Separation of concerns (Single Responsibility Principle) in Rails

Separation of concerns is one of the key goals of software development. It deals with good maintainance pattern. For this to be, every change have to be local. Then programmer has to separate things, the more he can do, avoiding bounded code.

To achieve this, he has to :

- **Separate out the things that change from those that stay the same**.
- **Program to an interface, not an implementation** : roughly speaking, code doesn't have to know which kind of object it deals with. Here duck typing is the king. The only thing you have to think of is building a good interface (a good "langage" to throw messages from one component to another). To conceive an interface, you have to think about the more general thing : car < vehicule < movable object. Best think about movable object.
- **Prefer composition over inheritance** : think about what your object has rather than what it is. And everything it has have to be another object.
- **Delegate, delegate, delegate** : your car doesn't start its engine, it says to its engine to start!
- **YAGNI** : you ain't gonna need it ! Do not anticipate what the future will be. Because doing this can lead you to a point from where the evolution to the actual future can be harder. You can't predict where the code will have to go...

Another thing related is to **avoid bounded code** : I will use **dependency injection** wherever it is possible together with maximazing **functional code** and minimazing **side_effects**.

> Explaination : a side effect occur when a method does anything else than returning a value (like mutating its arguments, or mutating anything else).

Functional code is easy and fast to test and easy to debug...

So here I am to apply all this advices/goals in a rails app.

Rails components

Controller

The central component. It is the part interacting with the browser, receiving http requests and responding by a content. Does it have to know about database structure ? No. Doesn't mean it doesn't have to deal with ActiveRecord instances? Only as far as it does not know it... Does it have to know about how to deal with the sql request ? No. It has to know who has to deal with the actual use case and which message have to be sent to it. Then it has to return the accurate response (depending of use-case return).

View

Does it have to know about database structure ? No. It has to know about what has to be displayed. So it has to receive something that is built from the database to respond to what needs to be displayed.

Model (ActiveRecord object)

Does it have to know about which kind of manipulation data are subject to ? Yes/No. Probably not ! This is what all this paper is about. It deals with storing to -- and validating or not ! -- and reading (performing requests) from database.

Who have to make a given action

Delegate, delegate, delegate rule applied to controller, tells us that it should only distribute the work to be done upon the right workers.

So controller have to send messages, but it doesn't manage how things have to be done. This is not its job.

I think I can divide the workflow in a few tasks.

- Processing POST, PATCH or DELETE actions,
- Collecting needed information from models,
- Presenting these informations within views,
- Handling actions that have nothing to do with data persistence or display (services)

