Contributing to Ruby on Rails

January 13, 2015

This guide covers ways in which *you* can become a part of the ongoing development of Ruby on Rails. After reading this guide, you will know:

- How to use GitHub to report issues.
- How to clone master and run the test suite.
- How to help resolve existing issues.
- How to contribute to the Ruby on Rails documentation.
- How to contribute to the Ruby on Rails code.

Ruby on Rails is not "someone else's framework." Over the years, hundreds of people have contributed to Ruby on Rails ranging from a single character to massive architectural changes or significant documentation - all with the goal of making Ruby on Rails better for everyone. Even if you don't feel up to writing code or documentation yet, there are a variety of other ways that you can contribute, from reporting issues to testing patches.

1 Reporting an Issue

Ruby on Rails uses GitHub Issue Tracking to track issues (primarily bugs and contributions of new code). If you've found a bug in Ruby on Rails, this is the place to start. You'll need to create a (free) GitHub account in order to submit an issue, to comment on them or to create pull requests.

Bugs in the most recent released version of Ruby on Rails are likely to get the most attention. Also, the Rails core team is always interested in feedback from those who can take the time to test *edge Rails* (the code for the version of Rails that is currently under development). Later in this guide you'll find out how to get edge Rails for testing.

1.1 Creating a Bug Report

If you've found a problem in Ruby on Rails which is not a security risk, do a search in GitHub under Issues in case it has already been reported. If you do not find any issue addressing it you may proceed to open a new one. (See the next section for reporting security issues.)

Your issue report should contain a title and a clear description of the issue at the bare minimum. You should include as much relevant information as possible and should at least post a code sample that demonstrates the issue. It would be even better if you could include a unit test that shows how the expected behavior is not occurring. Your goal should be to make it easy for yourself - and others - to replicate the bug and figure out a fix.

Then, don't get your hopes up! Unless you have a "Code Red, Mission Critical, the World is Coming to an End" kind of bug, you're creating this issue report in the hope that others with the same problem will be able to collaborate with you on solving it. Do not expect that the issue report will automatically see any activity or that others will jump to fix it. Creating an issue like this is mostly to help yourself start on the path of fixing the problem and for others to confirm it with an "I'm having this problem too" comment.

1.2 Create a Self-Contained gist for Active Record and Action Controller Issues

If you are filing a bug report, please use Active Record template for gems or Action Controller template for gems if the bug is found in a published gem, and Active Record template for master or Action Controller template for master if the bug happens in the master branch.

1.3 Special Treatment for Security Issues

Please do not report security vulnerabilities with public GitHub issue reports. The Rails security policy page details the procedure to follow for security issues.

1.4 What about Feature Requests?

Please don't put "feature request" items into GitHub Issues. If there's a new feature that you want to see added to Ruby on Rails, you'll need to write the code yourself - or convince someone else to partner with you to write the code. Later in this guide you'll find detailed instructions for proposing a patch to Ruby on Rails. If you enter a wish list item in GitHub Issues with no code, you can expect it to be marked "invalid" as soon as it's reviewed.

Sometimes, the line between 'bug' and 'feature' is a hard one to draw. Generally, a feature is anything that adds new behavior, while a bug is anything that fixes already existing behavior that is misbehaving. Sometimes, the core team will have to make a judgement call. That said, the distinction generally just affects which release your patch will get in to; we love feature submissions! They just won't get backported to maintenance branches.

If you'd like feedback on an idea for a feature before doing the work for make a patch, please send an email to the rails-core mailing list. You might get no response, which means that everyone is indifferent. You might find someone who's also interested in building that feature. You might get a "This won't be accepted." But it's the proper place to discuss new ideas. GitHub Issues are not a particularly good venue for the sometimes long and involved discussions new features require.

2 Helping to Resolve Existing Issues

As a next step beyond reporting issues, you can help the core team resolve existing issues. If you check the Everyone's Issues list in GitHub Issues, you'll find lots of issues already requiring attention. What can you do for these? Quite a bit, actually:

2.1 Verifying Bug Reports

For starters, it helps just to verify bug reports. Can you reproduce the reported issue on your own computer? If so, you can add a comment to the issue saying that you're seeing the same thing.

If something is very vague, can you help squash it down into something specific? Maybe you can provide additional information to help reproduce a bug, or help by eliminating needless steps that aren't required to demonstrate the problem.

