**.

2 Methods and Actions

A controller is a Ruby class which inherits from `ApplicationController` and has methods just like any other class. When your application receives a request, the routing will determine which controller and action to run, then Rails creates an instance of that controller and runs the method with the same name as the action.

```
class

ClientsController < ApplicationController

def

new

end
end
```

As an example, if a user goes to `/clients/new` in your application to add a new client, Rails will create an instance of `ClientsController` and run the `new` method. Note that the empty method from the example above could work just fine because Rails will by default render the `new.html.erb` view unless the action says otherwise. The `new` method could make available to the view a `@client` instance variable by creating a new `Client`:

```
def

new

@client

= Client.
new
end
```

The **Layouts & Rendering Guide** explains this in more detail.

`ApplicationController` inherits from `ActionController::Base`, which defines a number of helpful methods. This guide will cover some of these, but if you're curious to see what's in there, you can see all of them in the API documentation or in the source itself.

Only public methods are callable as actions. It is a best practice to lower the visibility of methods which are not intended to be actions, like auxiliary methods or filters.

3 Parameters

You will probably want to access data sent in by the user or other parameters in your controller actions. There are two kinds of parameters possible in a web application. The first are parameters that are sent as part of the URL, called query string parameters. The query string is everything after "?" in the URL. The second type of parameter is usually referred to as POST data. This information usually comes from an HTML form which has been filled in by the user. It's called POST data because it can only be sent as part of an HTTP POST request. Rails does not make any distinction between query string

parameters and POST parameters, and both are available in the `params` hash in your controller:

```
class

ClientsController < ActionController::Base

# This action uses query string parameters because it gets run

# by an HTTP GET request, but this does not make any difference

# to the way in which the parameters are accessed. The URL for

# this action would look like this in order to list activated

# clients: /clients?status=activated

def

index

if

params[
:status
] ==
"activated"

@clients

= Client.activated

else

@clients

= Client.unactivated

end

end


# This action uses POST parameters. They are most likely coming

# from an HTML form which the user has submitted. The URL for

# this RESTful request will be "/clients", and the data will be

# sent as part of the request body.

def

create

@client

= Client.
new
(params[
:client
])

if

@client
.save

redirect_to
@client

else

# This line overrides the default rendering behavior, which

# would have been to render the "create" view.
```

```
render
:action

=>
"new"

end

end
end
```

3.1 Hash and Array Parameters

The `params` hash is not limited to one-dimensional keys and values. It can contain arrays and (nested) hashes. To send an array of values, append an empty pair of square brackets "[]" to the key name:

```
GET /clients?ids[]=1&ids[]=2&ids[]=3
```

The actual URL in this example will be encoded as "/clients?ids%5b%5d=1&ids%5b%5d=2&ids%5b%5d=3" as "[" and "]" are not allowed in URLs. Most of the time you don't have to worry about this because the browser will take care of it for you, and Rails will decode it back when it receives it, but if you ever find yourself having to send those requests to the server manually you have to keep this in mind.

The value of `params[:ids]` will now be `["1", "2", "3"]`. Note that parameter values are always strings; Rails makes no attempt to guess or cast the type.

To send a hash you include the key name inside the brackets:

```
<
form

accept-charset
=
"UTF-8"

action
=
"/clients"

method
=
"post"
>

<
input

type
=
"text"

name
=
"client[name]"

value
=
"Acme"

/>

<
input

type
=
"text"

name
=
"client[phone]"
```

```
value
=
"12345"

/>

<
input

type
=
"text"

name
=
"client[address][postcode]"

value
=
"12345"

/>

<
input

type
=
"text"

name
=
"client[address][city]"

value
=
"Carrot City"

/>
</
form
>
```

When this form is submitted, the value of `params[:client]` will be `{"name" => “Acme”, “phone” => “12345”, “address” => {"postcode" => “12345”, “city” => “Carrot City”}}`. Note the nested hash in `params[:client][:address]`.

Note that the `params` hash is actually an instance of `HashWithIndifferentAccess` from Active Support, which acts like a hash that lets you use symbols and strings interchangeably as keys.

