Bonnes pratiques

Ruby, Rails & misc

Éditeur

- Configuration de son éditeur
  - espaces plutôt que tabs
  - niveau d’indentation par défaut
  - New line @EOF
  - Strip whitespaces @EOL

Global

- Rangement par ordre logique, alphabétique
- Aligner au maximum ce qui peut l'être
    - valable aussi pour une suite d'assignations de variables

Versioning

- Si n° de ticket le préciser en référence du commit (Refs #XXXX)
- En cas de migration, le préciser dans le commit et le changelog avec BUMP version mineure semver x.Y.z
- Git flow et pull requests pour finaliser une feature/ branch

Méthodes

- Pas plus de 3 ou 4 paramètres
- Un maximum de 10 lignes, hors assignations et espaces
- 120 caractères par ligne au maximum (80 c’est juste)

Ruby

Layout

- On <3 les espaces, vraiment :

```
sum = 1 + 2
[1, 2, 3].each { |e| puts { conditions: { active: true }}
```

- Séparer le contenu d'une méthode en plusieurs paragraphes logiques.

- Charger les libs par ordre alphabétique si possible

- Eviter les lignes trop longues :

```
def send_mail(source)
  Mailer.deliver(to: 'bob@example.com', from:
'us@example.com', subject: 'Important message', body:
source.text)
end
```

- devient

```
def send_mail(source)
  Mailer.deliver(
    to:      'bob@example.com',
    from:    'us@example.com',
    subject: 'Important message',
    body:    source.text)
end
```

Syntax

- Suivre la bonne pratique de formatage

```
case
when song.name == 'Misty'
  puts 'Not again!'
else
  song.play
end
```

- Pas trop de ternaire

```
# bad
some_condition ? (nested_condition ? nested_something :
nested_something_else) : something_else
```

```
# good
if some_condition
  nested_condition ? nested_something : nested_something_else
else
  something_else
end
```

- Utiliser `&&` `||` en lieu et place de `and` `or`

- Ne pas utiliser la deuxième pratique

```ruby
# good
do_something if some_condition

# another good option (according to Ruby guide)
some_condition and do_something
```

- Pas d’autres méthodes après un do...end

```ruby
# bad
names.select do |name|
  name.start_with?("S")
end.map { |name| name.upcase }
```

- Parenthèses lors d’une assignation

```ruby
# ok
if (x = self.next_value)
  # body omitted
end
```

- Pas (plus?) de return

```ruby
# bad
def some_method(some_arr)
  return some_arr.size
end

# good
def some_method(some_arr)
  some_arr.size
end
```

- New hash syntax pour les hash dont toutes les clefs sont des symboles

```
hash = { one: 1, two: 2 }
```

- Pas trop d’espaces non plus

```
# bad
def some_method(arg1=:default, arg2=nil, arg3=[])
  # do something...
end
```

- Nouvelle syntaxe lambda (ça pique)

```ruby
# old
lambda = lambda { |a, b| a + b }
lambda.call(1, 2)

# new
lambda = ->(a, b) { a + b }
lambda.(1, 2)
```

- Nommer _ les paramètres inutilisés

```ruby
result = hash.map { |_, v| v + 1 }
```

Strings

- Utilisation de ” uniquement en cas d’interpolation, sinon ‘
- On utilise la bad syntax de l’interpolation

```ruby
# bad
def to_s
  "#{@first_name} #{@last_name}"
end

# good
def to_s
  "#@first_name #@last_name"
end
```

Regular Expressions

- Utiliser des variables dans les expressions régulières

```
# bad
/(regexp)/ =~ string
...
process $1

# good
/(?<meaningful_var>regexp)/ =~ string
...
process meaningful_var
```

Naming

- Le nom des méthodes prédicats doivent être suffixés par un point d'interrogation

`Array#empty?`

- Ne pas préfixer ces méthodes ainsi que les champs booléens d'une table par is_

```ruby
# good
def available?
  # do something...
end
```

- Par quoi suffixer dans le cas d'un Time ?

- Suffixe _on pour les Date et _at pour les DateTime

```ruby
def change
  create_table :awesome_table do |t|
    t.date     :opening_on
    t.datetime :shipped_at
  end
end
```

- Que doit représenter *method!* ?
  - une méthode qui raise ?
  - une méthode qui modifie self ?
    - une méthode qui modifie les paramètres ?
- Ne pas utiliser de !

- Toujours utiliser des noms de variables clairs, même pour des `reduce`, `map`

- Utiliser la méthode `count` uniquement dans le cas de scopes/relations ActiveRecord.

