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

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 know:

- How to follow the flow of a request through a controller.
- How to restrict parameters passed to your controller.
- How and why to store data in the session or cookies.
- How to work with filters to execute code during request processing.
- How to use Action Controller's built-in HTTP authentication.
- How to stream data directly to the user's browser.
- How to filter sensitive parameters, so they do not appear in the application's log.
- How to deal with exceptions that may be raised during request processing.

What Does a Controller Do?

Action Controller is the C in MVC. After the router has determined which controller to use for a request, the 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 middleman 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 user data to the model.

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

Controller Naming Convention

The naming convention of controllers in Rails favors pluralization of the last word in the controller's name, although it is not strictly required (e.g. `ApplicationController`). For example, `ClientsController` is

preferable to `ClientController`, `SiteAdminsController` is preferable to `SiteAdminController` or `SitesAdminsController`, and so on.

Following this convention will allow you to use the default route generators (e.g. `resources`, etc) without needing to qualify each `:path` or `:controller`, and will keep named route helpers' usage consistent throughout your application. See Layouts and Rendering Guide for more details.

NOTE: The controller naming convention differs from the naming convention of models, which are expected to be named in singular form.

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 call its `new` method. Note that the empty method from the example above would work just fine because Rails will by default render the `new.html.erb` view unless the action says otherwise. By creating a new `Client`, the `new` method can make a `@client` instance variable accessible in the view:

```
def new
  @client = Client.new
end
```

The Layouts and 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 (with `private` or `protected`) which are not intended to be actions, like auxiliary methods or filters.

WARNING: Some method names are reserved by Action Controller. Accidentally redefining them as actions, or even as auxiliary methods, could result in `SystemStackError`. If you limit your controllers to only RESTful Resource Routing actions you should not need to worry about this.

NOTE: If you must use a reserved method as an action name, one workaround is to use a custom route to map the reserved method name to your non-reserved action method.

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 < ApplicationController
  # This action uses query string parameters because it gets run
  # by an HTTP GET request, but this does not make any difference
  # to how the parameters are accessed. The URL for
  # this action would look like this to list activated
  # clients: /clients?status=activated
  def index
    if params[:status] == "activated"
      @clients = Client.activated
    else
      @clients = Client.inactivated
    end
  end

  # This action uses POST parameters. They are most likely coming
  # from an HTML form that 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 "new"
    end
  end
end
```

Hash and Array Parameters

The `params` hash is not limited to one-dimensional keys and values. It can contain nested arrays and 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
```

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

The value of `params[:ids]` will now be `["1", "2", "3"]`. Note that parameter values are always strings; Rails does not attempt to guess or cast the type.

NOTE: Values such as `[nil]` or `[nil, nil, ...]` in `params` are replaced with `[]` for security reasons by default. See Security Guide for more information.

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]`.

The `params` object acts like a Hash, but lets you use symbols and strings interchangeably as keys.

JSON parameters

If you're writing a web service application, you might find yourself more comfortable accepting parameters in JSON format. If the "Content-Type" header of your request is set to "application/json", Rails will automatically load your parameters into the `params` hash, which you can access as you would normally.

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

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

Your controller will receive `params[:company]` as `{ "name" => "acme", "address" => "123 Carrot Street" }`.

Also, if you've turned on `config.wrap_parameters` in your initializer or called `wrap_parameters` in your controller, you can safely omit the root element in the JSON parameter. In this case, the parameters will be cloned and wrapped with a key chosen based on your controller's name. So the above JSON request can be written as:

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

And, assuming that you're sending the data to `CompaniesController`, it would then be wrapped within the `:company` key like this:

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

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

NOTE: Support for parsing XML parameters has been extracted into a gem named `actionpack-xml_parser`.

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 that captures the `:status` parameter in a "pretty" URL:

```
get '/clients/:status', to: '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", as if it were passed in the query string. Your controller will also receive `params[:action]` as "index" and `params[:controller]` as "clients".

`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 to `url_for` calls.

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

In a given request, the method is not actually called for every single generated URL. For performance reasons, the returned hash is cached, and there is at most one invocation per request.

Strong Parameters

With strong parameters, Action Controller parameters are forbidden to be used in Active Model mass assignments until they have been permitted. This means that you'll have to make a conscious decision about which attributes to permit for mass update. This is a better security practice to help prevent accidentally allowing users to update sensitive model attributes.

In addition, parameters can be marked as required and will flow through a predefined raise/rescue flow that will result in a 400 Bad Request being returned if not all required parameters are passed in.