3.2 JSON/XML parameters

If you’re writing a web service application, you might find yourself more comfortable on accepting parameters in JSON or XML format. Rails will automatically convert your parameters into `params` hash, which you’ll be able to access like you would normally do with form data.

So for example, if you are sending this JSON parameter:

```
{ "company": { "name": "acme", "address": "123 Carrot Street" } }
```

You’ll get `params[:company]` as `{ :name => “acme”, “address” => “123 Carrot Street” }`.

Also, if you’ve turned on `config.wrap_parameters` in your initializer or calling `wrap_parameters` in your controller, you can safely omit the root element in the JSON/XML parameter. The parameters will be cloned and wrapped in the key according to your controller’s name by default. So the above parameter can be written as:

```
{ "name": "acme", "address": "123 Carrot Street" }
```

And assume that you’re sending the data to `CompaniesController`, it would then be wrapped in `:company` key like this:

```
{
:name

=>
"acme"
,
:address

=>
"123 Carrot Street"
,
:company

=> {
:name

=>
"acme"
,
:address

=>
"123 Carrot Street"

}}
```

You can customize the name of the key or specific parameters you want to wrap by consulting the **API documentation**

3.3 Routing Parameters

The `params` hash will always contain the `:controller` and `:action` keys, but you should use the methods `controller_name` and `action_name` instead to access these values. Any other parameters defined by the routing, such as `:id` will also be available. As an example, consider a listing of clients where the list can show either active or inactive clients. We can add a route which captures the `:status` parameter in a "pretty" URL:

```
match
'/clients/:status'

=>
'clients#index'
,
:foo

=>
"bar"
```

In this case, when a user opens the URL `/clients/active`, `params[:status]` will be set to "active". When this route is used, `params[:foo]` will also be set to "bar" just like it was passed in the query string. In the same way `params[:action]` will contain "index".

3.4 `default_url_options`

You can set global default parameters for URL generation by defining a method called `default_url_options` in your controller. Such a method must return a hash with the desired defaults, whose keys must be symbols:

```
class

ApplicationController < ActionController::Base

def

default_url_options

{
:locale

=> I18n.locale}

end
end
```

These options will be used as a starting point when generating URLs, so it's possible they'll be overridden by the options passed in `url_for` calls.

If you define `default_url_options` in `ApplicationController`, as in the example above, it would be used for all URL generation. The method can also be defined in one specific controller, in which case it only affects URLs generated there.

4 Session

Your application has a session for each user in which you can store small amounts of data that will be persisted between requests. The session is only available in the controller and the view and can use one of a number of different storage mechanisms:

ActionDispatch::Session::CookieStore – Stores everything on the client.
ActiveRecord::SessionStore – Stores the data in a database using Active Record.
ActionDispatch::Session::CacheStore – Stores the data in the Rails cache.
ActionDispatch::Session::MemCacheStore – Stores the data in a memcached cluster (this is a legacy implementation; consider using CacheStore instead).

All session stores use a cookie to store a unique ID for each session (you must use a cookie, Rails will not allow you to pass the session ID in the URL as this is less secure).

For most stores this ID is used to look up the session data on the server, e.g. in a database table. There is one exception, and that is the default and recommended session store – the CookieStore – which stores all session data in the cookie itself (the ID is still available to you if you need it). This has the advantage of being very lightweight and it requires zero setup in a new application in order to use the session. The cookie data is cryptographically signed to make it tamper-proof, but it is not encrypted, so anyone with access to it can read its contents but not edit it (Rails will not accept it if it has been edited).

The CookieStore can store around 4kB of data — much less than the others — but this is usually enough. Storing large amounts of data in the session is discouraged no matter which session store your application uses. You should especially avoid storing complex objects (anything other than basic Ruby objects, the most common example being model instances) in the session, as the server might not be able to reassemble them between requests, which will result in an error.

