

Contributing to Rails ^{*1}

This guide covers ways in which *you* can become a part of the ongoing development of Rails. After reading it, you should be familiar with:

- Using Lighthouse to report issues with Rails
- Cloning edge Rails and running the test suite
- Helping to resolve existing issues
- Contributing to the Rails documentation
- Contributing to the Rails code

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

1 Reporting a Rails Issue

Rails uses a "*Lighthouse project*" to track issues (primarily bugs and contributions of new code). If you've found a bug in Rails, this is the place to start.

NOTE: Bugs in the most recent released version of 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 Rails, you can start by "*adding a new ticket*" to the Rails Lighthouse. At the minimum, your ticket needs a title and descriptive text. But that's only a minimum. You should include as much relevant information as possible. You need to at least post the code sample that has the issue. Even better is to 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.

You shouldn't assign the bug to a particular core developer (through the *Who's Responsible*

*1 Original: <http://guides.rubyonrails.org/contributing-to-rails.html>

select list) unless you know for sure which developer will be handling any patch. The core team periodically reviews issues and assigns developers and milestones to them.

You should set tags for your issue. Use the "bug" tag for a bug report, and add the "patch" tag if you are attaching a patch. Try to find some relevant tags from the existing tag list (which will appear as soon as you start typing in the *Choose some tags* textbox), rather than creating new tags.

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 ticket in the hope that others with the same problem will be able to collaborate with you on solving it. Do not expect that the ticket automatically will see any activity or that others will jump to fix it. Creating a ticket like this is mostly to help yourself start on the path of fixing the problem and for others to confirm it with a "I'm having this problem too" comment.

1.2 Special Treatment for Security Issues

If you've found a security vulnerability in Rails, please do *not* report it via a Lighthouse ticket. Lighthouse tickets are public as soon as they are entered. Instead, you should use the dedicated email address "*security@rubyonrails.org*" to report any vulnerabilities. This alias is monitored and the core team will work with you to quickly and completely address any such vulnerabilities.

1.3 What About Feature Requests?

Please don't put "feature request" tickets into Lighthouse. If there's a new feature that you want to see added to 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 Rails. If you enter a wishlist item in Lighthouse with no code, you can expect it to be marked "invalid" as soon as it's reviewed.

2 Running the Rails Test Suite

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

2.1 Install git

Rails uses git for source code control. You won't be able to do anything without the Rails source code, and this is a prerequisite. The “*git homepage*” has installation instructions. If you're on OS X, use the “*Git for OS X*” installer. If you're unfamiliar with git, there are a variety of resources on the net that will help you learn more:

- “*Everyday Git*” will teach you just enough about git to get by.
- The “*PeepCode screencast*” on git (\$9) is easier to follow.
- “*GitHub*” offers links to a variety of git resources.

2.2 Get the Rails Source Code

Don't fork the main Rails repository. Instead, you want to clone it to your own computer. Navigate to the folder where you want the source code (it will create its own `/rails` subdirectory) and run:

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

2.3 Set up and Run the Tests

All of the Rails tests must pass with any code you submit, otherwise you have no chance of getting code accepted. This means you need to be able to run the tests. For the tests that touch the database, this means creating the databases. If you're using MySQL:

```
mysql> create database activerecord_unittest;
mysql> create database activerecord_unittest2;
mysql> GRANT ALL PRIVILEGES ON activerecord_unittest.*
      to 'rails'@'localhost';
mysql> GRANT ALL PRIVILEGES ON activerecord_unittest2.*
      to 'rails'@'localhost';
```

If you're using another database, check the files under `activerecord/test/connections` in the Rails source code for default connection information. You can edit these files if you *must* on your machine to provide different credentials, but obviously you should not push any such changes back to Rails.

Now if you go back to the root of the Rails source on your machine and run `rake` with no parameters, you should see every test in all of the Rails components pass. If you want to run the all ActiveRecord tests (or just a single one) with another database adapter, enter this from the `activerecord` directory:

```
rake test_sqlite3
rake test_sqlite3 TEST=test/cases/validations_test.rb
```

You can change `sqlite3` with `jdbcmysql`, `jdbcsqlite3`, `jdbcpostgresql`, `mysql` or `postgresql`. Check out the file `activerecord/RUNNING_UNIT_TESTS` for information on running more targeted database tests, or the file `ci/ci_build.rb` to see the test suite that the Rails continuous integration server runs.

NOTE: If you're working with Active Record code, you must ensure that the tests pass for at least MySQL, PostgreSQL, SQLite 2, and SQLite 3. Subtle differences between the various Active Record database adapters have been behind the rejection of many patches that looked OK when tested only against MySQL.

3 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 “*open tickets*” list in Lighthouse, you’ll find hundreds of issues already requiring attention. What can you do for these? Quite a bit, actually:

3.1 Verifying Bug Reports

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

If something is very vague, can you help squish it down into something specific? Maybe you can provide additional information to help reproduce a bug, or eliminate needless steps that aren’t required to help 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 Rails source: looking at the existing test files will teach you how to write more tests for Rails. 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.

