Prelude

> Role models are important.
> -- Officer Alex J. Murphy / RoboCop

The goal of this guide is to present a set of best practices and style prescriptions for Ruby on Rails 4 development. It's a complementary guide to the already existing community-driven Ruby coding style guide.

Some of the advice here is applicable only to Rails 4.0+.

You can generate a PDF or an HTML copy of this guide using Transmuter.

Translations of the guide are available in the following languages:

- Chinese Simplified
- Chinese Traditional
- German
- Japanese
- Russian
- Turkish
- Korean

The Rails Style Guide

This Rails style guide recommends best practices so that real-world Rails programmers can write code that can be maintained by other real-world Rails programmers. A style guide that reflects real-world usage gets used, and a style guide that holds to an ideal that has been rejected by the people it is supposed to help risks not getting used at all – no matter how good it is.

The guide is separated into several sections of related rules. I've tried to add the rationale behind the rules (if it's omitted I've assumed it's pretty obvious).

I didn't come up with all the rules out of nowhere - they are mostly based on my extensive career as a professional software engineer, feedback and suggestions from members of the Rails community and various highly regarded Rails programming resources.

Table of Contents

Configuration

- Put custom initialization code in `config/initializers`. The code in initializers executes on application startup. [link]
- Keep initialization code for each gem in a separate file with the same name as the gem, for example `carrierwave.rb`, `active_admin.rb`, etc. [link]
- Adjust accordingly the settings for development, test and production environment (in the corresponding files under `config/environments/`) [link]
    - Mark additional assets for precompilation (if any):

-
  ```
  # config/environments/production.rb
  # Precompile additional assets (application.js, application.css,
  #and all non-JS/CSS are already added)
  config.assets.precompile += %w( rails_admin/rails_admin.css
  rails_admin/rails_admin.js )
  ```
- Keep configuration that's applicable to all environments in the `config/application.rb` file. [link]
- Create an additional `staging` environment that closely resembles the `production` one. [link]
- Keep any additional configuration in YAML files under the `config/` directory. [link] Since Rails 4.2 YAML configuration files can be easily loaded with the new `config_for` method:
  ```
  Rails::Application.config_for(:yaml_file)
  ```

Routing

- When you need to add more actions to a RESTful resource (do you really need them at all?) use `member` and `collection` routes. [link]
  ```
  # bad
  get 'subscriptions/:id/unsubscribe'
  resources :subscriptions

  # good
  resources :subscriptions do
  get 'unsubscribe', on: :member
  end
  ```

```
# bad
get 'photos/search'
resources :photos

# good
resources :photos do
get 'search', on: :collection
end
```

- If you need to define multiple `member/collection` routes use the alternative block syntax. [link]

```
resources :subscriptions do
member do
  get 'unsubscribe'
  # more routes
end
end

resources :photos do
collection do
  get 'search'
  # more routes
end
end
```

- Use nested routes to express better the relationship between ActiveRecord models. [link] `class`

```
Post < ActiveRecord::Base
has_many :comments
end

class Comments < ActiveRecord::Base
belongs_to :post
end

# routes.rb
resources :posts do
resources :comments
end
```

- If you need to nest routes more than 1 level deep then use the `shallow: true` option. This will save user from long urls `posts/1/comments/5/versions/7/edit` and you from long url helpers `edit_post_comment_version`. `resources :posts, shallow: true do`

```
resources :comments do
  resources :versions
end
end
```

- Use namespaced routes to group related actions. [link] `namespace :admin do`

```
# Directs /admin/products/* to Admin::ProductsController
# (app/controllers/admin/products_controller.rb)
resources :products
end
```

- Never use the legacy wild controller route. This route will make all actions in every controller accessible via GET requests. [link] `# very bad`

```
match ':controller(/:action(/:id(.:format)))'
```

- Don't use `match` to define any routes unless there is need to map multiple request types among `[:get, :post, :patch, :put, :delete]` to a single action using `:via` option. [link]

Controllers

- Keep the controllers skinny - they should only retrieve data for the view layer and shouldn't contain any business logic (all the business logic should naturally reside in the model). [link]
- Each controller action should (ideally) invoke only one method other than an initial find or new. [link]
- Share no more than two instance variables between a controller and a view. [link]

Models

- Introduce non-ActiveRecord model classes freely. [link]
- Name the models with meaningful (but short) names without abbreviations. [link]
- If you need model objects that support ActiveRecord behavior (like validation) without the ActiveRecord database functionality use the ActiveAttr gem. [link] `class Message`

```
include ActiveAttr::Model

attribute :name
attribute :email
attribute :content
attribute :priority

attr_accessible :name, :email, :content

validates :name, presence: true
validates :email, format: { with: /\A[-a-z0-9_+\.]+\@([-a-z0-9]+\.)+[a-z0-9]{2,4}\z/i
}
validates :content, length: { maximum: 500 }
end
```

For a more complete example refer to the RailsCast on the subject.