If your user sessions don't store critical data or don't need to be around for long periods (for instance if you just use the flash for messaging), you can consider using ActionDispatch::Session::CacheStore. This will store sessions using the cache implementation you have configured for your application. The advantage of this is that you can use your existing cache infrastructure for storing sessions without requiring any additional setup or administration. The downside, of course, is that the sessions will be ephemeral and could disappear at any time.

Read more about session storage in the **Security Guide**.

If you need a different session storage mechanism, you can change it in the `config/initializers/session_store.rb` file:

```
# Use the database for sessions instead of the cookie-based default,
# which shouldn't be used to store highly confidential information
# (create the session table with "script/rails g session_migration")
# YourApp::Application.config.session_store :active_record_store
```

Rails sets up a session key (the name of the cookie) when signing the session data. These can also be changed in `config/initializers/session_store.rb`:

```
# Be sure to restart your server when you modify this file.

YourApp::Application.config.session_store
:cookie_store
,
:key

=>
'_your_app_session'
```

You can also pass a `:domain` key and specify the domain name for the cookie:

```
# Be sure to restart your server when you modify this file.
```

```
YourApp::Application.config.session_store
:cookie_store
,
:key

=>
'_your_app_session'
,
:domain

=>
".example.com"
```

Rails sets up (for the CookieStore) a secret key used for signing the session data. This can be changed in `config/initializers/secret_token.rb`

```
# Be sure to restart your server when you modify this file.

# Your secret key for verifying the integrity of signed cookies.
# If you change this key, all old signed cookies will become invalid!
# Make sure the secret is at least 30 characters and all random,
# no regular words or you'll be exposed to dictionary attacks.
YourApp::Application.config.secret_token =
'49d3f3de9ed86c74b94ad6bd0...'
```

Changing the secret when using the `CookieStore` will invalidate all existing sessions.

4.1 Accessing the Session

In your controller you can access the session through the `session` instance method.

Sessions are lazily loaded. If you don't access sessions in your action's code, they will not be loaded. Hence you will never need to disable sessions, just not accessing them will do the job.

Session values are stored using key/value pairs like a hash:

```
class

ApplicationController < ActionController::Base


private


# Finds the User with the ID stored in the session with the key

# :current_user_id This is a common way to handle user login in

# a Rails application; logging in sets the session value and

# logging out removes it.

def

current_user

@_current_user

||= session[
:current_user_id
] &&

User.find_by_id(session[
:current_user_id
])

end
end
```

To store something in the session, just assign it to the key like a hash:

```
class

LoginsController < ApplicationController

# "Create" a login, aka "log the user in"

def

create

if

user = User.authenticate(params[
:username
], params[
:password
])

# Save the user ID in the session so it can be used in

# subsequent requests

session[
:current_user_id
] = user.id

redirect_to root_url

end

end
end
```

To remove something from the session, assign that key to be `nil`:

```
class

LoginsController < ApplicationController

# "Delete" a login, aka "log the user out"

def

destroy

# Remove the user id from the session

@_current_user

= session[
:current_user_id
] =
nil

redirect_to root_url

end
end
```

To reset the entire session, use `reset_session`.

4.2 The Flash

The flash is a special part of the session which is cleared with each request. This means that values stored there will only be available in the next request, which is useful for storing error messages etc. It is accessed in much the same way as the session, like a hash. Let's use the act of logging out as an example. The controller can send a message which will be displayed to the user on the next request:

```
class

LoginsController < ApplicationController

def

destroy

session[
:current_user_id
] =
nil

flash[
:notice
] =
"You have successfully logged out"

redirect_to root_url

end
end
```

Note it is also possible to assign a flash message as part of the redirection.

```
redirect_to root_url,
:notice

=>
"You have successfully logged out"
```

The `destroy` action redirects to the application's `root_url`, where the message will be displayed. Note that it's entirely up to the next action to decide what, if anything, it will do with what the previous action put in the flash. It's conventional to display eventual errors or notices from the flash in the application's layout:

```
<html>

<!-- <head/> -->

<body>

<%
if

flash[
:notice
] %>

<p
class
=
"notice"
><%= flash[
:notice
] %></p>

<%
end

%>

<%
if

flash[
:error
] %>

<p
class
```

```
=
"error"
><%= flash[
:error
] %></p>

<%
end

%>

<!-- more content -->

</body>
</html>
```

This way, if an action sets an error or a notice message, the layout will display it automatically.

