

## Pattern

A collection of lightweight, standardized, rails-oriented patterns used by RubyOnRails Developers @ Selleo

- Query - complex querying on active record relation
- Service - useful for handling processes involving multiple steps
- Collection - when in need to add a method that relates to the collection as whole
- Form - when you need a place for callbacks, want to replace strong parameters or handle virtual/composite resources
- Calculation - when you need a place for calculating a simple value (numeric, array, hash) and/or cache it
- Rule and Ruleset - when you need a place for conditional logic

## Installation

```
1 # Gemfile
2
3 #...
4 gem "rails-patterns"
5 #...
```

Then `bundle install`

## Query

### When to use it

One should consider using query objects pattern when in need to perform complex querying on active record relation. Usually one should avoid using scopes for such purpose. As a rule of thumb, if scope interacts with more than one column and/or joins in other tables, it should be moved to query object. Also whenever a chain of scopes is to be used, one should consider using query object too. Some more information on using query objects can be found in this article.

---

## Assumptions and rules

- Query objects are always used by calling class-level `.call` method
- Query objects require `ActiveRecord::Relation` or `ActiveRecord::Base` as constructor argument
- Default relation (see above) can be defined by using `queries` macro
- Query objects have to implement `#query` method that returns `ActiveRecord::Relation`
- Query objects provide access to consecutive keyword arguments using `#options` hash

## Other

Because of the fact, that QueryObject implements `.call` method, those can be used to construct scopes if required. (read more...)

## Examples

### Declaration

```
1 class RecentlyActivatedUsersQuery < Patterns::Query
2   queries User
3
4   private
5
6   def query
7     relation.active.where(activated_at: date_range)
8   end
9
10  def date_range
11    options.fetch(:date_range, default_date_range)
12  end
13
14  def default_date_range
15    Date.yesterday.beginning_of_day..Date.today.end_of_day
16  end
17 end
```

### Usage

```
1 RecentlyActivatedUsersQuery.call
2 RecentlyActivatedUsersQuery.call(User.without_test_users)
3 RecentlyActivatedUsersQuery.call(date_range: Date.today.
4   beginning_of_day..Date.today.end_of_day)
5 RecentlyActivatedUsersQuery.call(User.without_test_users, date_range:
6   Date.today.beginning_of_day..Date.today.end_of_day)
7
8 class User < ApplicationRecord
9   scope :recently_activated, RecentlyActivatedUsersQuery
```

## Service

### When to use it

Service objects are commonly used to mitigate problems with model callbacks that interact with external classes (read more...). Service objects are also useful for handling processes involving multiple steps. E.g. a controller that performs more than one operation on its subject (usually a model instance) is a possible candidate for Extract ServiceObject (or Extract FormObject) refactoring. In many cases service object can be used as scaffolding for replace method with object refactoring. Some more information on using services can be found in this article.

### Assumptions and rules

- Service objects are always used by calling class-level `.call` method
- Service objects have to implement `#call` method
- Calling service object's `.call` method executes `#call` and returns service object instance
- A result of `#call` method is accessible through `#result` method
- It is recommended for `#call` method to be the only public method of service object (besides state readers)
- It is recommended to name service object classes after commands (e.g. `ActivateUser` instead of `UserActivation`)

### Other

A bit higher level of abstraction is provided by `business_process` gem.

### Examples

#### Declaration

```
1 class ActivateUser < Patterns::Service
2   def initialize(user)
3     @user = user
4   end
5
6   def call
7     user.activate!
8     NotificationsMailer.user_activation_notification(user).deliver_now
9     user
10  end
11 end
```

---

```
10   end
11
12   private
13
14   attr_reader :user
15 end
```

#### Usage

```
1   user_activation = ActivateUser.call(user)
2   user_activation.result # <User id: 5803143, email: "tony@patterns.dev"
   ...
```

## Collection

### When to use it

One should consider using collection pattern when in need to add a method that relates to the collection as a whole. Popular example for such situation is for paginated collections, where for instance `#current_page` getter makes sense only in collection context. Also collections can be used as a container for mapping or grouping logic (especially if the mapping is not 1-1 in terms of size). Collection might also act as a replacement for models not inheriting from `ActiveRecord::Base` (e.g. `StatusesCollection`, `ColorsCollection` etc.). What is more, collections can be used if we need to encapsulate “flagging” logic - for instance if we need to render a separator element between collection elements based on some specific logic, we can move this logic from view layer to collection and yield an additional flag to control rendering in view.

