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. This doesn't mean it does not deal with ActiveRecord instances. It means that it does not know what it deals with... Does it has 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 has 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 has 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.

- Performing POST, PATCH or DELETE actions,
- Collecting needed information from models,
- Exposing 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 :

- Defining `Actions` classes to perform database related operations. Hence, model only contains validations and, eventually, basic scoping methods like exploring relations.. (finding budget owner..)
- Defining `Exposers` classes to provide dedicated locals to views. These exposers extend basic data (provided by actions or not) with dedicated presenter modules. Only presentationnal stuff lives here !
- To dry view code together with extracting logic from views, I use `Component` objects. They encapsulate all the logic needed inside views so non-encapsulated-rails-helpers are not anymore needed. Better is using `Presenter` ; I keep `Component` objects only for shared behaviour (like formatting currency values, rendering icons).
- Defining `Services` for all non database related treatment (like sending a mail or connecting to a payment platform)

Separation of concerns :

- Actions job is only to execute model related operations and almost all model related operation lives in an Action. Some lives in model extending capabilities of model instances..
- Only Exposers defines the locals that view will use and they only do that.
- Controllers only deal with routing behaviour, calling dedicated actions, services and exposers.
- Only views and components contains html and css code.

Final tree and ApplicationRecord extension

```
app
├── actions
│   ├── budgets
│   │   ├── create.rb
│   │   └── update.rb
│   ├── inputs
│   │   ├── history.rb
│   │   └── update.rb
│   ├── budgets.rb
│   └── inputs.rb
├── components
│   ├── bar_graph.rb
│   ├── component.rb
│   ├── field_label.rb
│   ├── flash.rb
│   ├── icon.rb
│   └── to_currency.rb
├── exposers
│   ├── budgets
│   │   ├── home.rb
│   │   └── edit.rb
│   ├── inputs
│   │   ├── index.rb
│   │   └── edit.rb
│   ├── budgets.rb
│   ├── exposer.rb
│   └── inputs.rb
├── lib
│   ├── context.rb
│   ├── dry_controller.rb
│   ├── params_manager.rb
│   ├── result.rb
│   └── session_manager.rb
├── presenters
│   ├── budget_presenter.rb
│   ├── entry_presenter.rb
│   ├── fallback.rb
│   ├── home_presenter.rb
│   ├── input_presenter.rb
│   └── report_presenter.rb
├── services
│   └── charge_bee_gateway.rb
```

Drying 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.capitalize )
  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...

Defining locals for views

Rails controllers implicitely render views named from controller method name. An explicit call is somewhere better to clarify code. Together with this, controllers automatically provide its instance variables to views. Here too, impliciteness is the law and it could be better to not pass instance variables and work with locals... Here the way I followed :

```
module DryController
  def execute( action, entity = nil, **options )
    res = BudgetAppContainer[ action ].call( entity, Context.new( options ) )
    block_given? ? yield( res ) : res
  end

  # short cuts...
  def locals_for( action, entity = nil, **options, &block )
    # dedicated to call exposers..
    execute "exposers.#{action}", entity, **options, &block
  end

  def perform( action, entity = nil, *args, **options, &block )
    # dedicated to call actions..
    execute "actions.#{action}", entity, **options, &block
```

```
  end
end
```

Typical controller methods calling processor or collector

```
def create
  perform 'inputs.create', params: permitted_params do |result|
    result.isSuccess { redirect_to :home, status: :see_other }

    result.isFailure do |input|
      locals: locals_for( 'inputs.new', input: input ) do |locals|
        render :new, locals: locals, status: :unprocessable_entity
      end
    end
  end
end

def index
  render locals: locals_for(
    'inputs.index',
    current: current,
    date: params.fetch( :date, Date.today ).to_date
  )
end
```

Base modules and classes

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

Context class

```ruby
# 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
```

Result class

```ruby
# A way to improve code readability when treating method return
class Result
  attr_reader :content

  def initialize( content, *args )
    @content = args.empty? ? content : args.unshift( content )
    @final = nil
  end

  def self.[]( content = true, *args )
    new( content, *args )
  end

  # on the fly Result creation
  # Result.set( :Success, 'hello' )
  # leads to a Success object subclassing Result..
  def self.set( klass, content = true, *args )
    set_result_class( klass )[ content, *args ]
  end

  def call( *args, **options, &block )
    return content unless content.respond_to? :call
    content.call( *args, **options, &block )
  end

  def method_missing( name, *args, **options, &block )
    return super unless name.to_s =~ /^is/
    is( name.to_s.gsub( /^is/, '' ), &block )
  end

  def is( klass, &block )
    klass = set_result_class( klass )
    @final ||= ( yield content if is_a? klass )
  end

  private

  def to_class( name )
    # Better not to use Rails specific code (constantize !)
    Object.const_get( name )
  end
```

```
  def self.set_result_class( klass )
    case klass
    when Symbol, String
      Object.const_defined?( klass ) ?
        Object.const_get( klass ) :
        Object.const_set( klass, Class.new( Result ) )
    when self.class
      klass
    end
  end

  def set_result_class( klass )
    self.class.set_result_class( klass )
  end
end
# Expected syntax
# 1/
# res = Result[ 'my name' ]
# res.call -> 'my_name'
# res.content -> [ 'my_name' ]
# res.is( Result ) { |str| here str = 'my_name' }
# res.isResult { |str| here str = 'my_name' }
#
# 2/
# res = Result[ 'my_name', 'your_name' ]
# res.call -> 'my_name', 'your_name'
# res.content -> [ 'my_name', 'your_name' ]
# res.is( Result ) { |str_a, str_b| here str_a = 'my_name', str_b = 'your_name' }
# res.isResult { |str_a, str_b| here str_a = 'my_name', str_b = 'your_name' }
#
# 3/
# res = Result[ -> ( *args, **options, &block ) { lambda code } ]
# res.call( a, b, c: 'hello' )
#   -> execute lambda with args = a, b ; options = c.hello
# res.call( a, b, c: 'hello' ) { |x, y| block code }
#   -> execute lambda with args = a, b ; options = c.hello and execute block if
#   lambda contains a 'yield x, y' line...
#
# Describing results :
# method_yielding_a_result do |access|
#   access.isGranted { |*content| ... }
#   access.isDenied { |*content| ... }
# end
# or
# method_yielding_a_result do |access|
#   access.is( :Granted ) { |*content| ... }
#   access.is( :Denied ) { |*content| ... }
# end
```