If you want a flash value to be carried over to another request, use the `keep` method:

```
class

MainController < ApplicationController

# Let's say this action corresponds to root_url, but you want

# all requests here to be redirected to UsersController#index.

# If an action sets the flash and redirects here, the values

# would normally be lost when another redirect happens, but you

# can use 'keep' to make it persist for another request.

def

index

# Will persist all flash values.

flash.keep


# You can also use a key to keep only some kind of value.

# flash.keep(:notice)

redirect_to users_url

end
end
```

4.2.1 `flash.now`

By default, adding values to the flash will make them available to the next request, but sometimes you may want to access those values in the same request. For example, if the `create` action fails to save a resource and you render the `new` template directly, that's not going to result in a new request, but you may still want to display a message using the flash. To do this, you can use `flash.now` in the same way you use the normal `flash`:

```
class

ClientsController < ApplicationController

def

create

@client

= Client.
new
```

```
(params[
:client
])

if

@client
.save

# ...

else

flash.now[
:error
] =
"Could not save client"

render
:action

=>
"new"

end

end
end
```

5 Cookies

Your application can store small amounts of data on the client — called cookies — that will be persisted across requests and even sessions. Rails provides easy access to cookies via the `cookies` method, which — much like the `session` — works like a hash:

```
class

CommentsController < ApplicationController

def

new

# Auto-fill the commenter's name if it has been stored in a cookie

@comment

= Comment.
new
(
:name

=> cookies[
:commenter_name
])

end


def

create

@comment

= Comment.
new
(params[
:comment
])

if

@comment
```

```
.save

flash[
:notice
] =
"Thanks for your comment!"

if

params[
:remember_name
]

# Remember the commenter's name.

cookies[
:commenter_name
] =
@comment
.name

else

# Delete cookie for the commenter's name cookie, if any.

cookies.delete(
:commenter_name
)

end

redirect_to
@comment
.article

else

render
:action

=>
"new"

end

end
end
```

Note that while for session values you set the key to `nil`, to delete a cookie value you should use `cookies.delete(:key)`.

6 Rendering xml and json data

ActionController makes it extremely easy to render `xml` or `json` data. If you generate a controller using scaffold then your controller would look something like this.

```
class

UsersController < ApplicationController

def

index

@users

= User.all

respond_to
do

|format|

format.html
```

```
# index.html.erb

format.xml  { render
:xml

=>
@users
}

format.json { render
:json

=>
@users
}

end

end
end
```

Notice that in the above case code is `render :xml => @users` and not `render :xml => @users.to_xml`. That is because if the input is not string then rails automatically invokes `to_xml` .

7 Filters

Filters are methods that are run before, after or “around” a controller action.

Filters are inherited, so if you set a filter on `ApplicationController`, it will be run on every controller in your application.

Before filters may halt the request cycle. A common before filter is one which requires that a user is logged in for an action to be run. You can define the filter method this way:

```
class

ApplicationController < ActionController::Base

before_filter
:require_login


private


def

require_login

unless

logged_in?

flash[
:error
] =
"You must be logged in to access this section"

redirect_to new_login_url
# halts request cycle

end

end


# The logged_in? method simply returns true if the user is logged

# in and false otherwise. It does this by "booleanizing" the

# current_user method we created previously using a double ! operator.

# Note that this is not common in Ruby and is discouraged unless you
```

```
# really mean to convert something into true or false.

def

logged_in?

!!current_user

end
end
```

The method simply stores an error message in the flash and redirects to the login form if the user is not logged in. If a before filter renders or redirects, the action will not run. If there are additional filters scheduled to run after that filter they are also cancelled.

