**Controller**

**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.

Controllers are sort of the "doers" of MVC. It's the controller's job to ask the model layer to fetch data, to process user input to save new data, and to either build and send a response or redirect the user to a new path.

**Methods and Actions**

When you start a Rails project, Rails generates a blank class named ApplicationController which extends ActionController::Base (which is where a controller's super-powers come from). Your controller code will go in subclasses of ApplicationController.

When your application receives a request, the router will determine which controller and **action** (method) to run. The router will then instantiate an instance of the controller, and call the method that is named by the action.

# app/controllers/clients\_controller.rb

class ClientsController < ApplicationController

def index

render json: Client.all

end

def show

render json: Client.find(params[:id])

end

end

# config/routes.rb

MyLittleProject::Application.routes.draw do

resources :clients, only: [:index, :show]

end

As an example, if a user goes to the path /clients/123, the router will recognize that it should call ClientsController#show. It will lookup the Client by their id, and send back a JSON representation of the model object.

Note that controller naming convention is to **pluralize the name of the model**, and tack on "Controller".

**Strong 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: the query string parameters. These are available in a hash-like object returned by the ActionController::Base#params method:

class ClientsController < ApplicationController

# Sample request:

#

# GET /clients?status=activated

def index

# we normally leave out the `self.` part of `self.params`

if self.params[:status] == 'activated'

@clients = Client.activated

else

@clients = Client.unactivated

end

# render the clients to JSON and send them back to the browser

render json: @clients

end

end

The second type of parameter is the request body. Any kind of request may contain a body, but in practice only POST and PUT/PATCH do. This information usually comes from an HTML form which has been filled in by the user. Rails mixes the query string parameters and request body parameters together in the single #params method.

When the user submits a form with new attributes for a model, these are often stored as a nested hash in the params hash. Using a nested hash to create or update a model is called "mass-assignment". You may have done this when writing a seeds file. For example:

Post.create!({ title: 'My First Post', body: 'body body body' })

We wish we could write:

class PostsController < ApplicationController

# This action expects parameters to be uploaded through the request

# body. Sample request:

#

# POST /clients

# { post: {

# title: 'CATS',

# body: 'meow meow meow'

# }

# }

def create

# The JSON body will be parsed by Rails; params[:post] is the

# nested hash of post attributes. This can be passed to

# `Post.new` to "mass-assign" the values.

@post = Post.new(params[:post]) # WILL NOT WORK

@post.save!

render json: "Thanks for making the new post named #{@post.title}"

end

end

This won't work; an exception would be raised. In the controller, we want to be careful to restrict which attributes a user can assign. For example; we want to let a user change the title of the post, but we don't want them to change the author. The params hash has some built in methods to passlist certain attributes for mass assignment:

**NB** A passlist is essentially a list of elements that are allowed to pass through some sort of 'gate'. The 'gate' here being mass assignment. In contrast, a blocklist is a list of elements that are not allowed to pass through a 'gate'.

class PostsController < ApplicationController

def create

@post = Post.new(params[:post].permit(:title, :body))

@post.save!

render json: "Thanks for making the new post named #{@post.title}"

end

end

The #permit method of the hash-like object returned by #params "passlists" the title and body attributes, allowing them to be mass-assigned. All other non-passlisted attributes (e.g., author\_id) will be ignored.

**DRYing out Strong Parameters**

A good pattern for controlling which attributes are passlisted in the params hash is writing a helper method to deal with this. Example:

class CatsController < ActionController::Base

# Using "Cat.create(params[:cat])" would raise an

# ActiveModel::ForbiddenAttributes exception because it'd be using

# mass assignment without an explicit permit step.

# This is the recommended form:

def create

Cat.create!(cat\_params)

end

# This will pass with flying colors as long as there's a cat key in

# the parameters, otherwise it'll raise an

# ActionController::MissingParameter exception, which will get

# caught by ActionController::Base and turned into a 400 Bad Request

# reply.

def update

# params[:id] is a routing parameter; more in a sec!

@cat = Cat.find(params[:id])

@cat.update!(cat\_params)

render json: @cat

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 cat\_params

params.require(:cat).permit(:name, :age)

end

end

**Routing Parameters**

I lied. There's a third type of parameter.

Controller **member** routes like show, update, and delete all use the same path: e.g., /clients/:id. A GET, PATCH, or DELETE request will call ClientsController#show, ClientsController#update, or ClientsController#delete respectively. The intent is that these HTTP requests affect the Client with the id specified in the URL.

The controller needs to know the id so it can decide which Client record to show/update/delete. To tell the controller what object we are talking about, the router will set params[:id] to the matched id from the requested path. This is sometimes called a route fragment parameter.

Any named fragment of the url can be accessed this way, but :id is by far the most common.

**The request-response lifecycle**

When a client makes an HTTP request, the webserver receives it and hands it off to Rails. The Rails router looks up the controller action to call. As mentioned, it **creates an instance of the controller** to handle the response. The router then calls the appropriate method on the controller instance.

The controller instance then takes over the request processing. It runs the given method. As part of its work in the method, it should render a response or issue a redirect.

After issuing the response, the request is over and the connection between client-and-server is closed. The controller instance is discarded.

In particular, setting instance variables in the controller **does not affect the processing of future requests**. State is saved either in the database (server-side) or the cookie (client-side). Since instance variables will be lost (in fact the whole controller object is lost) immediately after the response is issued, you cannot use instance-variable data stored from previous requests.

In that case: why ever use instance variables in controllers? We'll see why in tomorrow's ERB reading.

**Resources**

* [Rails Guide on Controllers](http://guides.rubyonrails.org/action_controller_overview.html)
* [Strong Parameters Documentation](http://edgeapi.rubyonrails.org/classes/ActionController/StrongParameters.html)