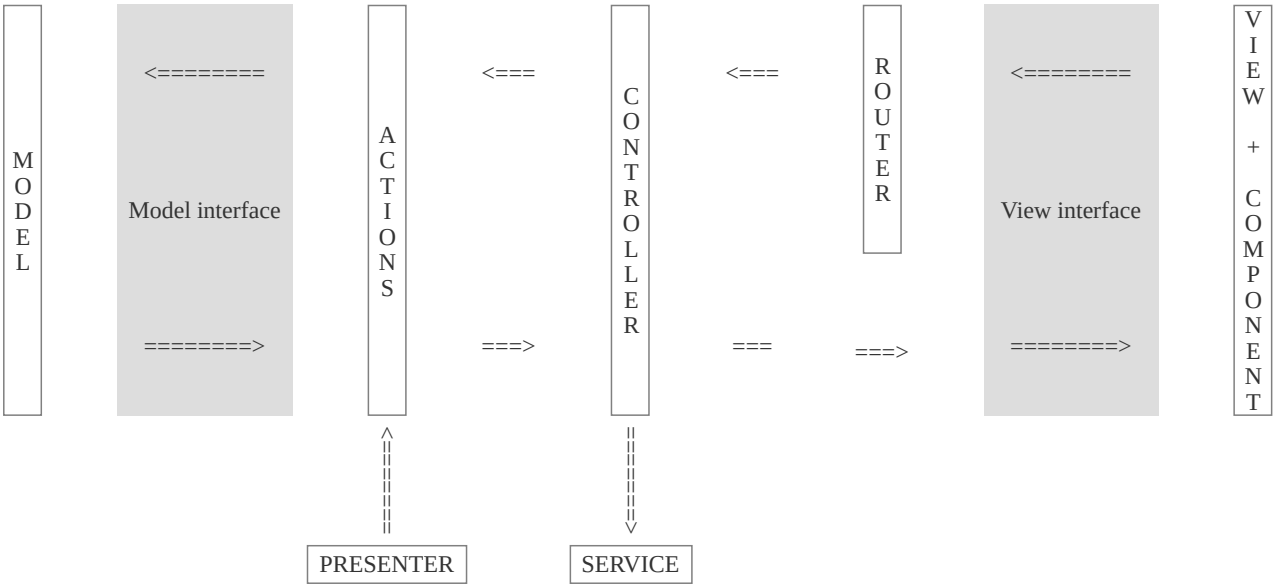
My solution

After a lot of tries, I finally decided that my good way of doing things is to use `dry-system` to abstract object instanciation and provide OO driven functional programming scheme. Then :

- each controller action is manage by a dedicated callable object.
- a dry-controller module enhance controller syntax to provide a clear interface with under the hood decoration behavior.
- a dry-exposer module manage all the way collected stuff are decorated for view rendering.
- each collected stuff presenting need is manage by a dedicated presenter module.
- Component objects encapsulate all common view rendering logic.
- partial templates are then logic free because all logic is either manage by a presenter or a component.

Following the convention naming : *name what it do or render, not what it is* `HistoryPresenter` , `ReportPresenter` , `HomeIcon` (or `HomeIconComponent`)

Separation of concerns :



Final tree and ApplicationRecord extension

```
app
├── actions
│   ├── entries
│   │   ├── index.rb
│   │   └── report.rb
│   ├── entries.rb
├── components
│   ├── component.rb
│   ├── icon.rb
│   └── to_currency.rb
├── exposers
│   ├── entries_exposer.rb
│   └── inputs_exposer.rb
├── lib
│   ├── context.rb
│   ├── dry_controller.rb
│   ├── dry_exposer.rb
│   ├── params_manager.rb
│   ├── session_manager.rb
├── presenters
│   ├── entries_presenter.rb
│   ├── fallback.rb
│   └── report_presenter.rb
```

Controllers

A way to handle parameter permissions within controllers :

```
# app/controllers/application_controller.rb
class ApplicationController < ActionController::Base
  include ParamsManager
end

# app/lib/params_manager.rb
module ParamsManager
  protected

  def permitted_params
    set_model_vars
    sanitize_decimal_values # YES, this is a controller concern !
    params
      .require( @model_name )
      .permit( @model.permitted_attributes )
  end

  private

  def set_model_vars
    @model_name = params[:controller].singularize.to_sym
    @model = Object.const_get( @model_name.to_s.camelize )
  end

  def sanitize_decimal_values
    return unless @model.respond_to? :numerical_attributes
    @model.numerical_attributes&.each do |field|
      params[ @model_name ]
        .fetch( field, '' )
        .to_s
        .gsub!( ',', '.' )
    end
  end

end

# app/models/stay.rb
class Stay < ApplicationRecord
  def self.permitted_attributes
    [ :title, :name, :whatever ]
  end
  def self.numerical_attributes
    [ :price ]
  end
end
```

No more need for a `stay_params` method...

Base classes and modules

All this could be encapsulated inside a `Logicore` module ... or not.