In this example the filter is added to `ApplicationController` and thus all controllers in the application inherit it. This will make everything in the application require the user to be logged in in order to use it. For obvious reasons (the user wouldn't be able to log in in the first place!), not all controllers or actions should require this. You can prevent this filter from running before particular actions with `skip_before_filter`:

```
class

LoginsController < ApplicationController

skip_before_filter
:require_login
,
:only

=> [
:new
,
:create
]
end
```

Now, the `LoginsController`'s `new` and `create` actions will work as before without requiring the user to be logged in. The `:only` option is used to only skip this filter for these actions, and there is also an `:except` option which works the other way. These options can be used when adding filters too, so you can add a filter which only runs for selected actions in the first place.

7.1 After Filters and Around Filters

In addition to before filters, you can also run filters after an action has been executed, or both before and after.

After filters are similar to before filters, but because the action has already been run they have access to the response data that's about to be sent to the client. Obviously, after filters cannot stop the action from running.

Around filters are responsible for running their associated actions by yielding, similar to how Rack middlewares work.

For example, in a website where changes have an approval workflow an administrator could be able to preview them easily, just apply them within a transaction:

```
class

ChangesController < ActionController::Base

around_filter
:wrap_in_transaction
,
:only

=>
:show


private
```

```
def

wrap_in_transaction

ActiveRecord::Base.transaction
do

begin

yield

ensure

raise

ActiveRecord::Rollback

end

end

end
end
```

Note that an around filter wraps also rendering. In particular, if in the example above the view itself reads from the database via a scope or whatever, it will do so within the transaction and thus present the data to preview.

They can choose not to yield and build the response themselves, in which case the action is not run.

7.2 Other Ways to Use Filters

While the most common way to use filters is by creating private methods and using *_filter to add them, there are two other ways to do the same thing.

The first is to use a block directly with the *_filter methods. The block receives the controller as an argument, and the `require_login` filter from above could be rewritten to use a block:

```
class

ApplicationController < ActionController::Base

before_filter
do

|controller|

redirect_to new_login_url
unless

controller.send(
:logged_in
?)

end
end
```

Note that the filter in this case uses `send` because the `logged_in?` method is private and the filter is not run in the scope of the controller. This is not the recommended way to implement this particular filter, but in more simple cases it might be useful.

The second way is to use a class (actually, any object that responds to the right methods will do) to handle the filtering. This is useful in cases that are more complex and can not be implemented in a readable and reusable way using the two other methods. As an example, you could rewrite the login filter again to use a class:

```
class

ApplicationController < ActionController::Base

before_filter LoginFilter
end
```

```
class

LoginFilter

def

self
.filter(controller)

unless

controller.send(
:logged_in
?)

controller.flash[
:error
] =
"You must be logged in"

controller.redirect_to controller.new_login_url

end

end
end
```

Again, this is not an ideal example for this filter, because it's not run in the scope of the controller but gets the controller passed as an argument. The filter class has a class method `filter` which gets run before or after the action, depending on if it's a before or after filter. Classes used as around filters can also use the same `filter` method, which will get run in the same way. The method must `yield` to execute the action. Alternatively, it can have both a `before` and an `after` method that are run before and after the action.

8 Request Forgery Protection

Cross-site request forgery is a type of attack in which a site tricks a user into making requests on another site, possibly adding, modifying or deleting data on that site without the user's knowledge or permission.

The first step to avoid this is to make sure all "destructive" actions (create, update and destroy) can only be accessed with non-GET requests. If you're following RESTful conventions you're already doing this. However, a malicious site can still send a non-GET request to your site quite easily, and that's where the request forgery protection comes in. As the name says, it protects from forged requests.

The way this is done is to add a non-guessable token which is only known to your server to each request. This way, if a request comes in without the proper token, it will be denied access.

If you generate a form like this:

```
<%= form_for
@user

do

|f| %>

<%= f.text_field
:username

%>

<%= f.text_field
:password

%>
<%
end

%>
```

You will see how the token gets added as a hidden field:

```
<
form

accept-charset
=
"UTF-8"

action
=
"/users/1"

method
=
"post"
>
<
input

type
=
"hidden"

value
=
"67250ab105eb5ad10851c00a5621854a23af5489"

name
=
"authenticity_token"
/>
<!-- fields -->
</
form
>
```

Rails adds this token to every form that's generated using the **form helpers**, so most of the time you don't have to worry about it. If you're writing a form manually or need to add the token for another reason, it's available through the method `form_authenticity_token`:

The `form_authenticity_token` generates a valid authentication token. That's useful in places where Rails does not add it automatically, like in custom Ajax calls.