3.2 Testing Patches

You can also help out by examining patches that have been submitted to Rails via Lighthouse. To apply someone's changes you need to first create a branch of the Rails source code:

```
git checkout -b testing_branch
```

Then you can apply their patch:

```
git am < their-patch-file.diff
```

After applying a patch, test it out! Here are some things to think about:

- Does the patch actually work?
- Are you happy with the tests? Can you follow what they're testing? Are there any tests missing?
- Does the documentation still seem right to you?
- 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 patch contains a good change, comment on the Lighthouse ticket 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 patch. Once three people have approved it, add the "verified" tag. This will bring it to the attention of a core team member who will review the changes looking for the same kinds of things.

4 Contributing to the Rails Documentation

Another area where you can help out if you're not yet ready to take the plunge to writing Rails core code is with Rails documentation. You can help with the Rails Guides or the Rails API documentation.

INFO: "docrails" is the documentation branch for Rails with an open commit policy. You can simply PM "lifo" on Github and ask for the commit rights. Documentation

changes made as part of the “docrails” project, are merged back to the Rails master code from time to time. Check out the “original announcement” for more details.

4.1 The Rails Guides

The “*Rails Guides*” are a set of online resources that are designed to make people productive with Rails and to understand how all of the pieces fit together. These guides (including this one!) are written as part of the “*docrails*” project. If you have an idea for a new guide, or improvements for an existing guide, you can refer to the “*contribution page*” for instructions on getting involved.

4.2 The Rails API Documentation

The “*Rails API documentation*” is automatically generated from the Rails source code via “*RDoc*”. If you find some part of the documentation to be incomplete, confusing, or just plain wrong, you can step in and fix it.

To contribute an update to the API documentation, you can contact “*lifo*” on GitHub and ask for commit rights to the *docrails* repository and push your changes to the *docrails* repository. Please follow the “*docrails RDoc conventions*” when contributing the changes.

5 The Rails Wiki

The “*Rails wiki*” is a collection of user-generated and freely-editable information about Rails. It covers everything from getting started to FAQs to how-tos and popular plugins. To contribute to the wiki, just find some useful information that isn’t there already and add it. There are style guidelines to help keep the wiki a coherent resources; see the section on “*contributing to the wiki*” for more details.

6 Contributing to the Rails Code

When you’re ready to take the plunge, one of the most helpful ways to contribute to Rails is to actually submit source code. Here’s a step-by-step listing of the things you need to do to make this a successful experience.

6.1 Learn the Language and the Framework

Learn at least *something* about Ruby and Rails. If you don’ t understand the syntax of the language, common Ruby idioms, and the code that already exists in Rails, you’ re

unlikely to be able to build a good patch (that is, one that will get accepted). You don't have to know every in-and-out of the language and the framework; some of the Rails code is fiendishly complex. But Rails is probably not appropriate as the first place that you ever write Ruby code. You should at least understand (though not necessarily memorize) “*The Ruby Programming Language*” and have browsed the Rails source code.

6.2 Fork the Rails Source Code

Fork Rails. You're not going to put your patches right into the master branch, OK? This is where you need that copy of Rails that you cloned earlier. Think of a name for your new branch and run

```
git checkout -b my_new_branch
```

It doesn't really matter what name you use, because this branch will only exist on your local computer.

6.3 Write Your Code

Now get busy and add your code to Rails (or edit the existing code). You're on your branch now, so you can write whatever you want (you can check to 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 documentation

6.4 Sanity Check

You should not be the only person who looks at the code before you submit it. You know at least one other Rails developer, right? Show them what you're doing and ask for feedback. 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.

6.5 Commit Your Changes

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

```
git commit -a -m "Here is a commit message"
```

6.6 Update Rails

Update your copy of Rails. It's pretty likely that other changes to core Rails have happened while you were working. Go get them:

```
git checkout master
git pull
```

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.

6.7 Create a Patch

Now you can create a patch file to share with other developers (and with the Rails core team). Still in your branch, run

```
git commit -a
git format-patch master --stdout > my_new_patch.diff
```

Sanity check the results of this operation: open the diff file in your text editor of choice and make sure that no unintended changes crept in.

6.8 Create a Lighthouse Ticket

Now create a ticket with your patch. Go to the *"new ticket"* page at Lighthouse. Fill in a reasonable title and description, remember to attach your patch file, and tag the ticket with the 'patch' tag and whatever other subject area tags make sense.

6.9 Get Some Feedback

Now you need to get other people to look at your patch, just as you've looked at other people's patches. You can use the `rubyonrails-core` mailing list or the `#rails-contrib` channel on IRC freenode for this. You might also try just talking to Rails developers that you know.

6.10 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 community knowledge. 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 plugin.

And then—think about your next contribution!

7 Changelog

“Lighthouse ticket”

- March 2, 2009: Initial draft by *“Mike Gunderloy”*

This work is licensed under a *“Creative Commons Attribution-Share Alike 3.0”* License. “Rails”, “Ruby on Rails”, and the Rails logo are trademarks of David Heinemeier Hansson. All rights reserved.