## Chapters

Rails

- Ruby on Rails
- Guides Index

# Benchmarking and Profiling Rails

This guide covers the benchmarking and profiling tactics/tools of Rails and Ruby in general. By referring to this guide, you will be able to:

- Understand the various types of benchmarking and profiling metrics
- Generate performance/benchmarking tests
- Use GC patched Ruby binary to measure memory usage and object allocation
- Understand the information provided by Rails inside the log files
- Learn about various tools facilitating benchmarking and profiling

## 1. Why Benchmark and Profile ?

Benchmarking and Profiling 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 plesant browsing experience to the end users and cutting cost of unnecessary hardwares is important for any web application.

### 1.1. What is the difference between benchmarking and profiling ?

Benchmarking is the process of finding out if a piece of code is slow or not. Whereas profiling is the process of finding out what exactly is slowing down that piece of code.

## 2. Using and understanding the log files

Rails logs files containt basic but very useful information about the time taken to serve every request. A typical log entry looks something like :

```
Processing ItemsController#index (for 127.0.0.1 at 2008-10-17 00:08:18) [GET]
  Session ID: BAh7BiIKZmxhc2hJQzonQWN0aHsABjoKQHVzZWR7AA==--83cff4fe0a897074a65335
  Parameters: {"action"=>"index", "controller"=>"items"}
Rendering template within layouts/items
Rendering items/index
Completed in 5ms (View: 2, DB: 0) | 200 OK [http://localhost/items]
```

For this section, we're only interested in the last line from that log entry:

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

This data is fairly straight forward to understand. Rails uses millisecond(ms) as the metric to measures 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.

## 3. Helper methods

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

```
Project.benchmark("Creating project") do
  project = Project.create("name" => "stuff")
  project.create_manager("name" => "David")
  project.milestones << Milestone.find(:all)
```

```
end
```

The above code benchmarks the multiple statments enclosed inside `Project.benchmark("Creating project") do..end` block and prints the results inside log files. The statement inside log files will look like:

```
Creating projectem (185.3ms)
```

Please refer to API docs for optional options to `benchmark()`

Similarly, you could use this helper method inside controllers ( Note that it's a class method here ):

```
def process_projects
  self.class.benchmark("Processing projects") do
    Project.process(params[:project_ids])
    Project.update_cached_projects
  end
end
```

and views:

```
<% benchmark("Showing projects partial") do %>
  <%= render :partial => @projects %>
<% end %>
```

## 4. Performance Test Cases

Rails provides a very easy to write performance test cases, which look just like the regular integration tests.

If you have a look at `test/performance/browsing_test.rb` in a newly created Rails application:

```
require 'test_helper'
require 'performance_test_help'

# Profiling results for each test method are written to tmp/performance.
class BrowsingTest < ActionController::PerformanceTest
  def test_homepage
    get '/'
  end
end
```

This is an automatically generated example performance test file, for testing performance of homepage(/) of the application.

### 4.1. Modes

#### 4.1.1. Benchmarking

#### 4.1.2. Profiling

### 4.2. Metrics

#### 4.2.1. Process Time

CPU Cycles.

#### 4.2.2. Memory

Memory taken.

#### 4.2.3. Objects

Objects allocated.

#### 4.2.4. GC Runs

Number of times the Ruby GC was run.

#### 4.2.5. GC Time

Time spent running the Ruby GC.

### 4.3. Preparing Ruby and Ruby-prof

Before we go ahead, Rails performance testing requires you to build a special Ruby binary with some super powers - GC patch for measuring GC Runs/Time. This process is very straight forward. If you've never compiled a Ruby binary before, you can follow the following steps to build a ruby binary inside your home directory:

#### 4.3.1. Compile

```
[lifo@null ~]$ mkdir rubygc
[lifo@null ~]$ wget ftp://ftp.ruby-lang.org/pub/ruby/1.8/ruby-1.8.6-p111.tar.gz
[lifo@null ~]$ tar -xzvf ruby-1.8.6-p111.tar.gz
[lifo@null ~]$ cd ruby-1.8.6-p111
[lifo@null ruby-1.8.6-p111]$ curl http://rubyforge.org/tracker/download.php/1814/7062/17676/3291/ruby186gc.patch | patch -p0
[lifo@null ruby-1.8.6-p111]$ ./configure --prefix=/Users/lifo/rubygc
[lifo@null ruby-1.8.6-p111]$ make && make install
```