ActiveRecord

- Avoid altering ActiveRecord defaults (table names, primary key, etc) unless you have a very good reason (like a database that's not under your control). [link] `# bad - don't do this if you can modify the schema`

```
class Transaction < ActiveRecord::Base
self.table_name = 'order'
...
end
```

- Group macro-style methods (`has_many`, `validates`, etc) in the beginning of the class definition. [link]

```
class User < ActiveRecord::Base
# keep the default scope first (if any)
default_scope { where(active: true) }

# constants come up next
COLORS = %w(red green blue)

# afterwards we put attr related macros
attr_accessor :formatted_date_of_birth

attr_accessible :login, :first_name, :last_name, :email, :password

# followed by association macros
belongs_to :country

has_many :authentications, dependent: :destroy

# and validation macros
validates :email, presence: true
validates :username, presence: true
validates :username, uniqueness: { case_sensitive: false }
validates :username, format: { with: /\A[A-Za-z][A-Za-z0-9._-]{2,19}\z/ }
validates :password, format: { with: /\A\S{8,128}\z/, allow_nil: true}

# next we have callbacks
before_save :cook
before_save :update_username_lower

# other macros (like devise's) should be placed after the callbacks

...
end
```

- Prefer `has_many :through` to `has_and_belongs_to_many`. Using `has_many :through` allows additional attributes and validations on the join model. [link] `# not so good - using`

```
has_and_belongs_to_many
class User < ActiveRecord::Base
has_and_belongs_to_many :groups
end

class Group < ActiveRecord::Base
has_and_belongs_to_many :users
end

# prefered way - using has_many :through
class User < ActiveRecord::Base
has_many :memberships
has_many :groups, through: :memberships
end

class Membership < ActiveRecord::Base
belongs_to :user
belongs_to :group
end

class Group < ActiveRecord::Base
has_many :memberships
has_many :users, through: :memberships
end
```

- Prefer `self[:attribute]` over `read_attribute(:attribute)`. [link] `# bad`

```
def amount
read_attribute(:amount) * 100
end

# good
def amount
self[:amount] * 100
end
```

- Prefer `self[:attribute] = value` over `write_attribute(:attribute, value)`. [link] `# bad`

```
def amount
write_attribute(:amount, 100)
end

# good
def amount
self[:amount] = 100
end
```

- Always use the new "sexy" validations. [link] `# bad`

```
validates_presence_of :email
validates_length_of :email, maximum: 100

# good
validates :email, presence: true, length: { maximum: 100 }
```

- When a custom validation is used more than once or the validation is some regular expression mapping, create a custom validator file. [link] `# bad`

```
class Person
validates :email, format: { with: /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i }
end

# good
class EmailValidator < ActiveModel::EachValidator
def validate_each(record, attribute, value)
  record.errors[attribute] << (options[:message] || 'is not a valid email') unless
value =~ /\A([^@\s]+)@((?:[-a-z0-9]+\.)+[a-z]{2,})\z/i
end
end

class Person
validates :email, email: true
end
```

- Keep custom validators under `app/validators`. [link]
- Consider extracting custom validators to a shared gem if you're maintaining several related apps or the validators are generic enough. [link]
- Use named scopes freely. [link] `class User < ActiveRecord::Base`

```
scope :active, -> { where(active: true) }
scope :inactive, -> { where(active: false) }

scope :with_orders, -> { joins(:orders).select('distinct(users.id)') }
end
```

- When a named scope defined with a lambda and parameters becomes too complicated, it is preferable to make a class method instead which serves the same purpose of the named scope and returns an `ActiveRecord::Relation` object. Arguably you can define even simpler scopes like this.