```
class PeopleController < ActionController::Base
  # This will raise an ActiveModel::ForbiddenAttributesError exception
  # because it's using mass assignment without an explicit permit
  # step.
  def create
    Person.create(params[:person])
  end

  # This will pass with flying colors as long as there's a person key
  # in the parameters, otherwise it'll raise an
  # ActionController::ParameterMissing exception, which will get
  # caught by ActionController::Base and turned into a 400 Bad
  # Request error.
  def update
    person = current_account.people.find(params[:id])
    person.update!(person_params)
    redirect_to person
  end

  private
    # Using a private method to encapsulate the permissible parameters
    # is just a good pattern since you'll be able to reuse the same
    # permit list between create and update. Also, you can specialize
    # this method with per-user checking of permissible attributes.
    def person_params
      params.require(:person).permit(:name, :age)
```

```
    end
end
```

Permitted Scalar Values Calling `permit` like:

```
params.permit(:id)
```

permits the specified key (`:id`) for inclusion if it appears in `params` and it has a permitted scalar value associated. Otherwise, the key is going to be filtered out, so arrays, hashes, or any other objects cannot be injected.

The permitted scalar types are `String`, `Symbol`, `NilClass`, `Numeric`, `TrueClass`, `FalseClass`, `Date`, `Time`, `DateTime`, `StringIO`, `IO`, `ActionDispatch::Http::UploadedFile`, and `Rack::Test::UploadedFile`.

To declare that the value in `params` must be an array of permitted scalar values, map the key to an empty array:

```
params.permit(id: [])
```

Sometimes it is not possible or convenient to declare the valid keys of a hash parameter or its internal structure. Just map to an empty hash:

```
params.permit(preferences: {})
```

but be careful because this opens the door to arbitrary input. In this case, `permit` ensures values in the returned structure are permitted scalars and filters out anything else.

To permit an entire hash of parameters, the `permit!` method can be used:

```
params.require(:log_entry).permit!
```

This marks the `:log_entry` parameters hash and any sub-hash of it as permitted and does not check for permitted scalars, anything is accepted. Extreme care should be taken when using `permit!`, as it will allow all current and future model attributes to be mass-assigned.

Nested Parameters You can also use `permit` on nested parameters, like:

```
params.permit(:name, { emails: [] },
              friends: [ :name,
                         { family: [ :name ], hobbies: [] }])
```

This declaration permits the `name`, `emails`, and `friends` attributes. It is expected that `emails` will be an array of permitted scalar values, and that `friends` will be an array of resources with specific attributes: they should have a `name` attribute (any permitted scalar values allowed), a `hobbies` attribute as an array of permitted scalar values, and a `family` attribute which is restricted to having a `name` (any permitted scalar values allowed here, too).

More Examples You may want to also use the permitted attributes in your `new` action. This raises the problem that you can't use `require` on the root key because, normally, it does not exist when calling `new`:

```
# using `fetch` you can supply a default and use
# the Strong Parameters API from there.
params.fetch(:blog, {}).permit(:title, :author)
```

The model class method `accepts_nested_attributes_for` allows you to update and destroy associated records. This is based on the `id` and `_destroy` parameters:

```
# permit :id and :_destroy
params.require(:author).permit(:name, books_attributes: [:title, :id, :_destroy])
```

Hashes with integer keys are treated differently, and you can declare the attributes as if they were direct children. You get these kinds of parameters when you use `accepts_nested_attributes_for` in combination with a `has_many` association:

```
# To permit the following data:
# {"book" => {"title" => "Some Book",
#             "chapters_attributes" => { "1" => {"title" => "First Chapter"},
#                                        "2" => {"title" => "Second Chapter"}}}}

params.require(:book).permit(:title, chapters_attributes: [:title])
```

Imagine a scenario where you have parameters representing a product name, and a hash of arbitrary data associated with that product, and you want to permit the product name attribute and also the whole data hash:

