Performance Testing Rails Applications

This guide covers the various ways of performance testing a Ruby on Rails application. By referring to this guide, you will be able to:

Understand the various types of benchmarking and profiling metrics

Generate performance and benchmarking tests

Install and use a GC-patched Ruby binary to measure memory usage and object allocation

Understand the benchmarking information provided by Rails inside the log files

Learn about various tools facilitating benchmarking and profiling

Performance testing is an integral part of the development cycle. It is very important that you don't make your end users wait for too long before the page is completely loaded. Ensuring a pleasant browsing experience for end users and cutting the cost of unnecessary hardware is important for any non-trivial web application.

Chapters

1 Performance Test Cases

Rails performance tests are a special type of integration tests, designed for benchmarking and profiling the test code. With performance tests, you can determine where your application's memory or speed problems are coming from, and get a more in-depth picture of those problems.

In a freshly generated Rails application, `test/performance/browsing_test.rb` contains an example of a performance test:

```
require
'test_helper'
require
'rails/performance_test_help'

# Profiling results for each test method are written to tmp/performance.
class

BrowsingTest < ActionDispatch::PerformanceTest

def

test_homepage

get
'/'

end
end
```

This example is a simple performance test case for profiling a GET request to the application's homepage.

1.1 Generating Performance Tests

Rails provides a generator called `performance_test` for creating new performance tests:

```
$ rails generate performance_test homepage
```

This generates `homepage_test.rb` in the `test/performance` directory:

```
require
'test_helper'
require
'rails/performance_test_help'

class

HomepageTest < ActionDispatch::PerformanceTest

# Replace this with your real tests.

def

test_homepage

get
'/'

end
end
```

1.2 Examples

Let's assume your application has the following controller and model:

```
# routes.rb
root
:to

=>
'home#index'
resources
:posts

# home_controller.rb
class

HomeController < ApplicationController

def

dashboard

@users
```

```
= User.last_ten.includes(
:avatars
)

@posts

= Post.all_today

end
end

# posts_controller.rb
class

PostsController < ApplicationController

def

create

@post

= Post.create(params[
:post
])

redirect_to(
@post
)

end
end

# post.rb
class

Post < ActiveRecord::Base

before_save
:recalculate_costly_stats


def

slow_method

# I fire gallzilion queries sleeping all around

end


private


def

recalculate_costly_stats

# CPU heavy calculations

end
end
```

1.2.1 Controller Example

Because performance tests are a special kind of integration test, you can use the `get` and `post` methods in them.

Here's the performance test for `HomeController#dashboard` and `PostsController#create`:

```
require
'test_helper'
require
'rails/performance_test_help'

class

PostPerformanceTest < ActionDispatch::PerformanceTest

def

setup

# Application requires logged-in user
```

```
login_as(
:lifo
)

end


def

test_homepage

get
'/dashboard'

end


def

test_creating_new_post

post
'/posts'
,
:post

=> {
:body

=>
'lifo is fooling you'

}

end
end
```

You can find more details about the `get` and `post` methods in the **Testing Rails Applications** guide.

1.2.2 Model Example

Even though the performance tests are integration tests and hence closer to the request/response cycle by nature, you can still performance test pure model code.

Performance test for `Post` model:

```
require
'test_helper'
require
'rails/performance_test_help'

class

PostModelTest < ActionDispatch::PerformanceTest

def

test_creation

Post.create
:body

=>
'still fooling you'
,
:cost

=>
'100'

end


def

test_slow_method

# Using posts(:awesome) fixture

posts(
:awesome
```

```
).slow_method

end
end
```

1.3 Modes

Performance tests can be run in two modes: Benchmarking and Profiling.

1.3.1 Benchmarking

Benchmarking makes it easy to quickly gather a few metrics about each test run. By default, each test case is run **4 times** in benchmarking mode.

To run performance tests in benchmarking mode:

```
$ rake test:benchmark
```

1.3.2 Profiling

Profiling allows you to make an in-depth analysis of each of your tests by using an external profiler. Depending on your Ruby interpreter, this profiler can be native (Rubinius, JRuby) or not (MRI, which uses RubyProf). By default, each test case is run **once** in profiling mode.

To run performance tests in profiling mode:

```
$ rake test:profile
```

1.4 Metrics

Benchmarking and profiling run performance tests and give you multiple metrics. The availability of each metric is determined by the interpreter being used—none of them support all metrics—and by the mode in use. A brief description of each metric and their availability across interpreters/modes is given below.

1.4.1 Wall Time

Wall time measures the real world time elapsed during the test run. It is affected by any other processes concurrently running on the system.