- Toujours préférer les équivalents

```
- collect  -> map
- detect   -> find
- find_all -> select
- inject   -> reduce
- length   -> size
```

Annotations

```ruby
def bar
  # FIXME: This has crashed occasionally since v3.2.1.
  # It may be related to the BarBazUtil upgrade.
  baz(:quux)
end
```

- Etre généreux :

```
- TODO
- FIXME
- OPTIMIZE
- HACK
- REVIEW
```

Classes

- Eviter d'avoir des classes trop longues :
  - répartir les méthodes en modules logiques qui seront inclus.
    - Préférer l'utilisation de modules plutôt que l'héritage.
- Be SOLID

- Toujours définir un to_s qui utilise une méthode plus explicite (ex: fullname)

- Économes sur le duck-typing

- Limiter l'héritage aux Models et préférer l'utilisation de modules (dans la mesure du possible :)

- Indenter le niveau de visibilité à la même hauteur que la définition de la classe

```
module SomeModule
  class SomeClass
    def public_method
      # ...
    end

  private
    def private_method
      # ...
    end
  end
end
```

- Préférer un module à une classe qui n'implémente que des méthodes de classe (statique pour les fans de jarjar)

```ruby
# bad
class SomeClass
  def self.some_method
    # ...
  end
end

# good
module SomeModule
  def self.some_method
    # ...
  end
end
```

- Définition des namespaces d'un module

```ruby
# bad
module SomeModule::SecondModule
  def some_method
    # ...
  end
end
```

```ruby
# good
module SomeModule
  module SecondModule
    def some_method
      # ...
    end
  end
end
```

- Définition d'un module qui sera inclus dans une classe (plus besoin d'étendre)

```
module SomeModule
  module ClassMethods
    def some_class_method
      # ...
    end
  end

  module InstanceMethods
    def some_instance_method
      # ...
    end
  end

  def self.included(receiver)
    receiver.extend         ClassMethods
    receiver.send :include, InstanceMethods
  end
end
```

Exceptions

- Utiliser des exceptions pour lever de **vraies** erreurs et **seulement** dans les **libs**
- Rester fidèle aux Exceptions Ruby en dehors des cas spécifiquement métier
- Ne pas gérer les erreurs dans le fil d'exécution de l'application

Ruby on Rails

Controllers

- Garder des contrôlleurs légers qui ne contiennent pas de logique métier

- Une action appelle une seule méthode de logique métier

- Pas plus de 2 variables d'instance (on n'en a même plus, mais on triche avec decent_exposure)

- Ordre de définition dans un contrôlleur

```
respond_to
before_filter
DSL (ex: decent_exposure)
```

- Flash messages, utiliser les mots clefs suivants

`:notice` pour les messages de succès (convention rails)
`:alert` pour les messages d'erreur (convention rails)
`:warning` pour les messages d'avertissement
`:info` pour les messages d'information

Models

- Nous n’utilisons pas ActiveAttr
    - Opaque au niveau des include
    - Autant le faire nous même

- Lorsqu’on cherche une instance avec un seul critère de recherche, préférer

`User.find_by_x` plutôt que `User.where(x: X).first`

- Callbacks dans les Observers
- Ordre de définition...

- extend
- include
- appel au DSL d'un plugin (ex: carrierwave)
- attr_reader
- attr_writter
- attr_accessible
- delegate
- associations :
  - belongs_to
  - has_one
  - has_many
  - habtm
- accepts_nested_attributes_for
- validations
- named_scopes
- named scopes transformés en méthodes de classe
- méthodes de classe
- méthodes getter (valeur par défaut)
- méthodes d'instance
- private
  - méthodes de validation
  - méthodes privées

Nous sommes preneurs d'avis sur delegate/associations

- Utiliser `has_many :xx, through: :yy` uniquement dans le cas où la relation porte des informations

- Utiliser le nouveau système de validations

```
validates :email, presence:   true,
                  uniqueness: true
```

Views

- Utiliser la logique de Presenter
    - Classe
    - Réunissant tous les helpers permettant d’afficher un Objet

    - Et concrètement

- Préférer judge à client_side_validations si vous êtes, comme nous, plutôt féniants des appels AJAX

- Existe aussi en version simple_form avec judge_simple_form

- Nommer `form` le FormBuilder

    - Nommer `<nested>_form` dans le cas d'un nested form

- Ne pas utiliser de répertoire shared pour les partials (complètement arbitraire :)
    - Utiliser le répertoire application ou bien le répertoire de la ressource concernée
- Utiliser des parenthèses pour les locales (hysteria)

```
t('awesome_translation')
```

Tests

- On utilise RSpec car c'est pour les enfants
- On ne teste pas les Controllers avec RSpec mais avec notre légume favori
- On ne teste pas les helpers. Un helper doit rester dans le domaine du cosmétique
    - Sinon on crie puis on créé un Presenter avec sa logique métier

- On utilise donc Cucumber pour nos scénarios
- On teste les Mailers (Cécile ?)

- Utiliser #method et .method

```
describe Article
  describe '#summary'
    #...
  end

  describe '.latest'
    #...
  end
end
```

Assets

- Regrouper les fichiers css/javascript par fonctionnalité (ex: confirmable.css, filters.css)
- Créer un fichier par contrôlleur/partial si nécessaire (ex: header.css, products.css)
- Ne pas faire de `require_tree`

Questions ?

- O(^%*&^% Bundler & Gemfile.lock par env
- On a générateur qui initialise des settings nécessaires à un initializer, le problème c'est que Rails lance les initializers (qui require nos settings en question) avant.

Sources

- https://github.com/bbatsov/ruby-style-guide
- https://github.com/bbatsov/rails-style-guide
- gems
    - decent_exposure
    - judge
    - judge_simple_form