Dry-system initializer

```
# config/initializers/container.rb
require "dry/system"
class BudgetAppContainer < Dry::System::Container
  configure do |config|
    config.root = Rails.root
    config.component_dirs.add './app/actions'
    config.component_dirs.add './app/exposers'
  end

  self.finalize! if Rails.env.production?
end
```

dry-controller

```
# app/lib/dry_controller.rb
module DryController
  def expose( action, entity = nil, *args, **options )
    @action = action
    collected = args.include?( :expose_only ) ? entity :
        BudgetAppContainer[ action ].call( entity, Context.new( options ) )

    name = result_name( options )
    instance_variable_set(
      "@#{name}", add_decoration_to( collected, name )
    )
  end

  alias perform expose

  private
  def result_name( options )
    options.fetch( :as, remaining_action )
  end

  def add_decoration_to( collected, name )
    # delegates to an exposer knowing how to decorate !!
    begin # check existence
      BudgetAppContainer[ "#{model}_exposer" ].call( collected, name)
    rescue => e
      MyLogger.debug { "No key #{model}_exposer in BudgetAppContainer : #{e}" }
      collected
    end
  end

  def model
    @action.split('.').first
  end

  def remaining_action
    @action.split('.')[1..].join('.')
  end
end
```

dry-exposer

```
# app/lib/dry_exposer.rb
class DryExposer
  @@mapping = Hash.new

  def self.expose( *actions, with: )
    added = actions.inject( Hash.new ) do |res, action|
      res.merge!( action => with )
    end
    if @@mapping.fetch( self, false )
      @@mapping[ self ].merge! added
    else
      @@mapping.merge! self => added
    end
  end

  def call( collected, name )
    case mod = @@mapping[ self.class ].fetch( name.to_sym, Module.new )
    when Hash
      collected.map do |c|
        MyLogger.debug { "extending #{c} with #{mod.fetch( :single, 'nothing' )}" }
        c.extend mod.fetch( :single, Module.new ), Fallback
      end.then { |collected|
        MyLogger.debug { %Q(
                         extended collection #{collected} with
                         #{mod.fetch( :collection, 'nothing' )}
                         ).squish
        }
        collected.extend mod.fetch( :collection, Module.new ), Fallback
      }
    else
      MyLogger.debug { "extending #{collected} with #{mod}" }
      collected.extend mod, Fallback
    end
  end
end
```

Context class

```
# A way to transform a hash to an object.. holding hash behaviour.. read-only by design.
# Works only with symbol or string keys.
class Context
  def initialize( context = {} )
    @context = context
  end

  def method_missing( name, *args )
    args.empty? ? get( name ) : super
  end

  def []( key ); get( key ); end

  private
  def get( key )
    @context.fetch( key.to_sym, @context.fetch( key.to_s, nil ) )
  end
end
```

Actions, Presenters and Exposers

```
# app/actions/entries.rb
module Entries
  class Action
    @@ended_reports = true
    @@reports_end_at = Date.today

    protected

    def get_month_query( query, context )
      context.month ?
        query.where( 'extract( month from date )::int = ?', context.month )
      : query
    end

    def get_date_query( query )
      @@ended_reports ?
        query.where( "date <= ?", @@reports_end_at ) : query
    end
  end
end

# app/actions/entries/index.rb
module Entries
  class Index < Entries::Action
    attr_reader :context, :previous_year

    def call( entity, context )
      previous_year = Date.today.year - 1
      Budget
        .find( context.budget_id )
        .entries
        .where( "year >= ?", previous_year )
        .then { |query| get_date_query( query ) }
        .select( "year, extract( month from date )::int as month" )
        .order( year: :desc )
        .order( month: :desc )
        .group( :year, :month )
    end
  end
end
```

```
# app/presenters/fallback.rb
module Fallback
  def method_missing( name, *args )
    return super unless name.to_s =~ /^the_/
    send( name.to_s.gsub( "the_", "" ).to_sym, *args )
  end
end
```

```ruby
# app/presenters/entries_presenter.rb
module EntriesPresenter

  def turbo_frame_id
    "report-#{year}-#{month.to_i}"
  end

  def the_report_title
    I18n.l(
      Date.today.beginning_of_month.change( year: year, month: month ), format: :month
    )
  end

  def the_actual_to_target
    return the_use_sum / the_income_sum if the_income_sum > 0
    the_use_sum == 0 ? 0 : 2
  end

  def the_income_sum
    respond_to?( :income_sum ) ? income_sum : 0
  end

  def the_use_sum
    respond_to?( :use_sum ) ? use_sum : 0
  end
end
```

```ruby
# app/exposers/entries_exposer.rb
class EntriesExposer < DryExposer
  expose :home_content, with: HomePresenter
  expose :reports, :report, :cat_report,
    with: { collection: ReportPresenter, single: EntriesPresenter }
end
```