1.4.2 Process Time

Process time measures the time taken by the process. It is unaffected by any other processes running concurrently on the same system. Hence, process time is likely to be constant for any given performance test, irrespective of the machine load.

1.4.3 CPU Time

Similar to process time, but leverages the more accurate CPU clock counter available on the Pentium and PowerPC platforms.

1.4.4 User Time

User time measures the amount of time the CPU spent in user-mode, i.e. within the process. This is not affected by other processes and by the time it possibly spends blocked.

1.4.5 Memory

Memory measures the amount of memory used for the performance test case.

1.4.6 Objects

Objects measures the number of objects allocated for the performance test case.

1.4.7 GC Runs

GC Runs measures the number of times GC was invoked for the performance test case.

1.4.8 GC Time

GC Time measures the amount of time spent in GC for the performance test case.

1.4.9 Metric Availability

1.4.9.1 Benchmarking

Interpreter	Wall Time	Process Time	CPU Time	User Time	Memory	Objects	GC Runs	GC Time
MRI	yes	yes	yes	no	yes	yes	yes	yes

REE	yes	yes	yes	no	yes	yes	yes	yes
Rubinius	yes	no	no	no	yes	yes	yes	yes
JRuby	yes	no	no	yes	yes	yes	yes	yes

1.4.9.2 Profiling

Interpreter	Wall Time	Process Time	CPU Time	User Time	Memory	Objects	GC Runs	GC Time
MRI	yes	yes	no	no	yes	yes	yes	yes
REE	yes	yes	no	no	yes	yes	yes	yes
Rubinius	yes	no	no	no	no	no	no	no
JRuby	yes	no	no	no	no	no	no	no

To profile under JRuby you'll need to run `export JRUBY_OPTS="-Xlaunch.inproc=false --profile.api"` **before** the performance tests.

1.5 Understanding the Output

Performance tests generate different outputs inside `tmp/performance` directory depending on their mode and metric.

1.5.1 Benchmarking

In benchmarking mode, performance tests generate two types of outputs.

1.5.1.1 Command Line

This is the primary form of output in benchmarking mode. Example:

```
BrowsingTest#test_homepage (31 ms warmup)
wall_time: 6 ms
memory: 437.27 KB
objects: 5,514
gc_runs: 0
gc_time: 19 ms
```

1.5.1.2 CSV Files

Performance test results are also appended to `.csv` files inside `tmp/performance`. For example, running the default `BrowsingTest#test_homepage` will generate following five files:

BrowsingTest#test_homepage_gc_runs.csv
BrowsingTest#test_homepage_gc_time.csv
BrowsingTest#test_homepage_memory.csv
BrowsingTest#test_homepage_objects.csv
BrowsingTest#test_homepage_wall_time.csv

As the results are appended to these files each time the performance tests are run in benchmarking mode, you can collect data over a period of time. This can be very helpful in analyzing the effects of code changes.

Sample output of `BrowsingTest#test_homepage_wall_time.csv`:

```
measurement,created_at,app,rails,ruby,platform
0.00738224999999992,2009-01-08T03:40:29Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00755874999999984,2009-01-08T03:46:18Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00762099999999993,2009-01-08T03:49:25Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00603075000000008,2009-01-08T04:03:29Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00619899999999995,2009-01-08T04:03:53Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00755449999999991,2009-01-08T04:04:55Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00595999999999997,2009-01-08T04:05:06Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00740450000000004,2009-01-09T03:54:47Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00603150000000008,2009-01-09T03:54:57Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
0.00771250000000012,2009-01-09T15:46:03Z,,3.0.0,ruby-1.8.7.249,x86_64-linux
```

1.5.2 Profiling

In profiling mode, performance tests can generate multiple types of outputs. The command line output is always presented but support for the others is dependent on the interpreter in use. A brief description of each type and their availability across interpreters is given below.

1.5.2.1 Command Line

This is a very basic form of output in profiling mode:

```
BrowsingTest#test_homepage (58 ms warmup)
process_time: 63 ms
memory: 832.13 KB
objects: 7,882
```

1.5.2.2 Flat

Flat output shows the metric—time, memory, etc—measure in each method. **Check Ruby-Prof documentation for a better explanation**.

1.5.2.3 Graph

Graph output shows the metric measure in each method, which methods call it and which methods it calls. **Check Ruby-Prof documentation for a better explanation**.

1.5.2.4 Tree

Tree output is profiling information in calltree format for use by **kcachegrind** and similar tools.