#### 4.3.2. Prepare aliases

Add the following lines in your ~/.profile for convenience:

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

#### 4.3.3. Install rubygems and some basic gems

```
[lifo@null ~]$ wget http://rubyforge.org/frs/download.php/38646/rubygems-1.2.0.tgz
[lifo@null ~]$ tar -xzvf rubygems-1.2.0.tgz
[lifo@null ~]$ cd rubygems-1.2.0
[lifo@null rubygems-1.2.0]$ gcruby setup.rb
[lifo@null rubygems-1.2.0]$ cd ~
[lifo@null ~]$ gcgem install rake
[lifo@null ~]$ gcgem install rails
```

#### 4.3.4. Install MySQL gem

```
[lifo@null ~]$ gcgem install mysql
```

If this fails, you can try to install it manually:

```
[lifo@null ~]$ cd /Users/lifo/rubygc/lib/ruby/gems/1.8/gems/mysql-2.7/
[lifo@null mysql-2.7]$ gcruby extconf.rb --with-mysql-config
[lifo@null mysql-2.7]$ make && make install
```

### 4.4. Installing Jeremy Kemper's ruby-prof

We also need to install Jeremy's ruby-prof gem using our newly built ruby:

```
[lifo@null ~]$ git clone git://github.com/jeremy/ruby-prof.git
[lifo@null ~]$ cd ruby-prof/
[lifo@null ruby-prof (master)]$ gcrake gem
[lifo@null ruby-prof (master)]$ gcgem install pkg/ruby-prof-0.6.1.gem
```

### 4.5. Generating performance test

Rails provides a simple generator for creating new performance tests:

```
[lifo@null application (master)]$ script/generate performance_test homepage
```

This will generate `test/performance/homepage_test.rb`:

```ruby
require 'test_helper'
require 'performance_test_help'

class HomepageTest < ActionController::PerformanceTest
  # Replace this with your real tests.
  def test_homepage
    get '/'
  end
end
```

Which you can modify to suit your needs.

### 4.6. Running tests

## 5. Understanding Performance Tests Outputs

### 5.1. Our First Performance Test

So how do we profile a request.

One of the things that is important to us is how long it takes to render the home page - so let's make a request to the home page. Once the request is complete, the results will be outputted in the terminal.

In the terminal run

```
[User profiling_tester]$ gcruby tests/performance/homepage.rb
```

After the tests runs for a few seconds you should see something like this.

```
HomepageTest#test_homepage (19 ms warmup)
        process_time: 26 ms
              memory: 298.79 KB
             objects: 1917

Finished in 2.207428 seconds.
```

Simple but efficient.

- Process Time refers to amount of time necessary to complete the action.
- memory is the amount of information loaded into memory
- object ??? #TODO find a good definition. Is it the amount of objects put into a ruby heap for this process?

In addition we also gain three types of itemized log files for each of these outputs. They can be found in your tmp directory of your application.

**The Three types are**

- Flat File - A simple text file with the data laid out in a grid

- Graphical File - A html colored coded version of the simple text file with hyperlinks between the various methods. Most useful is the bolding of the main processes for each portion of the action.
- Tree File - A file output that can be use in conjunction with KCachegrind to visualize the process

Note

KCachegrind is Linux only. For Mac this means you have to do a full KDE install to have it working in your OS. Which is over 3 gigs in size. For windows there is clone called wincachegrind but it is no longer actively being developed.