[link]

```
Ruby class User < ActiveRecord::Base def self.with_orders
joins(:orders).select('distinct(users.id)') end end
```

- Beware of the behavior of the `update_attribute` method. It doesn't run the model validations (unlike `update_attributes`) and could easily corrupt the model state. [link]
- Use user-friendly URLs. Show some descriptive attribute of the model in the URL rather than its `id`. There is more than one way to achieve this: [link]
  - Override the `to_param` method of the model. This method is used by Rails for constructing a URL to the object. The default implementation returns the `id` of the record as a String. It could be overridden to include another human-readable attribute. `class Person`

    ```
    def to_param
      "#{id} #{name}".parameterize
    end
    end
    ```
- In order to convert this to a URL-friendly value, `parameterize` should be called on the string. The `id` of the object needs to be at the beginning so that it can be found by the `find` method of ActiveRecord.
  - Use the `friendly_id` gem. It allows creation of human-readable URLs by using some descriptive attribute of the model instead of its `id`. `class Person`

    ```
    extend FriendlyId
    friendly_id :name, use: :slugged
    end
    ```

  Check the gem documentation for more information about its usage.
- Use `find_each` to iterate over a collection of AR objects. Looping through a collection of records from the database (using the `all` method, for example) is very inefficient since it will try to instantiate all the objects at once. In that case, batch processing methods allow you to work with the records in batches, thereby greatly reducing memory consumption. [link] `# bad`

  ```
  Person.all.each do |person|
  person.do_awesome_stuff
  end

  Person.where('age > 21').each do |person|
  person.party_all_night!
  end

  # good
  Person.find_each do |person|
  person.do_awesome_stuff
  end

  Person.where('age > 21').find_each do |person|
  person.party_all_night!
  end
  ```
- Since Rails creates callbacks for dependent associations, always call `before_destroy` callbacks that perform validation with `prepend: true`. [link] `# bad (roles will be deleted automatically even if super_admin? is true)`

```
has_many :roles, dependent: :destroy

before_destroy :ensure_deletable

def ensure_deletable
fail "Cannot delete super admin." if super_admin?
end

# good
has_many :roles, dependent: :destroy

before_destroy :ensure_deletable, prepend: true

def ensure_deletable
fail "Cannot delete super admin." if super_admin?
end
```

ActiveRecord Queries

- Avoid string interpolation in queries, as it will make your code susceptible to SQL injection attacks. [link]

```
# bad - param will be interpolated unescaped
Client.where("orders_count = #{params[:orders]}")

# good - param will be properly escaped
Client.where('orders_count = ?', params[:orders])
```

- Consider using named placeholders instead of positional placeholders when you have more than 1 placeholder in your query. [link] `# okish`

```
Client.where(
'created_at >= ? AND created_at <= ?',
params[:start_date], params[:end_date]
)

# good
Client.where(
'created_at >= :start_date AND created_at <= :end_date',
start_date: params[:start_date], end_date: params[:end_date]
)
```

- Favor the use of `find` over `where` when you need to retrieve a single record by id. [link] `# bad`

```
User.where(id: id).take

# good
User.find(id)
```

- Favor the use of `find_by` over `where` when you need to retrieve a single record by some attributes. [link]

```
# bad
User.where(first_name: 'Bruce', last_name: 'Wayne').first

# good
User.find_by(first_name: 'Bruce', last_name: 'Wayne')
```

- Use `find_each` when you need to process a lot of records. [link] `# bad - loads all the records at once`

```
# This is very inefficient when the users table has thousands of rows.
User.all.each do |user|
NewsMailer.weekly(user).deliver_now
end

# good - records are retrieved in batches
User.find_each do |user|
NewsMailer.weekly(user).deliver_now
end
```

- Favor the use of `where.not` over SQL. [link] `# bad`

```
User.where("id != ?", id)

# good
User.where.not(id: id)
```

- When specifying an explicit query in a method such as `find_by_sql`, use heredocs with `squish`. This allows you to legibly format the SQL with line breaks and indentations, while supporting syntax highlighting in many tools (including GitHub, Atom, and RubyMine). [link]

```
User.find_by_sql(<<SQL.squish)
SELECT
  users.id, accounts.plan
FROM
  users
INNER JOIN
  accounts
ON
  accounts.user_id = users.id
# further complexities...
SQL
```

`String#squish` removes the indentation and newline characters so that your server log shows a fluid string of SQL rather than something like this: `SELECT\n    users.id, accounts.plan\n  FROM\n    users\n  INNER JOIN\n    acounts\n  ON\n    accounts.user_id = users.id`