If you find a bug report without a test, it's very useful to contribute a failing test. This is also a great way to get started exploring the source code: looking at the existing test files will teach you how to write more tests. New tests are best contributed in the form of a patch, as explained later on in the "Contributing to the Rails Code" section.

Anything you can do to make bug reports more succinct or easier to reproduce is a help to folks trying to write code to fix those bugs - whether you end up writing the code yourself or not.

2.2 Testing Patches

You can also help out by examining pull requests that have been submitted to Ruby on Rails via GitHub. To apply someone's changes you need first to create a dedicated branch:

```
$ git checkout -b testing_branch
```

Then you can use their remote branch to update your codebase. For example, let's say the GitHub user JohnSmith has forked and pushed to a topic branch "orange" located at https://github.com/JohnSmith/rails.

```
$ git remote add JohnSmith git://github.com/JohnSmith/rails.git
$ git pull JohnSmith orange
```

After applying their branch, test it out! Here are some things to think about:

- Does the change actually work?
- Are you happy with the tests? Can you follow what they're testing? Are there any tests missing?
- Does it have the proper documentation coverage? Should documentation elsewhere be updated?
- Do you like the implementation? Can you think of a nicer or faster way to implement a part of their change?

Once you're happy that the pull request contains a good change, comment on the GitHub issue indicating your approval. Your comment should indicate that you like the change and what you like about it. Something like:

> I like the way you've restructured that code in generate_finder_sql - much nicer. The tests look good too.

If your comment simply says "+1", then odds are that other reviewers aren't going to take it too seriously. Show that you took the time to review the pull request.

3 Contributing to the Rails Documentation

Ruby on Rails has two main sets of documentation: the guides, which help you learn about Ruby on Rails, and the API, which serves as a reference.

You can help improve the Rails guides by making them more coherent, consistent or readable, adding missing information, correcting factual errors, fixing typos, or bringing it up to date with the latest edge Rails. To get involved in the translation of Rails guides, please see Translating Rails Guides.

You can either open a pull request to Rails or ask the Rails core team for commit access on docrails if you contribute regularly. Please do not open pull requests in docrails, if you'd like to get feedback on your change, ask for it in Rails instead.

Docrails is merged with master regularly, so you are effectively editing the Ruby on Rails documentation.

If you are unsure of the documentation changes, you can create an issue in the Rails issues tracker on GitHub.

When working with documentation, please take into account the API Documentation Guidelines and the Ruby on Rails Guides Guidelines.

As explained earlier, ordinary code patches should have proper documentation coverage. Docrails is only used for isolated documentation improvements.

To help our CI servers you should add [ci skip] to your documentation commit message to skip build on that commit. Please remember to use it for commits containing only documentation changes.

Docrails has a very strict policy: no code can be touched whatsoever, no matter how trivial or small the change. Only RDoc and guides can be edited via docrails. Also, CHANGELOGs should never be edited in docrails.

4 Contributing to the Rails Code

4.1 Setting Up a Development Environment

To move on from submitting bugs to helping resolve existing issues or contributing your own code to Ruby on Rails, you *must* be able to run its test suite. In this section of the guide you'll learn how to setup the tests on your own computer.

4.1.1 The Easy Way The easiest and recommended way to get a development environment ready to hack is to use the Rails development box.

4.1.2 The Hard Way In case you can't use the Rails development box, see this other guide.

4.2 Clone the Rails Repository

To be able to contribute code, you need to clone the Rails repository:

```
$ git clone git://github.com/rails/rails.git
```

and create a dedicated branch:

```
$ cd rails
$ git checkout -b my_new_branch
```

It doesn't matter much what name you use, because this branch will only exist on your local computer and your personal repository on GitHub. It won't be part of the Rails Git repository.

4.3 Running an Application Against Your Local Branch

In case you need a dummy Rails app to test changes, the `--dev` flag of `rails new` generates an application that uses your local branch:

```
$ cd rails
$ bundle exec rails new ~/my-test-app --dev
```

The application generated in `~/my-test-app` runs against your local branch and in particular sees any modifications upon server reboot.

4.4 Write Your Code

Now get busy and add/edit code. You're on your branch now, so you can write whatever you want (make sure you're on the right branch with `git branch -a`). But if you're planning to submit your change back for inclusion in Rails, keep a few things in mind:

- Get the code right.
- Use Rails idioms and helpers.
- Include tests that fail without your code, and pass with it.
- Update the (surrounding) documentation, examples elsewhere, and the guides: whatever is affected by your contribution.