```
def product_params
  params.require(:product).permit(:name, data: {})
end
```

Outside the Scope of Strong Parameters The strong parameter API was designed with the most common use cases in mind. It is not meant as a silver bullet to handle all of your parameter filtering problems. However, you can easily mix the API with your own code to adapt to your situation.

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 several of different storage mechanisms:

- `ActionDispatch::Session::CookieStore` - Stores everything on the client.
- `ActionDispatch::Session::CacheStore` - Stores the data in the Rails cache.

- `ActionDispatch::Session::ActiveRecordStore` - Stores the data in a database using Active Record (requires the `activerecord-session_store` gem).
- `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 to use the session. The cookie data is cryptographically signed to make it tamper-proof. And it is also encrypted so anyone with access to it can't read its contents. (Rails will not accept it if it has been edited).

The CookieStore can store around 4 kB 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 (such as 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 an initializer:

```
# 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 "rails g active_record:session_migration")
# Rails.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 an initializer:

```
# Be sure to restart your server when you modify this file.
Rails.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.
Rails.application.config.session_store :cookie_store, key: '_your_app_session', domain: ".e
```

Rails sets up (for the CookieStore) a secret key used for signing the session data in `config/credentials.yml.enc`. This can be changed with `bin/rails credentials:edit`.

```
# aws:
#   access_key_id: 123
#   secret_access_key: 345

# Used as the base secret for all MessageVerifiers in Rails, including the one protecting co
secret_key_base: 492f...
```

NOTE: Changing the secret_key_base when using the `CookieStore` will invalidate all existing sessions.

Accessing the Session

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

NOTE: 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, delete the key/value pair:

```
class LoginsController < ApplicationController
  # "Delete" a login, aka "log the user out"
  def destroy
    # Remove the user id from the session
    session.delete(:current_user_id)
    # Clear the memoized current user
    @_current_user = nil
    redirect_to root_url
  end
end
```

To reset the entire session, use `reset_session`.

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 passing error messages, etc.

The flash is accessed via the `flash` method. Like the session, the flash is represented as 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.delete(:current_user_id)
    flash[:notice] = "You have successfully logged out."
    redirect_to root_url
  end
end
```

Note that it is also possible to assign a flash message as part of the redirection. You can assign `:notice`, `:alert` or the general-purpose `:flash`:

```
redirect_to root_url, notice: "You have successfully logged out."
redirect_to root_url, alert: "You're stuck here!"
redirect_to root_url, flash: { referral_code: 1234 }
```

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 any error alerts or notices from the flash in the application's layout:

```
<html>
  <!-- <head/> -->
  <body>
    <% flash.each do |name, msg| -%>
      <%= content_tag :div, msg, class: name %>
    <% end -%>

    <!-- more content -->
  </body>
</html>
```

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

You can pass anything that the session can store; you're not limited to notices and alerts:

```
<% if flash[:just_signed_up] %>
  <p class="welcome">Welcome to our site!</p>
<% end %>
```

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

```
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
```

`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(client_params)
    if @client.save
      # ...
    else
      flash.now[:error] = "Could not save client"
      render action: "new"
    end
  end
end
```

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(author: cookies[:commenter_name])
  end

  def create
    @comment = Comment.new(comment_params)
    if @comment.save
      flash[:notice] = "Thanks for your comment!"
      if params[:remember_name]
        # Remember the commenter's name.
        cookies[:commenter_name] = @comment.author
      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 can set the key to `nil`, to delete a cookie value you should use `cookies.delete(:key)`.

Rails also provides a signed cookie jar and an encrypted cookie jar for storing sensitive data. The signed cookie jar appends a cryptographic signature on the cookie values to protect their integrity. The encrypted cookie jar encrypts the values in addition to signing them, so that they cannot be read by the end-user. Refer to the API documentation for more details.

These special cookie jars use a serializer to serialize the assigned values into strings and deserializes them into Ruby objects on read.

You can specify what serializer to use:

```
Rails.application.config.action_dispatch.cookies_serializer = :json
```

The default serializer for new applications is `:json`. For compatibility with old applications with existing cookies, `:marshal` is used when `serializer` option is not specified.

You may also set this option to `:hybrid`, in which case Rails would transparently deserialize existing (`Marshal`-serialized) cookies on read and re-write them in the `JSON` format. This is useful for migrating existing applications to the `:json` serializer.