The **Security Guide** has more about this and a lot of other security-related issues that you should be aware of when developing a web application.

9 The Request and Response Objects

In every controller there are two accessor methods pointing to the request and the response objects associated with the request cycle that is currently in execution. The `request` method contains an instance of `AbstractRequest` and the `response` method returns a response object representing what is going to be sent back to the client.

9.1 The `request` Object

The request object contains a lot of useful information about the request coming in from the client. To get a full list of the available methods, refer to the **API documentation**. Among the properties that you can access on this object are:

Property of `request`	Purpose
host	The hostname used for this request.
domain(n=2)	The hostname's first n segments, starting from the right (the TLD).
format	The content type requested by the client.
method	The HTTP method used for the request.
get?, post?, put?, delete?, head?	Returns true if the HTTP method is GET/POST/PUT/DELETE/HEAD.
headers	Returns a hash containing the headers associated with the request.
port	The port number (integer) used for the request.
protocol	Returns a string containing the protocol used plus "://", for example "http://".

query_string	The query string part of the URL, i.e., everything after “?”.
remote_ip	The IP address of the client.
url	The entire URL used for the request.

9.1.1 `path_parameters`, `query_parameters`, and `request_parameters`

Rails collects all of the parameters sent along with the request in the `params` hash, whether they are sent as part of the query string or the post body. The request object has three accessors that give you access to these parameters depending on where they came from. The `query_parameters` hash contains parameters that were sent as part of the query string while the `request_parameters` hash contains parameters sent as part of the post body. The `path_parameters` hash contains parameters that were recognized by the routing as being part of the path leading to this particular controller and action.

9.2 The `response` Object

The response object is not usually used directly, but is built up during the execution of the action and rendering of the data that is being sent back to the user, but sometimes – like in an after filter – it can be useful to access the response directly. Some of these accessor methods also have setters, allowing you to change their values.

Property of `response`	Purpose
body	This is the string of data being sent back to the client. This is most often HTML.
status	The HTTP status code for the response, like 200 for a successful request or 404 for file not found.
location	The URL the client is being redirected to, if any.
content_type	The content type of the response.
charset	The character set being used for the response. Default is “utf-8”.
headers	Headers used for the response.

9.2.1 Setting Custom Headers

If you want to set custom headers for a response then `response.headers` is the place to do it. The headers attribute is a hash which maps header names to their values, and Rails will set some of them automatically. If you want to add or change a header, just assign it to `response.headers` this way:

```
response.headers[
"Content-Type"
] =
"application/pdf"
```

10 HTTP Authentications

Rails comes with two built-in HTTP authentication mechanisms:

- Basic Authentication
- Digest Authentication

10.1 HTTP Basic Authentication

HTTP basic authentication is an authentication scheme that is supported by the majority of browsers and other HTTP clients. As an example, consider an administration section which will only be available by entering a username and a password into the browser’s HTTP basic dialog window. Using the built-in authentication is quite easy and only requires you to use one method, `http_basic_authenticate_with`.

```
class

AdminController < ApplicationController

http_basic_authenticate_with
:name

=>
"humbaba"
,
:password
```

```
=>
"5baa61e4"
end
```

With this in place, you can create namespaced controllers that inherit from `AdminController`. The filter will thus be run for all actions in those controllers, protecting them with HTTP basic authentication.

10.2 HTTP Digest Authentication

HTTP digest authentication is superior to the basic authentication as it does not require the client to send an unencrypted password over the network (though HTTP basic authentication is safe over HTTPS). Using digest authentication with Rails is quite easy and only requires using one method, `authenticate_or_request_with_http_digest`.

```
class

AdminController < ApplicationController

USERS

= {
"lifo"

=>
"world"

}


before_filter
:authenticate


private


def

authenticate

authenticate_or_request_with_http_digest
do

|username|

USERS
[username]

end

end
end
```

As seen in the example above, the `authenticate_or_request_with_http_digest` block takes only one argument – the username. And the block returns the password. Returning `false` or `nil` from the `authenticate_or_request_with_http_digest` will cause authentication failure.