Below are examples for Flat Files and Graphical Files

### 5.2. Flat Files

Example: Flat File Output Processing Time

Thread ID: 2279160 Total: 0.026097

```
%self     total     self     wait    child    calls  name
 6.41      0.06     0.04     0.00     0.02      571  Kernel#===
 3.17      0.00     0.00     0.00     0.00      172  Hash#[]
 2.42      0.00     0.00     0.00     0.00       13  MonitorMixin#mon_exit
 2.05      0.00     0.00     0.00     0.00       15  Array#each
 1.56      0.00     0.00     0.00     0.00        6  Logger#add
 1.55      0.00     0.00     0.00     0.00       13  MonitorMixin#mon_enter
 1.36      0.03     0.00     0.00     0.03        1  ActionController::Integration::Session#process
 1.31      0.00     0.00     0.00     0.00       13  MonitorMixin#mon_release
 1.15      0.00     0.00     0.00     0.00        8  MonitorMixin#synchronize-1
 1.09      0.00     0.00     0.00     0.00       23  Class#new
 1.03      0.01     0.00     0.00     0.01        5  MonitorMixin#synchronize
 0.89      0.00     0.00     0.00     0.00       74  Hash#default
 0.89      0.00     0.00     0.00     0.00        6  Hodel3000CompliantLogger#format_message
 0.80      0.00     0.00     0.00     0.00        9  c
 0.80      0.00     0.00     0.00     0.00       11  ActiveRecord::ConnectionAdapters::ConnectionHandler#retrieve_connection_pool
 0.79      0.01     0.00     0.00     0.01        1  ActionController::Benchmarking#perform_action_without_rescue
 0.18      0.00     0.00     0.00     0.00       17  <Class::Object>#allocate
```

So what do these columns tell us:

- %self - The percentage of time spent processing the method. This is derived from self_time/total_time
- total - The time spent in this method and its children.
- self - The time spent in this method.
- wait - Time processed was queued
- child - The time spent in this method's children.
- calls - The number of times this method was called.
- name - The name of the method.

Name can be displayed three seperate ways: **#toplevel - The root method that calls all other methods** MyObject#method - Example Hash#each, The class Hash is calling the method each * <Object:MyObject>#test - The <> characters indicate a singleton method on a singleton class. Example <Class::Object>#allocate

Methods are sorted based on %self. Hence the ones taking the most time and resources will be at the top.

So for Array#each which is calling each on the class array. We find that it processing time is 2% of the total and was called 15 times. The rest of the information is 0.00 because the process is so fast it isn't recording times less then 100 ms.

Example: Flat File Memory Output

Thread ID: 2279160 Total: 509.724609

```
%self     total     self     wait    child    calls  name
 4.62     23.57    23.57     0.00     0.00       34  String#split
 3.95     57.66    20.13     0.00    37.53        3  <Module::YAML>#quick_emit
 2.82     23.70    14.35     0.00     9.34        2  <Module::YAML>#quick_emit-1
 1.37     35.87     6.96     0.00    28.91        1  ActionView::Helpers::FormTagHelper#form_tag
 1.35      7.69     6.88     0.00     0.81        1  ActionController::HttpAuthentication::Basic::ControllerMethods#authenticate_with_http_basic
 1.06      6.09     5.42     0.00     0.67       90  String#gsub
 1.01      5.13     5.13     0.00     0.00       27  Array#-
```

Very similar to the processing time format. The main difference here is that instead of calculating time we are now concerned with the amount of KB put into memory **(or is it strictly into the heap) can I get clarification on this minor point?**

So for <Module::YAML>#quick_emit which is singleton method on the class YAML it uses 57.66 KB in total, 23.57 through its own actions, 6.69 from actions it calls itself and that it was called twice.

Example: Flat File Objects

Thread ID: 2279160 Total: 6537.000000

```
%self     total     self     wait    child    calls  name
15.16   1096.00   991.00     0.00   105.00       66  Hash#each
 5.25    343.00   343.00     0.00     0.00        4  Mysql::Result#each_hash
 4.74   2203.00   310.00     0.00  1893.00       42  Array#each
 3.75   4529.00   245.00     0.00  4284.00        1  ActionView::Base::CompiledTemplates#_run_erb_47app47views47layouts47application46html46erb
 2.00    136.00   131.00     0.00     5.00       90  String#gsub
 1.73    113.00   113.00     0.00     0.00       34  String#split
 1.44    111.00    94.00     0.00    17.00       31  Array#each-1
```

#TODO Find correct terminology for how to describe what this is exactly profiling as in are there really 2203 array objects or 2203 pointers to array objects?.