It is also possible to pass a custom serializer that responds to `load` and `dump`:

```
Rails.application.config.action_dispatch.cookies_serializer = MyCustomSerializer
```

When using the `:json` or `:hybrid` serializer, you should beware that not all Ruby objects can be serialized as JSON. For example, `Date` and `Time` objects will be serialized as strings, and `Hash`es will have their keys stringified.

```
class CookiesController < ApplicationController
  def set_cookie
    cookies.encrypted[:expiration_date] = Date.tomorrow # => Thu, 20 Mar 2014
    redirect_to action: 'read_cookie'
  end

  def read_cookie
    cookies.encrypted[:expiration_date] # => "2014-03-20"
  end
end
```

It's advisable that you only store simple data (strings and numbers) in cookies. If you have to store complex objects, you would need to handle the conversion manually when reading the values on subsequent requests.

If you use the cookie session store, this would apply to the `session` and `flash` hash as well.

Rendering XML and JSON data

ActionController makes it extremely easy to render `XML` or `JSON` data. If you've generated a controller using scaffolding, it 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
```

You may notice in the above code that we're using `render xml: @users`, not `render xml: @users.to_xml`. If the object is not a String, then Rails will automatically invoke `to_xml` for us.

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 are registered via `before_action`. They 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_action :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
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 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_action`:

```
class LoginsController < ApplicationController
  skip_before_action :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 skip this filter only 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.

NOTE: Calling the same filter multiple times with different options will not work, since the last filter definition will overwrite the previous ones.

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 registered via `after_action`. They 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. Please note that "after" filters are executed only after a successful action, but not when an exception is raised in the request cycle.

"around" filters are registered via `around_action`. They 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 preview them easily by applying them within a transaction:

```
class ChangesController < ApplicationController
  around_action :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 also wraps rendering. In particular, in the example above, if the view itself reads from the database (e.g. via a scope), it will do so within the transaction and thus present the data to preview.

You can choose not to yield and build the response yourself, in which case the action will not be run.

Other Ways to Use Filters

While the most common way to use filters is by creating private methods and using `before_action`, `after_action`, or `around_action` to add them, there are two other ways to do the same thing.

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

```
class ApplicationController < ActionController::Base
  before_action do |controller|
    unless controller.send(:logged_in?)
      flash[:error] = "You must be logged in to access this section"
      redirect_to new_login_url
    end
  end
end
```

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

Specifically for `around_action`, the block also yields in the `action`:

```
around_action { |_controller, action| time(&action) }
```

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 cannot 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_action LoginFilter
end
```

```ruby
class LoginFilter
  def self.before(controller)
    unless controller.send(:logged_in?)
      controller.flash[:error] = "You must be logged in to access this section"
      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 must implement a method with the same name as the filter, so for the `before_action` filter, the class must implement a `before` method, and so on. The `around` method must `yield` to execute the action.

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_with model: @user do |form| %>
  <%= form.text_field :username %>
  <%= form.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.

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 `ActionDispatch::Request` and the `response` method returns a response object representing what is going to be sent back to the client.

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 Rails API documentation and Rack 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?`, `patch?`, `put?`, `delete?`, `head?`	Returns true if the HTTP method is GET/POST/PATCH/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.

`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.

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. To get a full list of the available methods, refer to the Rails API documentation and Rack Documentation.

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.

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"
```

NOTE: In the above case it would make more sense to use the `content_type` setter directly.

HTTP Authentications

Rails comes with three built-in HTTP authentication mechanisms:

- Basic Authentication

- Digest Authentication
- Token Authentication

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 AdminsController < ApplicationController
  http_basic_authenticate_with name: "humbaba", password: "5baa61e4"
end
```

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

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 AdminsController < ApplicationController
  USERS = { "lifo" => "world" }

  before_action :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.

HTTP Token Authentication

HTTP token authentication is a scheme to enable the usage of Bearer tokens in the HTTP `Authorization` header. There are many token formats available and describing them is outside the scope of this document.

As an example, suppose you want to use an authentication token that has been issued in advance to perform authentication and access. Implementing token authentication with Rails is quite easy and only requires using one method, `authenticate_or_request_with_http_token`.

```
class PostsController < ApplicationController
  TOKEN = "secret"

  before_action :authenticate

  private
    def authenticate
      authenticate_or_request_with_http_token do |token, options|
        ActiveSupport::SecurityUtils.secure_compare(token, TOKEN)
      end
    end
end
```

As seen in the example above, the `authenticate_or_request_with_http_token` block takes two arguments - the token and a `Hash` containing the options that were parsed from the HTTP `Authorization` header. The block should return `true` if the authentication is successful. Returning `false` or `nil` on it will cause an authentication failure.

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".

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 4 kB 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.

WARNING: 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.

TIP: 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.

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
```

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
```

NOTE: Configuration files are not reloaded on each request, so you have to restart the server 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
```

Live Streaming of Arbitrary Data

Rails allows you to stream more than just files. In fact, you can stream anything you would like in a response object. The `ActionController::Live` module allows you to create a persistent connection with a browser. Using this module, you will be able to send arbitrary data to the browser at specific points in time.