1.5.2.5 Output Availability

	Flat	Graph	Tree
MRI	yes	yes	yes
REE	yes	yes	yes
Rubinius	yes	yes	no
JRuby	yes	yes	no

1.6 Tuning Test Runs

Test runs can be tuned by setting the `profile_options` class variable on your test class.

```
require
'test_helper'
require
'rails/performance_test_help'

# Profiling results for each test method are written to tmp/performance.
class

BrowsingTest < ActionDispatch::PerformanceTest

self
.profile_options = {
:runs

=>
5
,

:metrics

=> [
:wall_time
,
:memory
] }


def

test_homepage

get
```

```
    '/'

  end
end
```

In this example, the test would run 5 times and measure wall time and memory. There are a few configurable options:

Option	Description	Default	Mode
`:runs`	Number of runs.	Benchmarking: 4, Profiling: 1	Both
`:output`	Directory to use when writing the results.	`tmp/performance`	Both
`:metrics`	Metrics to use.	See below.	Both
`:formats`	Formats to output to.	See below.	Profiling

Metrics and formats have different defaults depending on the interpreter in use.

Interpreter	Mode	Default metrics	Default formats
MRI/REE	Benchmarking	`[:wall_time, :memory, :objects, :gc_runs, :gc_time]`	N/A
	Profiling	`[:process_time, :memory, :objects]`	`[:flat, :graph_html, :call_tree, :call_stack]`
Rubinius	Benchmarking	`[:wall_time, :memory, :objects, :gc_runs, :gc_time]`	N/A
	Profiling	`[:wall_time]`	`[:flat, :graph]`
JRuby	Benchmarking	`[:wall_time, :user_time, :memory, :gc_runs, :gc_time]`	N/A
	Profiling	`[:wall_time]`	`[:flat, :graph]`

As you've probably noticed by now, metrics and formats are specified using a symbol array with each name **underscored.**

1.7 Performance Test Environment

Performance tests are run in the `test` environment. But running performance tests will set the following configuration parameters:

```
ActionController::Base.perform_caching = true
ActiveSupport::Dependencies.mechanism = :require
Rails.logger.level = ActiveSupport::BufferedLogger::INFO
```

As `ActionController::Base.perform_caching` is set to `true`, performance tests will behave much as they do in the `production` environment.

1.8 Installing GC-Patched MRI

To get the best from Rails' performance tests under MRI, you'll need to build a special Ruby binary with some super powers.

The recommended patches for each MRI version are:

Version	Patch
1.8.6	ruby186gc
1.8.7	ruby187gc
1.9.2 and above	gcdata

All of these can be found on **RVM's *patches* directory** under each specific interpreter version.

Concerning the installation itself, you can either do this easily by using **RVM** or you can build everything from source, which is a little bit harder.

1.8.1 Install Using RVM

The process of installing a patched Ruby interpreter is very easy if you let RVM do the hard work. All of the following RVM commands will provide you with a patched Ruby interpreter:

```
$ rvm install 1.9.2-p180 --patch gcdata
$ rvm install 1.8.7 --patch ruby187gc
$ rvm install 1.9.2-p180 --patch ~/Downloads/downloaded_gcdata_patch.patch
```

You can even keep your regular interpreter by assigning a name to the patched one:

```
$ rvm install 1.9.2-p180 --patch gcdata --name gcdata
$ rvm use 1.9.2-p180 # your regular ruby
$ rvm use 1.9.2-p180-gcdata # your patched ruby
```

And it's done! You have installed a patched Ruby interpreter.

1.8.2 Install From Source

This process is a bit more complicated, but straightforward nonetheless. If you've never compiled a Ruby binary before, follow these steps to build a Ruby binary inside your home directory.

1.8.2.1 Download and Extract

```
$ mkdir rubygc
$ wget <the version you want from ftp://ftp.ruby-lang.org/pub/ruby>
$ tar -xzvf <ruby-version.tar.gz>
$ cd <ruby-version>
```

1.8.2.2 Apply the Patch

```
$ curl http://github.com/wayneeseguin/rvm/raw/master/patches/ruby/1.9.2/p180/gcdata.patch | patch -p0 #
$ curl http://github.com/wayneeseguin/rvm/raw/master/patches/ruby/1.8.7/ruby187gc.patch | patch -p0 # if
```

1.8.2.3 Configure and Install

The following will install Ruby in your home directory's `/rubygc` directory. Make sure to replace `<homedir>` with a full patch to your actual home directory.

```
$ ./configure --prefix=/<homedir>/rubygc
$ make && make install
```

1.8.2.4 Prepare Aliases

For convenience, add the following lines in your `~/.profile`:

```
alias gcruby='~/rubygc/bin/ruby'
alias gcrake='~/rubygc/bin/rake'
alias gcgem='~/rubygc/bin/gem'
alias gcirb='~/rubygc/bin/irb'
alias gcrails='~/rubygc/bin/rails'
```

Don't forget to use your aliases from now on.