Changes that are cosmetic in nature and do not add anything substantial to the stability, functionality, or testability of Rails will generally not be accepted (read more about our rationales behind this decision).

4.4.1 Follow the Coding Conventions Rails follows a simple set of coding style conventions:

- Two spaces, no tabs (for indentation).
- No trailing whitespace. Blank lines should not have any spaces.
- Indent after private/protected.
- Use Ruby >= 1.9 syntax for hashes. Prefer `{ a: :b }` over `{ :a => :b }`.
- Prefer `&&`/`||` over `and`/`or`.
- Prefer class << self over self.method for class methods.
- Use `MyClass.my_method(my_arg)` not `my_method( my_arg )` or `my_method my_arg`.
- Use `a = b` and not `a=b`.
- Use assert_not methods instead of refute.
- Prefer `method { do_stuff }` instead of `method{do_stuff}` for single-line blocks.
- Follow the conventions in the source you see used already.

The above are guidelines - please use your best judgment in using them.

4.5 Benchmark Your Code

If your change has an impact on the performance of Rails, please use the benchmark-ips gem to provide benchmark results for comparison.

Here's an example of using benchmark-ips:

```
require 'benchmark/ips'

Benchmark.ips do |x|
  x.report('addition') { 1 + 2 }
  x.report('addition with send') { 1.send(:+, 2) }
end
```

This will generate a report with the following information:

```
Calculating -------------------------------------
            addition     69114 i/100ms
  addition with send     64062 i/100ms
-------------------------------------------------
            addition  5307644.4 (± 3.5%) i/s -   26539776 in   5.007219s
  addition with send  3702897.9 (± 3.5%) i/s -   18513918 in   5.006723s
```

Please see the benchmark/ips README for more information.

4.6 Running Tests

It is not customary in Rails to run the full test suite before pushing changes. The railties test suite in particular takes a long time, and even more if the source code is mounted in `/vagrant` as happens in the recommended workflow with the rails-dev-box.

As a compromise, test what your code obviously affects, and if the change is not in railties, run the whole test suite of the affected component. If all tests are passing, that's enough to propose your contribution. We have Travis CI as a safety net for catching unexpected breakages elsewhere.

4.6.1 Entire Rails: To run all the tests, do:

```
$ cd rails
$ bundle exec rake test
```

4.6.2 For a Particular Component You can run tests only for a particular component (e.g. Action Pack). For example, to run Action Mailer tests:

```
$ cd actionmailer
$ bundle exec rake test
```

4.6.3 Running a Single Test You can run a single test through ruby. For instance:

```
$ cd actionmailer
$ ruby -w -Itest test/mail_layout_test.rb -n test_explicit_class_layout
```

The `-n` option allows you to run a single method instead of the whole file.

4.6.3.1 Testing Active Record

This is how you run the Active Record test suite only for SQLite3:

```
$ cd activerecord
$ bundle exec rake test:sqlite3
```

You can now run the tests as you did for `sqlite3`. The tasks are respectively

```
test:mysql
test:mysql2
test:postgresql
```

Finally,

```
$ bundle exec rake test
```

will now run the four of them in turn.
You can also run any single test separately:

```
$ ARCONN=sqlite3 ruby -Itest test/cases/associations/has_many_associations_test.rb
```

To run a single test against all adapters, use:

```
$ bundle exec rake TEST=test/cases/associations/has_many_associations_test.rb
```

You can invoke `test_jdbcmysql`, `test_jdbcsqlite3` or `test_jdbcpostgresql` also. See the file `activerecord/RUNNING_UNIT_TESTS.rdoc` for information on running more targeted database tests, or the file `ci/travis.rb` for the test suite run by the continuous integration server.

4.7 Warnings

The test suite runs with warnings enabled. Ideally, Ruby on Rails should issue no warnings, but there may be a few, as well as some from third-party libraries. Please ignore (or fix!) them, if any, and submit patches that do not issue new warnings.

If you are sure about what you are doing and would like to have a more clear output, there's a way to override the flag:

```
$ RUBYOPT=-W0 bundle exec rake test
```

4.8 Updating the CHANGELOG

The CHANGELOG is an important part of every release. It keeps the list of changes for every Rails version.