Migrations

- Keep the `schema.rb` (or `structure.sql`) under version control. [link]
- Use `rake db:schema:load` instead of `rake db:migrate` to initialize an empty database. [link]
- Enforce default values in the migrations themselves instead of in the application layer. [link]
```
# bad - application enforced default value
def amount
self[:amount] or 0
end
```
While enforcing table defaults only in Rails is suggested by many Rails developers, it's an extremely brittle approach that leaves your data vulnerable to many application bugs. And you'll have to consider the fact that most non-trivial apps share a database with other applications, so imposing data integrity from the Rails app is impossible.
- Enforce foreign-key constraints. As of Rails 4.2, ActiveRecord supports foreign key constraints natively. [link]
- When writing constructive migrations (adding tables or columns), use the `change` method instead of `up` and `down` methods. [link]
```
# the old way
class AddNameToPeople < ActiveRecord::Migration
def up
  add_column :people, :name, :string
end

def down
  remove_column :people, :name
end
end

# the new prefered way
class AddNameToPeople < ActiveRecord::Migration
def change
  add_column :people, :name, :string
end
end
```
- Don't use model classes in migrations. The model classes are constantly evolving and at some point in the future migrations that used to work might stop, because of changes in the models used. [link]

Views

- Never call the model layer directly from a view. [link]
- Never make complex formatting in the views, export the formatting to a method in the view helper or the model. [link]

- Mitigate code duplication by using partial templates and layouts. [link]

Internationalization

- No strings or other locale specific settings should be used in the views, models and controllers. These texts should be moved to the locale files in the `config/locales` directory. [link]
- When the labels of an ActiveRecord model need to be translated, use the `activerecord` scope: [link]

```
en:
activerecord:
  models:
    user: Member
  attributes:
    user:
      name: 'Full name'
```

  Then `User.model_name.human` will return "Member" and `User.human_attribute_name("name")` will return "Full name". These translations of the attributes will be used as labels in the views.
- Separate the texts used in the views from translations of ActiveRecord attributes. Place the locale files for the models in a folder `locales/models` and the texts used in the views in folder `locales/views`. [link]
  - When organization of the locale files is done with additional directories, these directories must be described in the `application.rb` file in order to be loaded.

```
# config/application.rb
config.i18n.load_path += Dir[Rails.root.join('config', 'locales', '**', '*.{rb,yml}')]
```

- Place the shared localization options, such as date or currency formats, in files under the root of the `locales` directory. [link]
- Use the short form of the I18n methods: `I18n.t` instead of `I18n.translate` and `I18n.l` instead of `I18n.localize`. [link]
- Use "lazy" lookup for the texts used in views. Let's say we have the following structure: [link]

```
en:
users:
  show:
    title: 'User details page'
```

  The value for `users.show.title` can be looked up in the template `app/views/users/show.html.haml` like this: `= t '.title'`
- Use the dot-separated keys in the controllers and models instead of specifying the `:scope` option. The dot-separated call is easier to read and trace the hierarchy. [link]

```
# bad
I18n.t :record_invalid, :scope => [:activerecord, :errors, :messages]

# good
I18n.t 'activerecord.errors.messages.record_invalid'
```

- More detailed information about the Rails I18n can be found in the Rails Guides [link]

Assets

Use the assets pipeline to leverage organization within your application.

- Reserve `app/assets` for custom stylesheets, javascripts, or images. [link]
- Use `lib/assets` for your own libraries that don’t really fit into the scope of the application. [link]
- Third party code such as jQuery or bootstrap should be placed in `vendor/assets`. [link]
- When possible, use gemified versions of assets (e.g. jquery-rails, jquery-ui-rails, bootstrap-sass, zurb-foundation). [link]

Mailers

- Name the mailers `SomethingMailer`. Without the Mailer suffix it isn't immediately apparent what's a mailer and which views are related to the mailer. [link]
- Provide both HTML and plain-text view templates. [link]
- Enable errors raised on failed mail delivery in your development environment. The errors are disabled by default. [link] `# config/environments/development.rb`

```
config.action_mailer.raise_delivery_errors = true
```

- Use a local SMTP server like Mailcatcher in the development environment. [link] `# config/environments/development.rb`

```
config.action_mailer.smtp_settings = {
address: 'localhost',
port: 1025,
# more settings
}
```