### 5.3. Graph Files

While the information gleamed from flat files is very useful we still don't know which processes each method is calling. We only know how many. This is not true for a graph file. Below is a text representation of a graph file. The actual graph file is an html entity and an example of which can be found Here

#TODO (Handily the graph file has links both between it many processes and to the files that actually contain them for debugging. )

Example: Graph File

Thread ID: 21277412

```
  %total   %self     total      self    children              calls   Name
/____________________________________________________________________________/
100.00%   0.00%     8.77      0.00      8.77                   1     #toplevel*
                    8.77      0.00      8.77                 1/1     Object#run_primes
/____________________________________________________________________________/
                    8.77      0.00      8.77                 1/1     #toplevel
100.00%   0.00%     8.77      0.00      8.77                   1     Object#run_primes*
                    0.02      0.00      0.02                 1/1     Object#make_random_array
                    2.09      0.00      2.09                 1/1     Object#find_largest
                    6.66      0.00      6.66                 1/1     Object#find_primes
/____________________________________________________________________________/
                    0.02      0.02      0.00                 1/1     Object#make_random_array
0.18%     0.18%     0.02      0.02      0.00                   1     Array#each_index
                    0.00      0.00      0.00             500/500     Kernel.rand
                    0.00      0.00      0.00             500/501     Array#[]=
/____________________________________________________________________________/
```

As you can see the calls have been separated into slices, no longer is the order determined by process time but instead from hierarchy. Each slice profiles a primary entry, with the primary entry's parents being shown above itself and it's children found below. A primary entry can be ascertained by it having values in the %total and %self columns. Here the main entry here have been bolded for connivence.

So if we look at the last slice. The primary entry would be Array#each_index. It takes 0.18% of the total process time and it is only called once. It is called from Object#make_random_array which is only called once. It's children are Kernal.rand which is called by it all 500 its times that it was call in this action and Arry#[]= which was called 500 times by Array#each_index and once by some other entry.

### 5.4. Tree Files

It's pointless trying to represent a tree file textually so here's a few pretty pictures of it's usefulness

KCachegrind Graph

Graph created by KCachegrind

KCachegrind List

List created by KCachegrind

#TODO Add a bit more information to this.

## 6. Getting to the Point of all of this

Now I know all of this is a bit dry and academic. But it's a very powerful tool when you know how to leverage it properly. Which we are going to take a look at in our next section

## 7. Real Life Example

### 7.1. The setup

So I have been building this application for the last month and feel pretty good about the ruby code. I'm readying it for beta testers when I discover to my shock that with less then twenty people it starts to crash. It's a pretty simple Ecommerce site so I'm very confused by what I'm seeing. On running looking through my log files I find to my shock that the lowest time for a page run is running around 240 ms. My database finds aren't the problems so I'm lost as to what is happening to cause all this. Lets run a benchmark.

```
class HomepageTest < ActionController::PerformanceTest
  # Replace this with your real tests.
  def test_homepage
    get '/'
  end
end
```

Example: Output

```
HomepageTest#test_homepage (115 ms warmup)
        process_time: 591 ms
        memory: 3052.90 KB
        objects: 59471
```

Obviously something is very very wrong here. 3052.90 Kb to load my minimal homepage. For Comparison for another site running well I get this for my homepage test.

Example: Default

```
HomepageTest#test_homepage (19 ms warmup)
        process_time: 26 ms
              memory: 298.79 KB
             objects: 1917
```

that over a factor of ten difference. Lets look at our flat process time file to see if anything pops out at us.

Example: Process time

```
20.73      0.39     0.12     0.00     0.27      420  Pathname#cleanpath_aggressive
17.07      0.14     0.10     0.00     0.04     3186  Pathname#chop_basename
 6.47      0.06     0.04     0.00     0.02     6571  Kernel#===
 5.04      0.06     0.03     0.00     0.03      840  Pathname#initialize
 5.03      0.05     0.03     0.00     0.02        4  ERB::Compiler::ExplicitScanner#scan
 4.51      0.03     0.03     0.00     0.00     9504  String#==
 2.94      0.46     0.02     0.00     0.44     1393  String#gsub
 2.66      0.09     0.02     0.00     0.07      480  Array#each
 2.46      0.01     0.01     0.00     0.00     3606  Regexp#to_s
```

Yes indeed we seem to have found the problem. Pathname#cleanpath_aggressive is taking nearly a quarter our process time and Pathname#chop_basename another 17%. From here I do a few more benchmarks to make sure that these processes are slowing down the other pages. They are so now I know what I must do. **If we can get rid of or shorten these processes we can make our pages run much quicker**.