11 Streaming and File Downloads

Sometimes you may want to send a file to the user instead of rendering an HTML page. All controllers in Rails have the `send_data` and the `send_file` methods, which will both stream data to the client. `send_file` is a convenience method that lets you provide the name of a file on the disk and it will stream the contents of that file for you.

To stream data to the client, use `send_data`:

```
require
"prawn"
class
```

```
ClientsController < ApplicationController

# Generates a PDF document with information on the client and

# returns it. The user will get the PDF as a file download.

def

download_pdf

client = Client.find(params[
:id
])

send_data generate_pdf(client),

:filename

=>
"#{client.name}.pdf"
,

:type

=>
"application/pdf"

end


private


def

generate_pdf(client)

Prawn::Document.
new

do

text client.name,
:align

=>
:center

text
"Address: #{client.address}"

text
"Email: #{client.email}"

end
.render

end
end
```

The `download_pdf` action in the example above will call a private method which actually generates the PDF document and returns it as a string. This string will then be streamed to the client as a file download and a filename will be suggested to the user. Sometimes when streaming files to the user, you may not want them to download the file. Take images, for example, which can be embedded into HTML pages. To tell the browser a file is not meant to be downloaded, you can set the `:disposition` option to “inline”. The opposite and default value for this option is “attachment”.

11.1 Sending Files

If you want to send a file that already exists on disk, use the `send_file` method.

```
class

ClientsController < ApplicationController
```

```
# Stream a file that has already been generated and stored on disk.

def

download_pdf

client = Client.find(params[
:id
])

send_file(
"#{Rails.root}/files/clients/#{client.id}.pdf"
,

:filename

=>
"#{client.name}.pdf"
,

:type

=>
"application/pdf"
)

end
end
```

This will read and stream the file 4kB at the time, avoiding loading the entire file into memory at once. You can turn off streaming with the `:stream` option or adjust the block size with the `:buffer_size` option.

If `:type` is not specified, it will be guessed from the file extension specified in `:filename`. If the content type is not registered for the extension, `application/octet-stream` will be used.

> Be careful when using data coming from the client (params, cookies, etc.) to locate the file on disk, as this is a security risk that might allow someone to gain access to files they are not meant to see.

It is not recommended that you stream static files through Rails if you can instead keep them in a public folder on your web server. It is much more efficient to let the user download the file directly using Apache or another web server, keeping the request from unnecessarily going through the whole Rails stack.

11.2 RESTful Downloads

While `send_data` works just fine, if you are creating a RESTful application having separate actions for file downloads is usually not necessary. In REST terminology, the PDF file from the example above can be considered just another representation of the client resource. Rails provides an easy and quite sleek way of doing “RESTful downloads”. Here’s how you can rewrite the example so that the PDF download is a part of the `show` action, without any streaming:

```
class

ClientsController < ApplicationController

# The user can request to receive this resource as HTML or PDF.

def

show

@client

= Client.find(params[
:id
])


respond_to
do
```

```
|format|

format.html

format.pdf { render
:pdf

=> generate_pdf(
@client
) }

end

end
end
```

In order for this example to work, you have to add the PDF MIME type to Rails. This can be done by adding the following line to the file `config/initializers/mime_types.rb`:

```
Mime::Type.register
"application/pdf"
,
:pdf
```

Configuration files are not reloaded on each request, so you have to restart the server in order for their changes to take effect.