Exposer class

```
# provide 'view_locals' and 'expose' DSL methods to be used in subclasses
class Exposer
  attr_reader :entity, :context

  # Provide an 'expose' class method to be used in child classes.
  # This is the main method to be called on those classes to provide the
  # desired locals for views.
  #
  # Many syntaxes are supported :
  #
  # 1/
  # expose :name, obj
  # --> a local named 'name' will be available in view, its content being
  # either obj or obj.call(entity, context) (if obj is callable)
  #
  # 2/
  # expose :name do
  #   obj
  # end
```

```ruby
# --> a local named 'name' will be available in view, its content being
# obj.call( entity, context ). This form is mandatory if the callable object
# is only available in instances (like while using include Deps[ 'obj' ])
#
# By default, a local named 'name' will be decorated with
# NamePresenter or NamesPresenter if it is enumerable
# This name convention could be override providing a with: optionnal argument
# examples :
# expose :name, with: [ :report, :entry ]
# expose :name, with: [ :inputs ]
def self.expose( key, stuff = nil, with: nil, &block )
  define_method key_to_meth( key ) do
    Hash[ stuff: stuff, with: with ]
  end
  define_method key_to_block( key ), &block if block_given?
end

def call( entity, context )
  @entity, @context = entity, context
  key_methods.inject( {} ) do |result, meth|
    key = meth_to_key( meth )
    result.merge( key => exposing( key, send( meth ) ) )
  end
end

private

def exposing( key, hash )
    block = send( key_to_block( key ) ) if respond_to?( key_to_block( key ) )
    stuff = hash[:stuff] || block
    decorate( key, exposure( key, stuff ), hash[:with] )
end

def exposure( key, stuff )
  return context[ key ] if context[ key ]
  stuff.respond_to?( :call ) ? stuff.call( entity, context ) : stuff
end

def decorate( key, collected, decorators )
  collection_decorator, item_decorator = set_decorators( key, decorators )
  if mod = key_to_presenter( collection_decorator )
    collected.extend mod, Fallback
    MyLogger.debug( tag: "Exposer" ) { "decorating #{collected} with #{mod}" }
  end

  if collected.respond_to?( :each )
    if mod = key_to_presenter( item_decorator )
      collected.each { |x| x.extend mod, Fallback }
      MyLogger.debug( tag: "Exposer" ) { "decorating #{collected}-items with #{mod}" }
    end
  end
  collected
end

def set_decorators( key, decorators )
  return [ key.to_s, key.to_s.singularize ] unless decorators
  case decorators
  when Array
    decorators
  else
    [ decorators.to_s, decorators.to_s.singularize ]
  end
end

def key_to_presenter( key )
  begin
    Object.const_get( "#{to_const_name( key )}Presenter" )
  rescue
    nil
  end
end
```

```
  def to_const_name( key )
    key.to_s.split('_').map(&:capitalize).join
  end

  # intermediate methods naming
  def self.prefix
    "local_"
  end

  def prefix
    self.class.prefix
  end

  def self.key_to_meth( key )
    [ prefix, key ].join.to_sym
  end

  def meth_to_key( meth )
    meth.to_s.gsub( prefix, "" ).to_sym
  end

  def self.key_to_block( key )
    "#{key}_block".to_sym
  end

  def key_to_block( key )
    self.class.key_to_block( key )
  end

  def key_methods
    methods.grep( /^#{prefix}/ )
  end
end
```

Action classes and subclasses

```
#app/actions/inputs.rb
module Inputs
  class Action
    # common stuff
  end
end

#app/actions/inputs/create.rb
module Inputs
  class Create < Inputs::Action
    def call( entity, context )
      input = Input.create( context.params )
      if input.valid?
        create_journal_entries_for( input )
        status = :Success
      else
        status = :Failure
      end
      Result.set status, input
    end
  end
end
```

Exposer classes and subclasses

```
# app/exposers/entries/report.rb
module Entries
  class Report < Exposer
    include Deps[ 'actions.entries.report_content', 'actions.entries.cat_report' ]

      expose :report, with: [ :report, :entry ] do
        report_content
      end
      expose( :cat_report, with: [ :report, :entry ] ) { cat_report }
    end

  end
end
```

```
module Budgets
  class Edit < Exposer
    include Deps[ 'actions.budgets.current' ]

    expose :budget do
      context.id ? current.call( entity, context ) : nil
    end
  end
end
```

Presenter modules

```
module BudgetPresenter
  # methods to extend budget instances..
end
```

```
module Fallback
# Provide a 'the_' prefixing behaviour to all model attributes so views do not
have to know database terminology.. This resolves 'the_attribute' to
'attribute' if presenter does not implement 'the_attribute' method..
  def method_missing( name, *args )
    return super unless name.to_s =~ /^the_/
    send( name.to_s.gsub( "the_", "" ).to_sym, *args )
  end
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
```