Now both of these are main ruby processes so are goal right now is to find out what other process is calling them. Glancing at our Graph file I see that #cleanpath is calling #cleanpath_aggressive. #cleanpath is being called by String#gsub and from there some html template errors. But my page seems to be rendering fine. why would it be calling template errors. I'm decide to check my object flat file to see if I can find any more information.

Example: Objects Created

```
20.74  34800.00 12324.00     0.00 22476.00      420  Pathname#cleanpath_aggressive
16.79  18696.00  9978.00     0.00  8718.00     3186  Pathname#chop_basename
11.47  13197.00  6813.00     0.00  6384.00      480  Array#each
 8.51  41964.00  5059.00     0.00 36905.00     1386  String#gsub
 6.07   3606.00  3606.00     0.00     0.00     3606  Regexp#to_s
```

nope nothing new here. Lets look at memory usage

Example: Memory Consuption

```
 40.17   1706.80  1223.70     0.00   483.10     3186  Pathname#chop_basename
 14.92    454.47   454.47     0.00     0.00     3606  Regexp#to_s
  7.09   2381.36   215.99     0.00  2165.37     1386  String#gsub
  5.08    231.19   154.73     0.00    76.46      420  Pathname#prepend_prefix
  2.34     71.35    71.35     0.00     0.00     1265  String#initialize_copy
```

Ok so it seems Regexp#to_s is the second costliest process. At this point I try to figure out what could be calling a regular expression cause I very rarely use them. Going over my standard layout I discover at the top.

```
<%if request.env["HTTP_USER_AGENT"].match(/Opera/)%>
<%= stylesheet_link_tag "opera" %>
<% end %>
```

That's wrong. I mistakenly am using a search function for a simple compare function. Lets fix that.

```
<%if request.env["HTTP_USER_AGENT"] =~ /Opera/%>
<%= stylesheet_link_tag "opera" %>
<% end %>
```

I'll now try my test again.

```
process_time: 75 ms
              memory: 519.95 KB
             objects: 6537
```

Much better. The problem has been solved. Now I should have realized earlier due to the String#gsub that my problem had to be with reqexp serch function but such knowledge comes with time. Looking through the mass output data is a skill.

## 8. Get Yourself a Game Plan

You end up dealing with a large amount of data whenever you profile an application. It's crucial to use a rigorous approach to analyzing your application's performance else fail miserably in a vortex of numbers. This leads us to -

### 8.1. The Analysis Process

I’m going to give an example methodology for conducting your benchmarking and profiling on an application. It is based on your typical scientific method.

For something as complex as Benchmarking you need to take any methodology with a grain of salt but there are some basic strictures that you can depend on.

Formulate a question you need to answer which is simple, tests the smallest measurable thing possible, and is exact. This is typically the hardest part of the experiment. From there some steps that you should follow are.

- Develop a set of variables and processes to measure in order to answer this question!
- Profile based on the question and variables. Key problems to avoid when designing this experiment are:
    - Confounding: Test one thing at a time, keep everything the same so you don't poison the data with uncontrolled processes.
    - Cross Contamination: Make sure that runs from one test do not harm the other tests.
    - Steady States: If you're testing long running process. You must take the ramp up time and performance hit into your initial measurements.
    - Sampling Error: Data should perform have a steady variance or range. If you get wild swings or sudden spikes, etc. then you must either account for the reason why or you have a sampling error.
    - Measurement Error: Aka Human error, always go through your calculations at least twice to make sure there are no mathematical errors. .
- Do a small run of the experiment to verify the design.
- Use the small run to determine a proper sample size.
- Run the test.
- Perform the analysis on the results and determine where to go from there.

Note: Even though we are using the typical scientific method; developing a hypothesis is not always useful in terms of profiling.

## 9. Other Profiling Tools

There are a lot of great profiling tools out there. Some free, some not so free. This is a sort list detailing some of them.

### 9.1. httperf

http://www.hpl.hp.com/research/linux/httperf/

A necessary tool in your arsenal. Very useful for load testing your website.

#TODO write and link to a short article on how to use httperf. Anybody have a good tutorial availble.

### 9.2. Rails Analyzer

The Rails Analyzer project contains a collection of tools for Rails. It's open source and pretty speedy. It's not being actively worked on but is still contains some very useful tools.

- The Production Log Analyzer examines Rails log files and gives back a report. It also includes action_grep which will give you all log results for a particular action.
- The Action Profiler similar to Ruby-Prof profiler.
- rails_stat which gives a live counter of requests per second of a running Rails app.
- The SQL Dependency Grapher allows you to visualize the frequency of table dependencies in a Rails application.