### Assumptions and rules

- Collections include `Enumerable`
- Collections can be initialized using `.new`, `.from` and `.for` (aliases)
- Collections have to implement `#collection` method that returns object responding to `#each`
- Collections provide access to consecutive keyword arguments using `#options` hash
- Collections provide access to first argument using `#subject`

### Examples

#### Declaration

```
1 class ColorsCollection < Patterns::Collection
```

---

```

2  AVAILABLE_COLORS = { red: "#FF0000", green: "#00FF00", blue: "#0000FF"
3    " }
4  private
5
6  def collection
7    AVAILABLE_COLORS
8  end
9  end
10
11 class CustomerEventsByTypeCollection < Patterns::Collection
12   private
13
14   def collection
15     subject.
16     events.
17     group_by(&:type).
18     transform_values{ |events| events.map{ |e| e.public_send(options.
19       fetch(:label_method, "description")) }}
20   end
21 end

```

#### Usage

```

1 ColorsCollection.new
2 CustomerEventsByTypeCollection.for(customer)
3 CustomerEventsByTypeCollection.for(customer, label_method: "name")

```

## Form

### When to use it

Form objects, just like service objects, are commonly used to mitigate problems with model callbacks that interact with external classes (read more...). Form objects can also be used as replacement for `ActionController::StrongParameters` strategy, as all writable attributes are re-defined within each form. Finally form objects can be used as wrappers for virtual (with no model representation) or composite (saving multiple models at once) resources. In the latter case this may act as replacement for `ActiveRecord::NestedAttributes`. In some cases `FormObject` can be used as scaffolding for replace method with object refactoring. Some more information on using form objects can be found in this article.

### Assumptions and rules

- Forms include `ActiveModel::Validations` to support validation.

- 
- Forms include `Virtus.model` to support `attribute` static method with all corresponding capabilities.
  - Forms can be initialized using `.new`.
  - Forms accept optional resource object as first constructor argument.
  - Forms accept optional attributes hash as latter constructor argument.
  - Forms have to implement `#persist` method that returns falsey (if failed) or truthy (if succeeded) value.
  - Forms provide access to first constructor argument using `#resource`.
  - Forms are saved using their `#save` or `#save!` methods.
  - Forms will attempt to pre-populate their fields using `resource#attributes` and public getters for `resource`
  - Form's fields are populated with passed-in attributes hash reverse-merged with pre-populated attributes if possible.
  - Forms provide `#as` builder method that populates internal `@form_owner` variable (can be used to store current user).
  - Forms allow defining/overriding their `#param_key` method result by using `.param_key` static method. This defaults to `#resource#model_name#param_key`.
  - Forms delegate `#persisted?` method to `#resource` if possible.
  - Forms do handle  `ActionController::Parameters`  as attributes hash (using `to_unsafe_h`)
  - It is recommended to wrap `#persist` method in transaction if possible and if multiple model are affected.

## Examples

### Declaration

```
1 class UserForm < Patterns::Form
2   param_key "person"
3
4   attribute :first_name, String
5   attribute :last_name, String
6   attribute :age, Integer
7   attribute :full_address, String
8   attribute :skip_notification, Boolean
9
10  validate :first_name, :last_name, presence: true
11
12  private
13
14  def persist
15    update_user and
16    update_address and
17    deliver_notification
```

---

```

18   end
19
20   def update_user
21     resource.update_attributes(attributes.except(:full_address, :
22       skip_notification))
23   end
24   def update_address
25     resource.address.update_attributes(full_address: full_address)
26   end
27
28   def deliver_notification
29     skip_notification || UserNotifier.user_update_notification(user,
30       form_owner).deliver
31   end
32
33   class ReportConfigurationForm < Patterns::Form
34     param_key "report"
35
36     attribute :include_extra_data, Boolean
37     attribute :dump_as_csv, Boolean
38     attribute :comma_separated_column_names, String
39     attribute :date_start, Date
40     attribute :date_end, Date
41
42     private
43
44     def persist
45       SendReport.call(attributes)
46     end
47   end

```

### Usage

```

1 form = UserForm.new(User.find(1), params[:person])
2 form.save
3
4 form = UserForm.new(User.new, params[:person]).as(current_user)
5 form.save!
6
7 ReportConfigurationForm.new
8 ReportConfigurationForm.new({ include_extra_data: true, dump_as_csv:
   true })

```

---

## Calculation

### When to use it

Calculation objects provide a place to calculate simple values (i.e. numeric, arrays, hashes), especially when calculations require interacting with multiple classes, and thus do not fit into any particular one. Calculation objects also provide simple abstraction for caching their results.

### Assumptions and rules

- Calculations have to implement `#result` method that returns any value (result of calculation).
- Calculations do provide `.set_cache_expiry_every` method, that allows defining caching period.
- When `.set_cache_expiry_every` is not used, result is not being cached.
- Calculations return result by calling any of following methods: `.calculate`, `.result_for` or `.result`.
- First argument passed to calculation is accessible by `#subject` private method.
- Arguments hash passed to calculation is accessible by `#options` private method.
- Caching takes into account arguments passed when building cache key.
- To build cache key, `#cache_key` of each argument value is used if possible.
- By default `Rails.cache` is used as cache store.