Typical controller methods

```ruby
def index
  expose 'inputs.history', as: :inputs,
    budget_id: current_budget_id,
    date: params.fetch( :date, Date.today ).to_date
end

def create
  perform 'inputs.create', as: :input,
    params: permitted_params,
    profile: current_budget.profile

  unless @input.valid?
    render :new, status: :unprocessable_entity and return
  end

  flash.notice = 'Successfully created !'
  redirect_to :home, status: :see_other
end

def show
  expose 'inputs.input', current_input, :expose_only
end
```

Component class and subclasses

```
class Component
  delegate :render, to: :view_context
  attr_reader :view_context

  def initialize; end
  def partial_name; nil; end

  def rendered_object
     inline_template? ?
      { inline: erb_template } :
      { partial: [ partial_folder, partial_name ].join }
  end

  def render_in( view_context, &block )
    @view_context = view_context # hence view_context known in render? method
    return unless render?
    render **rendered_object, locals: provided_vars
  end

  def render?; true; end
  def provided_vars; {}; end

  private
  def inline_template?; respond_to?( :erb_template ); end
  def partial_folder; "components/"; end
end
```

A typical Component

```
class Icon < Component
  attr_reader :name, :options
  def erb_template
    <<~ERB
      <%= content_tag icon_tag, '', icon: icon_name, **options %>
    ERB
  end

  def initialize( name, **options ); @name, @options = name, options; end
  def provided_vars
    { icon_name: "mdi:#{name}", icon_tag: "iconify-icon", options: options }
  end
end
```

A non-typical Component

```
class ToCurrency < Component
  attr_reader :value, :default, :session

  def initialize( value, default = '_' ); @value, @default = value, default; end

  def render_in( view_context, &block ) # overriding super class
    return default unless value
    @view_context = view_context
    @session = view_context.session
    view_context.number_to_currency( value, **format, unit: unit )
  end
  private
  def format; CurrencyManager::CURRENCY_FORMATS[session.currency_format.to_sym]; end
  def unit; session.currency_unit; end
end
```

A typical View

```
<% @index_content.each do |item| %>
  <%= render EditIcon.new %>
  <%= item.the_title %>
  <%= render ToCurrency.new( item.the_value ) %>
<% end %>
```

More encapsulated or isolated stuff : Session

Providing methods to access session content permits to treat default values or on-the-way storage..

```
class ApplicationController < ActionController::Base
  before_action :extend_session

  def extend_session
    session.extend SessionManager
    session.controller = self
  end
end
```

```
module SessionManager
  attr_reader :controller

  def controller=( controller ); @controller = controller; end

  def update!
    budget = get_budget
    self[:budget_id] = budget.id
    ...
  end

  # under here, accessor methods... They provide a way to handle missing key
  def budget_id
    store_budget_id_in_session unless self[:budget_id]
    self[:budget_id]
  end

  def prefered_language
    self[:prefered_language] || budget_prefered_language
  end

  def currency_unit; self[:currency_unit]; end
  def currency_format; self[:currency_format]; end

  private
  def get_budget
    controller.current_user.budgets&.first
  end

  def store_budget_id_in_session
    ...; self.update!
  end

  def budget_prefered_language
    # provide default
  end
end
```