You should add an entry to the CHANGELOG of the framework that you modified if you're adding or removing a feature, committing a bug fix or adding deprecation notices. Refactorings and documentation changes generally should not go to the CHANGELOG.

A CHANGELOG entry should summarize what was changed and should end with author's name and it should go on top of a CHANGELOG. You can use multiple lines if you need more space and you can attach code examples indented with 4 spaces. If a change is related to a specific issue, you should attach the issue's number. Here is an example CHANGELOG entry:

```
*   Summary of a change that briefly describes what was changed. You can use multiple
    lines and wrap them at around 80 characters. Code examples are ok, too, if needed:

        class Foo
          def bar
            puts 'baz'
          end
        end

    You can continue after the code example and you can attach issue number. GH#1234

    *Your Name*
```

Your name can be added directly after the last word if you don't provide any code examples or don't need multiple paragraphs. Otherwise, it's best to make as a new paragraph.

4.9 Sanity Check

You should not be the only person who looks at the code before you submit it. If you know someone else who uses Rails, try asking them if they'll check out your work. If you don't know anyone else using Rails, try hopping into the IRC room or posting about your idea to the rails-core mailing list. Doing this in private before you push a patch out publicly is the "smoke test" for a patch: if you can't convince one other developer of the beauty of your code, you're unlikely to convince the core team either.

4.10 Commit Your Changes

When you're happy with the code on your computer, you need to commit the changes to Git:

```
$ git commit -a
```

At this point, your editor should be fired up and you can write a message for this commit. Well formatted and descriptive commit messages are extremely helpful for the others, especially when figuring out why given change was made, so please take the time to write it.

Good commit message should be formatted according to the following example:

```
Short summary (ideally 50 characters or less)

More detailed description, if necessary. It should be wrapped to 72
characters. Try to be as descriptive as you can, even if you think that
the commit content is obvious, it may not be obvious to others. You
should add such description also if it's already present in bug tracker,
it should not be necessary to visit a webpage to check the history.

Description can have multiple paragraphs and you can use code examples
inside, just indent it with 4 spaces:

    class ArticlesController
```

```
      def index
        render json: Article.limit(10)
      end
    end

You can also add bullet points:

- you can use dashes or asterisks

- also, try to indent next line of a point for readability, if it's too
  long to fit in 72 characters
```

Please squash your commits into a single commit when appropriate. This simplifies future cherry picks, and also keeps the git log clean.

4.11 Update Your Branch

It's pretty likely that other changes to master have happened while you were working. Go get them:

```
$ git checkout master
$ git pull --rebase
```

Now reapply your patch on top of the latest changes:

```
$ git checkout my_new_branch
$ git rebase master
```

No conflicts? Tests still pass? Change still seems reasonable to you? Then move on.

4.12 Fork

Navigate to the Rails GitHub repository and press "Fork" in the upper right hand corner.

Add the new remote to your local repository on your local machine:

```
$ git remote add mine git@github.com:<your user name>/rails.git
```

Push to your remote:

```
$ git push mine my_new_branch
```

You might have cloned your forked repository into your machine and might want to add the original Rails repository as a remote instead, if that's the case here's what you have to do.

In the directory you cloned your fork:

```
$ git remote add rails git://github.com/rails/rails.git
```

Download new commits and branches from the official repository:

```
$ git fetch rails
```

Merge the new content:

```
$ git checkout master
$ git rebase rails/master
```

Update your fork:

```
$ git push origin master
```

If you want to update another branch:

```
$ git checkout branch_name
$ git rebase rails/branch_name
$ git push origin branch_name
```

4.13 Issue a Pull Request

Navigate to the Rails repository you just pushed to (e.g. https://github.com/your-user-name/rails) and click on "Pull Requests" seen in the right panel. On the next page, press "New pull request" in the upper right hand corner.

Click on "Edit", if you need to change the branches being compared (it compares "master" by default) and press "Click to create a pull request for this comparison".

Ensure the changesets you introduced are included. Fill in some details about your potential patch including a meaningful title. When finished, press "Send pull request". The Rails core team will be notified about your submission.

4.14 Get some Feedback

Most pull requests will go through a few iterations before they get merged. Different contributors will sometimes have different opinions, and often patches will need revised before they can get merged.

Some contributors to Rails have email notifications from GitHub turned on, but others do not. Furthermore, (almost) everyone who works on Rails is a volunteer, and so it may take a few days for you to get your first feedback on a pull request. Don't despair! Sometimes it's quick, sometimes it's slow. Such is the open source life.