- Provide default settings for the host name. [link] `# config/environments/development.rb`

```
config.action_mailer.default_url_options = { host: "#{local_ip}:3000" }

# config/environments/production.rb
config.action_mailer.default_url_options = { host: 'your_site.com' }

# in your mailer class
default_url_options[:host] = 'your_site.com'
```

- If you need to use a link to your site in an email, always use the `_url`, not `_path` methods. The `_url` methods include the host name and the `_path` methods don't. [link] `# bad`

```
You can always find more info about this course
<%= link_to 'here', course_path(@course) %>

# good
```

```
You can always find more info about this course
<%= link_to 'here', course_url(@course) %>
```

- Format the from and to addresses properly. Use the following format: [link]
```
# in your mailer class
default from: 'Your Name <info@your_site.com>'
```
- Make sure that the e-mail delivery method for your test environment is set to `test`: [link]
```
# config/environments/test.rb

config.action_mailer.delivery_method = :test
```
- The delivery method for development and production should be `smtp`: [link]
```
# config/environments/development.rb, config/environments/production.rb

config.action_mailer.delivery_method = :smtp
```
- When sending html emails all styles should be inline, as some mail clients have problems with external styles. This however makes them harder to maintain and leads to code duplication. There are two similar gems that transform the styles and put them in the corresponding html tags: premailer-rails and roadie. [link]
- Sending emails while generating page response should be avoided. It causes delays in loading of the page and request can timeout if multiple email are sent. To overcome this emails can be sent in background process with the help of sidekiq gem. [link]

Time

- Config your timezone accordingly in `application.rb`. [link]
```
config.time_zone = 'Eastern European Time'
# optional - note it can be only :utc or :local (default is :utc)
config.active_record.default_timezone = :local
```
- Don't use `Time.parse`. [link]
```
# bad
Time.parse('2015-03-02 19:05:37') # => Will assume time string given is in the system's time zone.

# good
Time.zone.parse('2015-03-02 19:05:37') # => Mon, 02 Mar 2015 19:05:37 EET +02:00
```
- Don't use `Time.now`. [link]
```
# bad
Time.now # => Returns system time and ignores your configured time zone.

# good
Time.zone.now # => Fri, 12 Mar 2014 22:04:47 EET +02:00
Time.current # Same thing but shorter.
```

Bundler

- Put gems used only for development or testing in the appropriate group in the Gemfile. [link]
- Use only established gems in your projects. If you're contemplating on including some little-known gem you should do a careful review of its source code first. [link]
- OS-specific gems will by default result in a constantly changing `Gemfile.lock` for projects with multiple developers using different operating systems. Add all OS X specific gems to a `darwin` group in the Gemfile, and all Linux specific gems to a `linux` group: [link]
  ```
  # Gemfile
  group :darwin do
  gem 'rb-fsevent'
  gem 'growl'
  end

  group :linux do
  gem 'rb-inotify'
  end
  ```
  To require the appropriate gems in the right environment, add the following to `config/application.rb`:
  ```
  platform = RUBY_PLATFORM.match(/(linux|darwin)/)[0].to_sym
  Bundler.require(platform)
  ```
- Do not remove the `Gemfile.lock` from version control. This is not some randomly generated file - it makes sure that all of your team members get the same gem versions when they do a `bundle install`. [link]

Managing processes

- If your projects depends on various external processes use foreman to manage them. [link]

Further Reading

There are a few excellent resources on Rails style, that you should consider if you have time to spare:

- The Rails 4 Way
- Ruby on Rails Guides
- The RSpec Book
- The Cucumber Book
- Everyday Rails Testing with RSpec
- Better Specs for RSpec

Contributing

Nothing written in this guide is set in stone. It's my desire to work together with everyone interested in Rails coding style, so that we could ultimately create a resource that will be beneficial to the entire Ruby community.

Feel free to open tickets or send pull requests with improvements. Thanks in advance for your help!

You can also support the project (and RuboCop) with financial contributions via gittip.

Support via Gittip

How to Contribute?

It's easy, just follow the contribution guidelines.

License

Spread the Word

A community-driven style guide is of little use to a community that doesn't know about its existence. Tweet about the guide, share it with your friends and colleagues. Every comment, suggestion or opinion we get makes the guide just a little bit better. And we want to have the best possible guide, don't we?

Cheers,
Bozhidar