API Documentation Guidelines

This guide documents the Ruby on Rails API documentation guidelines.

After reading this guide, you will know:

- How to write effective prose for documentation purposes.
- Style guidelines for documenting different kinds of Ruby code.

RDoc

The Rails API documentation is generated with RDoc. To generate it, make sure you are in the rails root directory, run `bundle install` and execute:

```
$ bundle exec rake rdoc
```

Resulting HTML files can be found in the ./doc/rdoc directory.

Please consult the RDoc documentation for help with the markup, and also take into account these additional directives.

Links

Rails API documentation are not meant to be viewed on GitHub and therefore links should use the RDoc `link` markup relative to the current API.

This is due to differences between GitHub Markdown and the generated RDoc that is published at api.rubyonrails.org and edgeapi.rubyonrails.org.

For example, we use `[link:classes/ActiveRecord/Base.html]` to create a link to the `ActiveRecord::Base` class generated by RDoc.

This is preferred over absolute URLs such as `[https://api.rubyonrails.org/classes/ActiveRecord/Base.h` which would take the reader outside their current documentation version (e.g. edgeapi.rubyonrails.org).

Wording

Write simple, declarative sentences. Brevity is a plus: get to the point.

Write in present tense: "Returns a hash that…", rather than "Returned a hash that…" or "Will return a hash that…".

Start comments in upper case. Follow regular punctuation rules:

```ruby
# Declares an attribute reader backed by an internally-named
# instance variable.
def attr_internal_reader(*attrs)
  # ...
end
```

Communicate to the reader the current way of doing things, both explicitly and implicitly. Use the idioms recommended in edge. Reorder sections to emphasize favored approaches if needed, etc. The documentation should be a model for best practices and canonical, modern Rails usage.

Documentation has to be concise but comprehensive. Explore and document edge cases. What happens if a module is anonymous? What if a collection is empty? What if an argument is nil?

The proper names of Rails components have a space in between the words, like "Active Support". `ActiveRecord` is a Ruby module, whereas Active Record is an ORM. All Rails documentation should consistently refer to Rails components by their proper name, and if in your next blog post or presentation you remember this tidbit and take it into account that'd be phenomenal.

Spell names correctly: Arel, minitest, RSpec, HTML, MySQL, JavaScript, ERB. When in doubt, please have a look at some authoritative source like their official documentation.

Use the article "an" for "SQL", as in "an SQL statement". Also "an SQLite database".

Prefer wordings that avoid "you"s and "your"s. For example, instead of

```
If you need to use `return` statements in your callbacks, it is recommended that you explic
```

use this style:

```
If `return` is needed it is recommended to explicitly define a method.
```

That said, when using pronouns in reference to a hypothetical person, such as "a user with a session cookie", gender neutral pronouns (they/their/them) should be used. Instead of:

- he or she... use they.
- him or her... use them.
- his or her... use their.
- his or hers... use theirs.
- himself or herself... use themselves.

English

Please use American English (*color*, *center*, *modularize*, etc). See a list of American and British English spelling differences here.

Oxford Comma

Please use the Oxford comma (“red, white, and blue”, instead of “red, white and blue”).

Example Code

Choose meaningful examples that depict and cover the basics as well as interesting points or gotchas.

Use two spaces to indent chunks of code–that is, for markup purposes, two spaces with respect to the left margin. The examples themselves should use Rails coding conventions.

Short docs do not need an explicit “Examples” label to introduce snippets; they just follow paragraphs:

```
# Converts a collection of elements into a formatted string by
# calling +to_s+ on all elements and joining them.
#
#   Blog.all.to_fs # => "First PostSecond PostThird Post"
```

On the other hand, big chunks of structured documentation may have a separate “Examples” section:

```
# ==== Examples
#
#   Person.exists?(5)
#   Person.exists?('5')
#   Person.exists?(name: "David")
#   Person.exists?(['name LIKE ?', "%#{query}%"])
```

The results of expressions follow them and are introduced by “# =>”, vertically aligned:

```
# For checking if an integer is even or odd.
#
#   1.even? # => false
#   1.odd?  # => true
#   2.even? # => true
#   2.odd?  # => false
```

If a line is too long, the comment may be placed on the next line:

```
#   label(:article, :title)
#   # => <label for="article_title">Title</label>
#
#   label(:article, :title, "A short title")
#   # => <label for="article_title">A short title</label>
#
```

```
#   label(:article, :title, "A short title", class: "title_label")
#   # => <label for="article_title" class="title_label">A short title</label>
```

Avoid using any printing methods like `puts` or `p` for that purpose.

On the other hand, regular comments do not use an arrow:

```
#   polymorphic_url(record)  # same as comment_url(record)
```

Booleans

In predicates and flags prefer documenting boolean semantics over exact values.

When "true" or "false" are used as defined in Ruby use regular font. The singletons `true` and `false` need fixed-width font. Please avoid terms like "truthy", Ruby defines what is true and false in the language, and thus those words have a technical meaning and need no substitutes.

As a rule of thumb, do not document singletons unless absolutely necessary. That prevents artificial constructs like `!!` or ternaries, allows refactors, and the code does not need to rely on the exact values returned by methods being called in the implementation.

For example:

```
`config.action_mailer.perform_deliveries` specifies whether mail will actually be delivered
```

the user does not need to know which is the actual default value of the flag, and so we only document its boolean semantics.

An example with a predicate:

```
# Returns true if the collection is empty.
#
# If the collection has been loaded
# it is equivalent to <tt>collection.size.zero?</tt>. If the
# collection has not been loaded, it is equivalent to
# <tt>!collection.exists?</tt>. If the collection has not already been
# loaded and you are going to fetch the records anyway it is better to
# check <tt>collection.length.zero?</tt>.
def empty?
  if loaded?
    size.zero?
  else
    @target.blank? && !scope.exists?
  end
end
```

The API is careful not to commit to any particular value, the method has predicate semantics, that's enough.