If it's been over a week, and you haven't heard anything, you might want to try and nudge things along. You can use the rubyonrails-core mailing list for this. You can also leave another comment on the pull request.

While you're waiting for feedback on your pull request, open up a few other pull requests and give someone else some! I'm sure they'll appreciate it in the same way that you appreciate feedback on your patches.

4.15 Iterate as Necessary

It's entirely possible that the feedback you get will suggest changes. Don't get discouraged: the whole point of contributing to an active open source project is to tap into the knowledge of the community. If people are encouraging you to tweak your code, then it's worth making the tweaks and resubmitting. If the feedback is that your code doesn't belong in the core, you might still think about releasing it as a gem.

4.15.1 Squashing commits One of the things that we may ask you to do is to "squash your commits", which will combine all of your commits into a single commit. We prefer pull requests that are a single commit. This makes it easier to backport changes to stable branches, squashing makes it easier to revert bad commits, and the git history can be a bit easier to follow. Rails is a large project, and a bunch of extraneous commits can add a lot of noise.

In order to do this, you'll need to have a git remote that points at the main Rails repository. This is useful anyway, but just in case you don't have it set up, make sure that you do this first:

```
$ git remote add upstream https://github.com/rails/rails.git
```

You can call this remote whatever you'd like, but if you don't use `upstream`, then change the name to your own in the instructions below.

Given that your remote branch is called `my_pull_request`, then you can do the following:

```
$ git fetch upstream
$ git checkout my_pull_request
$ git rebase upstream/master
$ git rebase -i

< Choose 'squash' for all of your commits except the first one. >
< Edit the commit message to make sense, and describe all your changes. >

$ git push origin my_pull_request -f
```

You should be able to refresh the pull request on GitHub and see that it has been updated.

4.15.2 Updating pull request Sometimes you will be asked to make some changes to the code you have already committed. This can include amending existing commits. In this case Git will not allow you to push the changes as the pushed branch and local branch do not match. Instead of opening a new pull request, you can force push to your branch on GitHub as described earlier in squashing commits section:

```
$ git push origin my_pull_request -f
```

This will update the branch and pull request on GitHub with your new code. Do note that using force push may result in commits being lost on the remote branch; use it with care.

4.16 Older Versions of Ruby on Rails

If you want to add a fix to older versions of Ruby on Rails, you'll need to set up and switch to your own local tracking branch. Here is an example to switch to the 4-0-stable branch:

```
$ git branch --track 4-0-stable origin/4-0-stable
$ git checkout 4-0-stable
```

You may want to put your Git branch name in your shell prompt to make it easier to remember which version of the code you're working with.

4.16.1 Backporting Changes that are merged into master are intended for the next major release of Rails. Sometimes, it might be beneficial for your changes to propagate back to the maintenance releases for older stable branches. Generally, security fixes and bug fixes are good candidates for a backport, while new features and patches that introduce a change in behavior will not be accepted. When in doubt, it is best to consult a Rails team member before backporting your changes to avoid wasted effort.

For simple fixes, the easiest way to backport your changes is to extract a diff from your changes in master and apply them to the target branch.

First make sure your changes are the only difference between your current branch and master:

```
$ git log master..HEAD
```

Then extract the diff:

```
$ git format-patch master --stdout > ~/my_changes.patch
```

Switch over to the target branch and apply your changes:

```
$ git checkout -b my_backport_branch 3-2-stable
$ git apply ~/my_changes.patch
```

This works well for simple changes. However, if your changes are complicated or if the code in master has deviated significantly from your target branch, it might require more work on your part. The difficulty of a backport varies greatly from case to case, and sometimes it is simply not worth the effort.

Once you have resolved all conflicts and made sure all the tests are passing, push your changes and open a separate pull request for your backport. It is also worth noting that older branches might have a different set of build targets than master. When possible, it is best to first test your backport locally against the Ruby versions listed in `.travis.yml` before submitting your pull request.

And then... think about your next contribution!

5 Rails Contributors

All contributions, either via master or docrails, get credit in Rails Contributors.

6 Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our documentation contributions section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check Edge Guides first to verify if the issues are already fixed or not on the master branch. Check the Ruby on Rails Guides Guidelines for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please open an issue.

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the rubyonrails-docs mailing list.