Now the user can request to get a PDF version of a client just by adding ".pdf" to the URL:

```
GET /clients/1.pdf
```

12 Parameter Filtering

Rails keeps a log file for each environment in the `log` folder. These are extremely useful when debugging what's actually going on in your application, but in a live application you may not want every bit of information to be stored in the log file. You can filter certain request parameters from your log files by appending them to `config.filter_parameters` in the application configuration. These parameters will be marked [FILTERED] in the log.

```
config.filter_parameters <<
:password
```

13 Rescue

Most likely your application is going to contain bugs or otherwise throw an exception that needs to be handled. For example, if the user follows a link to a resource that no longer exists in the database, Active Record will throw the `ActiveRecord::RecordNotFound` exception.

Rails' default exception handling displays a "500 Server Error" message for all exceptions. If the request was made locally, a nice traceback and some added information gets displayed so you can figure out what went wrong and deal with it. If the request was remote Rails will just display a simple "500 Server Error" message to the user, or a "404 Not Found" if there was a routing error or a record could not be found. Sometimes you might want to customize how these errors are caught and how they're displayed to the user. There are several levels of exception handling available in a Rails application:

13.1 The Default 500 and 404 Templates

By default a production application will render either a 404 or a 500 error message. These messages are contained in static HTML files in the `public` folder, in `404.html` and `500.html` respectively. You can customize these files to add some extra information and layout, but remember that they are static; i.e. you can't use RHTML or layouts in them, just plain HTML.

13.2 `rescue_from`

If you want to do something a bit more elaborate when catching errors, you can use `rescue_from`, which handles exceptions of a certain type (or multiple types) in an entire controller and its subclasses.

When an exception occurs which is caught by a `rescue_from` directive, the exception object is passed to the handler. The handler can be a method or a `Proc` object passed to the `:with` option. You can also use a block directly instead of an explicit `Proc` object.

Here's how you can use `rescue_from` to intercept all `ActiveRecord::RecordNotFound` errors and do something with them.

```
class

ApplicationController < ActionController::Base

rescue_from ActiveRecord::RecordNotFound,
:with

=>
:record_not_found


private


def

record_not_found

render
:text

=>
"404 Not Found"
,
:status

=>
404

end
end
```

Of course, this example is anything but elaborate and doesn't improve on the default exception handling at all, but once you can catch all those exceptions you're free to do whatever you want with them. For example, you could create custom exception classes that will be thrown when a user doesn't have access to a certain section of your application:

```
class

ApplicationController < ActionController::Base

rescue_from User::NotAuthorized,
:with

=>
:user_not_authorized


private


def

user_not_authorized

flash[
:error
] =
"You don't have access to this section."

redirect_to
:back

end
end

class
```

```
ClientsController < ApplicationController

# Check that the user has the right authorization to access clients.

before_filter
:check_authorization


# Note how the actions don't have to worry about all the auth stuff.

def

edit

@client

= Client.find(params[
:id
])

end


private


# If the user is not authorized, just throw the exception.

def

check_authorization

raise

User::NotAuthorized
unless

current_user.admin?

end
end
```

Certain exceptions are only rescuable from the `ApplicationController` class, as they are raised before the controller gets initialized and the action gets executed. See Pratik Naik's **article** on the subject for more information.

14 Force HTTPS protocol

Sometime you might want to force a particular controller to only be accessible via an HTTPS protocol for security reasons. Since Rails 3.1 you can now use `force_ssl` method in your controller to enforce that:

```
class

DinnerController

force_ssl
end
```

Just like the filter, you could also passing `:only` and `:except` to enforce the secure connection only to specific actions.

```
class

DinnerController

force_ssl
:only

=>
:cheeseburger
```

```
# or

force_ssl
:except

=>
:cheeseburger
end
```

Please note that if you found yourself adding `force_ssl` to many controllers, you may found yourself wanting to force the whole application to use HTTPS instead. In that case, you can set the `config.force_ssl` in your environment file.

Feedback

You're encouraged to help improve the quality of this guide.

If you see any typos or factual errors you are confident to patch, please clone **docrails** and push the change yourself. That branch of Rails has public write access. Commits are still reviewed, but that happens after you've submitted your contribution. **docrails** is cross-merged with master periodically.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Check the **Ruby on Rails Guides Guidelines** for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please **open an issue**.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the **rubyonrails-docs mailing list**.