File Names

As a rule of thumb, use filenames relative to the application root:

```
config/routes.rb            # YES
routes.rb                   # NO
RAILS_ROOT/config/routes.rb # NO
```

Fonts

Fixed-width Font

Use fixed-width fonts for:

- Constants, in particular class and module names.
- Method names.
- Literals like `nil`, `false`, `true`, `self`.
- Symbols.
- Method parameters.
- File names.

```ruby
class Array
  # Calls +to_param+ on all its elements and joins the result with
  # slashes. This is used by +url_for+ in Action Pack.
  def to_param
    collect { |e| e.to_param }.join '/'
  end
end
```

WARNING: Using `+...+` for fixed-width font only works with simple content like ordinary method names, symbols, paths (with forward slashes), etc. Please use `<tt>...</tt>` for everything else, notably class or module names with a namespace as in `<tt>ActiveRecord::Base</tt>`.

You can quickly test the RDoc output with the following command:

```
$ echo "+:to_param+" | rdoc --pipe
# => <p><code>:to_param</code></p>
```

Regular Font

When "true" and "false" are English words rather than Ruby keywords use a regular font:

```ruby
# Runs all the validations within the specified context.
# Returns true if no errors are found, false otherwise.
#
# If the argument is false (default is +nil+), the context is
# set to <tt>:create</tt> if <tt>new_record?</tt> is true,
# and to <tt>:update</tt> if it is not.
```

```
#
# Validations with no <tt>:on</tt> option will run no
# matter the context. Validations with # some <tt>:on</tt>
# option will only run in the specified context.
def valid?(context = nil)
  # ...
end
```

Description Lists

In lists of options, parameters, etc. use a hyphen between the item and its description (reads better than a colon because normally options are symbols):

```
# * <tt>:allow_nil</tt> - Skip validation if attribute is +nil+.
```

The description starts in upper case and ends with a full stop-it's standard English.

Dynamically Generated Methods

Methods created with `(module|class)_eval(STRING)` have a comment by their side with an instance of the generated code. That comment is 2 spaces away from the template:

```
for severity in Severity.constants
  class_eval <<-EOT, __FILE__, __LINE__ + 1
    def #{severity.downcase}(message = nil, progname = nil, &block)  # def debug(message =
      add(#{severity}, message, progname, &block)                    #   add(DEBUG, message
    end                                                              # end
                                                                     #
    def #{severity.downcase}?                                        # def debug?
      #{severity} >= @level                                          #   DEBUG >= @level
    end                                                              # end
  EOT
end
```

If the resulting lines are too wide, say 200 columns or more, put the comment above the call:

```
# def self.find_by_login_and_activated(*args)
#   options = args.extract_options!
#   ...
# end
self.class_eval %{
  def self.#{method_id}(*args)
    options = args.extract_options!
    ...
  end
}
```

Method Visibility

When writing documentation for Rails, it's important to understand the difference between public user-facing API vs internal API.

Rails, like most libraries, uses the private keyword from Ruby for defining internal API. However, public API follows a slightly different convention. Instead of assuming all public methods are designed for user consumption, Rails uses the `:nodoc:` directive to annotate these kinds of methods as internal API.

This means that there are methods in Rails with `public` visibility that aren't meant for user consumption.

An example of this is `ActiveRecord::Core::ClassMethods#arel_table`:

```ruby
module ActiveRecord::Core::ClassMethods
  def arel_table # :nodoc:
    # do some magic..
  end
end
```

If you thought, "this method looks like a public class method for `ActiveRecord::Core`", you were right. But actually the Rails team doesn't want users to rely on this method. So they mark it as `:nodoc:` and it's removed from public documentation. The reasoning behind this is to allow the team to change these methods according to their internal needs across releases as they see fit. The name of this method could change, or the return value, or this entire class may disappear; there's no guarantee and so you shouldn't depend on this API in your plugins or applications. Otherwise, you risk your app or gem breaking when you upgrade to a newer release of Rails.

As a contributor, it's important to think about whether this API is meant for end-user consumption. The Rails team is committed to not making any breaking changes to public API across releases without going through a full deprecation cycle. It's recommended that you `:nodoc:` any of your internal methods/classes unless they're already private (meaning visibility), in which case it's internal by default. Once the API stabilizes the visibility can change, but changing public API is much harder due to backwards compatibility.

A class or module is marked with `:nodoc:` to indicate that all methods are internal API and should never be used directly.

To summarize, the Rails team uses `:nodoc:` to mark publicly visible methods and classes for internal use; changes to the visibility of API should be considered carefully and discussed over a pull request first.

Regarding the Rails Stack

When documenting parts of Rails API, it's important to remember all of the pieces that go into the Rails stack.

This means that behavior may change depending on the scope or context of the method or class you're trying to document.

In various places there is different behavior when you take the entire stack into account, one such example is `ActionView::Helpers::AssetTagHelper#image_tag`:

```
# image_tag("icon.png")
#   # => <img src="/assets/icon.png" />
```

Although the default behavior for `#image_tag` is to always return `/images/icon.png`, we take into account the full Rails stack (including the Asset Pipeline) we may see the result seen above.

We're only concerned with the behavior experienced when using the full default Rails stack.

In this case, we want to document the behavior of the *framework*, and not just this specific method.

If you have a question on how the Rails team handles certain API, don't hesitate to open a ticket or send a patch to the issue tracker.