1.8.2.5 Install RubyGems (1.8 only!)

Download **RubyGems** and install it from source. Rubygem's README file should have necessary installation instructions. Please note that this step isn't necessary if you've installed Ruby 1.9 and above.

1.9 Using Ruby-Prof on MRI and REE

Add Ruby-Prof to your applications' Gemfile if you want to benchmark/profile under MRI or REE:

```
gem
'ruby-prof'
,
:git

=>
'git://github.com/wycats/ruby-prof.git'
```

Now run `bundle install` and you're ready to go.

2 Command Line Tools

Writing performance test cases could be an overkill when you are looking for one time tests. Rails ships with two command line tools that enable quick and dirty performance testing:

2.1 `benchmarker`

Usage:

```
Usage: rails benchmarker 'Ruby.code' 'Ruby.more_code' ... [OPTS]

-r, --runs N                     Number of runs.

Default: 4

-o, --output PATH                Directory to use when writing the results.

Default: tmp/performance

-m, --metrics a,b,c              Metrics to use.

Default: wall_time,memory,objects,gc_runs,gc_time
```

Example:

```
$ rails benchmarker 'Item.all' 'CouchItem.all' --runs 3 --metrics wall_time,memory
```

2.2 `profiler`

Usage:

```
Usage: rails profiler 'Ruby.code' 'Ruby.more_code' ... [OPTS]

-r, --runs N                     Number of runs.

Default: 1

-o, --output PATH                Directory to use when writing the results.

Default: tmp/performance

--metrics a,b,c              Metrics to use.

Default: process_time,memory,objects

-m, --formats x,y,z              Formats to output to.

Default: flat,graph_html,call_tree
```

Example:

```
$ rails profiler 'Item.all' 'CouchItem.all' --runs 2 --metrics process_time --formats flat
```

Metrics and formats vary from interpreter to interpreter. Pass `--help` to each tool to see the defaults for your interpreter.

3 Helper Methods

Rails provides various helper methods inside Active Record, Action Controller and Action View to measure the time taken by a given piece of code. The method is called `benchmark()` in all the three components.

3.1 Model

```
Project.benchmark(
"Creating project"
)
do
```

```
project = Project.create(
"name"

=>
"stuff"
)

project.create_manager(
"name"

=>
"David"
)

project.milestones << Milestone.all
end
```

This benchmarks the code enclosed in the `Project.benchmark("Creating project") do...end` block and prints the result to the log file:

```
Creating project (
185
.3ms)
```

Please refer to the **API docs** for additional options to `benchmark()`

3.2 Controller

Similarly, you could use this helper method inside **controllers**

```
def

process_projects

self
.
class
.benchmark(
"Processing projects"
)
do

Project.process(params[
:project_ids
])

Project.update_cached_projects

end
end
```

`benchmark` is a class method inside controllers

3.3 View

And in **views**:

```
<%

benchmark(
"Showing projects partial"
)
do

%>

<%=

render
@projects

%>
<%

end
```

```
%>
```

4 Request Logging

Rails log files contain very useful information about the time taken to serve each request. Here's a typical log file entry:

```
Processing ItemsController#index (for 127.0.0.1 at 2009-01-08 03:06:39) [GET]
Rendering template within layouts/items
Rendering items/index
Completed in 5ms (View: 2, DB: 0) | 200 OK [http://0.0.0.0/items]
```

For this section, we're only interested in the last line:

```
Completed in 5ms (View: 2, DB: 0) | 200 OK [http://0.0.0.0/items]
```

This data is fairly straightforward to understand. Rails uses millisecond(ms) as the metric to measure the time taken. The complete request spent 5 ms inside Rails, out of which 2 ms were spent rendering views and none was spent communication with the database. It's safe to assume that the remaining 3 ms were spent inside the controller.

Michael Koziarski has an **interesting blog post** explaining the importance of using milliseconds as the metric.

5 Useful Links

5.1 Rails Plugins and Gems

- **Rails Analyzer**
- **Palmist**
- **Rails Footnotes**
- **Query Reviewer**

5.2 Generic Tools

- **httperf**
- **ab**
- **JMeter**
- **kcachegrind**

5.3 Tutorials and Documentation

- **ruby-prof API Documentation**
- **Request Profiling Railscast** – Outdated, but useful for understanding call graphs

6 Commercial Products

Rails has been lucky to have a few companies dedicated to Rails-specific performance tools. A couple of those are:

- **New Relic**
- **Scout**

Feedback

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

If you see any typos or factual errors you are confident to patch, please clone **docrails** and push the change yourself. That branch of Rails has public write access. Commits are still reviewed, but that happens after you've submitted your contribution. **docrails** is cross-merged with master periodically.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. 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**.