### Examples

#### Declaration

```
1 class AverageHotelDailyRevenue < Patterns::Calculation
2   set_cache_expiry_every 1.day
3
4   private
5
6   def result
7     reservations.sum(:price) / days_in_year
8   end
9
10  def reservations
11    Reservation.where(
12      date: (beginning_of_year..end_of_year),
13      hotel_id: subject.id
14    )
15  end
16
17  def days_in_year
18    end_of_year.yday
```



---

```
19   end
20
21   def year
22     options.fetch(:year, Date.current.year)
23   end
24
25   def beginning_of_year
26     Date.new(year).beginning_of_year
27   end
28
29   def end_of_year
30     Date.new(year).end_of_year
31   end
32 end
```

#### Usage

```
1 hotel = Hotel.find(123)
2 AverageHotelDailyRevenue.result_for(hotel)
3 AverageHotelDailyRevenue.result_for(hotel, year: 2015)
4
5 TotalCurrentRevenue.calculate
6 AverageDailyRevenue.result
```

## Rule and Ruleset

### When to use it

Rule objects provide a place for dislocating/extracting conditional logic.

Use it when: - given complex condition is duplicated in multiple places in your codebase - part of condition logic can be reused in some other place - there is a need to instantiate condition itself for some reason (i.e. to represent it in the interface) - responsibility of your class is blurred by complex conditional logic, and as a result... - ...tests for your class require multiple condition branches / nested contexts

### Assumptions and rules

- Rule has `#satisfied?`, `#applicable?`, `#not_applicable?` and `#forceable?` methods available.
- Rule has to implement at least `#satisfied?` method. `#not_applicable?` and `#forceable?` are meant to be overridable.
- `#forceable?` makes sense in scenario where condition is capable of being force-satisfied regardless if its actually satisfied or not. Is `true` by default.

- 
- Override `#not_applicable?` when method is applicable only under some specific conditions. Is `false` by default.
  - Rule requires a subject as first argument.
  - Multiple rules and rulesets can be combined into new ruleset as both share same interface and can be used interchangeably (composite pattern).
  - By default empty ruleset is satisfied.

**Forcing rules** On some occasions there is a situation in which some condition should be overridable. Let's say we may want send shipping notification even though given order was not paid for and under regular circumstances such notification should not be sent. In this case, while regular logic with some automated process would not trigger delivery, an action triggered by user from UI could do it, by passing `force: true` option to `#satisfied?` methods.

It might be good idea to test for `#forceable?` on the UI level to control visibility of such link/button.

Overriding `#forceable` can be useful to prevent some edge cases, i.e. `ContactInformationProvidedRule` might check if customer for given order has provided any contact means by which a notification could be delivered. If not, ruleset containing such rule (and the rule itself) would not be “forceable” and UI could reflect that by querying `#forceable?`.

**Regular and strong rulesets** While regular `Ruleset` can be satisfied or forced if any of its rules in not applicable, the `StrongRuleset` is not satisfied and not “forceable” if any of its rules is not applicable.

**#not\_applicable? vs #applicable?** It might be surprising that is is the negated version of the `#applicable?` predicate methods that is overridable. However, from the actual usage perspective, it usually easier to conceptually define when condition makes no sense than other way around.

## Examples

### Declaration

```
1 class OrderIsSentRule < Patterns::Rule
2   def satisfied?
3     subject.sent?
4   end
5 end
6
7 class OrderIsPaidRule < Patterns::Rule
8   def satisfied?
9     subject.paid?
```

---

```
10   end
11
12   def forceable?
13     true
14   end
15 end
16
17 OrderCompletedNotificationRuleset = Class.new(Patterns::Ruleset)
18 OrderCompletedNotificationRuleset.
19   add_rule(:order_is_sent_rule).
20   add_rule(:order_is_paid_rule)
```

#### Usage

```
1 OrderIsPaidRule.new(order).satisfied?
2 OrderCompletedNotificationRuleset.new(order).satisfied?
3
4 ResendOrderNotification.call(order) if
  OrderCompletedNotificationRuleset.new(order).satisfied?(force: true)
```

### Further reading

- 7 ways to decompose fat active record models

### About Selleo



Software development teams with an entrepreneurial sense of ownership at their core delivering great digital products and building culture people want to belong to. We are a community of engaged co-workers passionate about crafting impactful web solutions which transform the way our clients do business.

All names and logos for Selleo are trademark of Selleo Labs Sp. z o.o. (formerly Selleo Sp. z o.o. Sp.k.)