Incorporating Live Streaming Including `ActionController::Live` inside of your controller class will provide all actions inside the controller the ability to stream data. You can mix in the module like so:

```ruby
class MyController < ActionController::Base
  include ActionController::Live

  def stream
    response.headers['Content-Type'] = 'text/event-stream'
    100.times {
      response.stream.write "hello world\n"
      sleep 1
    }
  ensure
    response.stream.close
  end
end
```

The above code will keep a persistent connection with the browser and send 100 messages of `"hello world\n"`, each one second apart.

There are a couple of things to notice in the above example. We need to make sure to close the response stream. Forgetting to close the stream will leave the socket open forever. We also have to set the content type to `text/event-stream` before we write to the response stream. This is because headers cannot be written after the response has been committed (when `response.committed?` returns a truthy value), which occurs when you `write` or `commit` the response stream.

Example Usage Let's suppose that you were making a Karaoke machine, and a user wants to get the lyrics for a particular song. Each `Song` has a particular number of lines and each line takes time `num_beats` to finish singing.

If we wanted to return the lyrics in Karaoke fashion (only sending the line when the singer has finished the previous line), then we could use `ActionController::Live` as follows:

```ruby
class LyricsController < ActionController::Base
  include ActionController::Live

  def show
    response.headers['Content-Type'] = 'text/event-stream'
    song = Song.find(params[:id])

    song.each do |line|
      response.stream.write line.lyrics
      sleep line.num_beats
    end
  ensure
    response.stream.close
  end
end
```

The above code sends the next line only after the singer has completed the previous line.

Streaming Considerations Streaming arbitrary data is an extremely powerful tool. As shown in the previous examples, you can choose when and what to send across a response stream. However, you should also note the following things:

- Each response stream creates a new thread and copies over the thread local variables from the original thread. Having too many thread local variables can negatively impact performance. Similarly, a large number of threads can also hinder performance.
- Failing to close the response stream will leave the corresponding socket open forever. Make sure to call `close` whenever you are using a response stream.
- WEBrick servers buffer all responses, and so including `ActionController::Live` will not work. You must use a web server which does not automatically buffer responses.

Log 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.

Parameters Filtering

You can filter out sensitive 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
```

NOTE: Provided parameters will be filtered out by partial matching regular expression. Rails adds a list of default filters, including `:passw`, `:secret`, and `:token`, in the appropriate initializer (`initializers/filter_parameter_logging.rb`) to handle typical application parameters like `password`, `password_confirmation` and `my_token`.

Redirects Filtering

Sometimes it's desirable to filter out from log files some sensitive locations your application is redirecting to. You can do that by using the `config.filter_redirect` configuration option:

```
config.filter_redirect << 's3.amazonaws.com'
```

You can set it to a String, a Regexp, or an array of both.

```
config.filter_redirect.concat ['s3.amazonaws.com', /private_path/]
```

Matching URLs will be marked as '[FILTERED]'.

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:

The Default 500 and 404 Templates

By default, in the production environment the application will render either a 404, or a 500 error message. In the development environment all unhandled exceptions are simply raised. 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 style, but remember that they are static HTML; i.e. you can't use ERB, SCSS, CoffeeScript, or layouts for them.

`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 plain: "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_back(fallback_location: root_path)
    end
end

class ClientsController < ApplicationController
  # Check that the user has the right authorization to access clients.
  before_action :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
```

WARNING: Using `rescue_from` with `Exception` or `StandardError` would cause serious side-effects as it prevents Rails from handling exceptions properly. As such, it is not recommended to do so unless there is a strong reason.

NOTE: When running in the production environment, all `ActiveRecord::RecordNotFound` errors render the 404 error page. Unless you need a custom behavior you don't need to handle this.

NOTE: Certain exceptions are only rescuable from the `ApplicationController` class, as they are raised before the controller gets initialized, and the action gets executed.

Force HTTPS protocol

If you'd like to ensure that communication to your controller is only possible via HTTPS, you should do so by enabling the `ActionDispatch::SSL` middleware via `config.force_ssl` in your environment configuration.