Their project homepage can be found at http://rails-analyzer.rubyforge.org/

The one major caveat is that it needs your log to be in a different format from how rails sets it up specifically SyslogLogger.

#### 9.2.1. SyslogLogger

SyslogLogger is a Logger work-alike that logs via syslog instead of to a file. You can add SyslogLogger to your Rails production environment to aggregate logs between multiple machines.

More information can be found out at http://rails-analyzer.rubyforge.org/hacks/classes/SyslogLogger.html

If you don't have access to your machines root system or just want something a bit easier to implement there is also a module developed by Geoffrey Grosenbach

#### 9.2.2. A Hodel 3000 Compliant Logger for the Rest of Us

Directions taken from link to module file

Just put the module in your lib directory and add this to your environment.rb in it's config portion.

```
require 'hodel_3000_compliant_logger'
config.logger = Hodel3000CompliantLogger.new(config.log_path)
```

It's that simple. Your log output on restart should look like this.

Example: Hodel 3000 Example

```
Jul 15 11:45:43 matthew-bergmans-macbook-pro-15 rails[16207]:
Parameters: {"action"=>"shipping", "controller"=>"checkout"}
Jul 15 11:45:43 matthew-bergmans-macbook-pro-15 rails[16207]:
[4;36;1mBook Columns (0.003155)[0m   [0;1mSHOW FIELDS FROM `books`[0m
Jul 15 11:45:43 matthew-bergmans-macbook-pro-15 rails[16207]:
[4;35;1mBook Load (0.000881)[0m   [0mSELECT * FROM `books` WHERE (`books`.`id` = 1 AND (`books`.`sold` = 1)) [0m
Jul 15 11:45:43 matthew-bergmans-macbook-pro-15 rails[16207]:
[4;36;1mShippingAddress Columns (0.002683)[0m   [0;1mSHOW FIELDS FROM `shipping_addresses`[0m
Jul 15 11:45:43 matthew-bergmans-macbook-pro-15 rails[16207]:
[4;35;1mBook Load (0.000362)[0m   [0mSELECT ounces FROM `books` WHERE (`books`.`id` = 1) [0m
Jul 15 11:45:43 matthew-bergmans-macbook-pro-15 rails[16207]:
Rendering template within layouts/application
Jul 15 11:45:43 matthew-bergmans-macbook-pro-15 rails[16207]:
Rendering checkout/shipping
Jul 15 11:45:43 matthew-bergmans-macbook-pro-15 rails[16207]:
[4;36;1mBook Load (0.000548)[0m   [0;1mSELECT * FROM `books`
WHERE (sold = 0) LIMIT 3[0m
Jul 15 11:45:43 matthew-bergmans-macbook-pro-15 rails[16207]:
[4;35;1mAuthor Columns (0.002571)[0m   [0mSHOW FIELDS FROM `authors`[0m
Jul 15 11:45:43 matthew-bergmans-macbook-pro-15 rails[16207]:
[4;36;1mAuthor Load (0.000811)[0m   [0;1mSELECT * FROM `authors` WHERE (`authors`.`id` = 1) [0m
Jul 15 11:45:43 matthew-bergmans-macbook-pro-15 rails[16207]:
Rendered store/_new_books (0.01358)
Jul 15 11:45:43 matthew-bergmans-macbook-pro-15 rails[16207]:
Completed in 0.37297 (2 reqs/sec) | Rendering: 0.02971 (7%) | DB: 0.01697 (4%) | 200 OK [https://secure.jeffbooks/checkout/shipping]
```

### 9.3. Palmist

An open source mysql query analyzer. Full featured and easy to work with. Also requires Hodel 3000 http://www.flyingmachinestudios.com/projects/

### 9.4. New Relic

http://www.newrelic.com/

Pretty nifty performance tools, pricey though. They do have a basic free service both for when in development and when you put your application into production. Very simple installation and signup.

#TODO more in-depth without being like an advertisement.

#### 9.4.1. Manage

Like new relic a production monitoring tool.

## 10. Changelog

Lighthouse ticket

- October 17, 2008: First revision by Pratik
- September 6, 2008: Initial version by Matthew Bergman <MzbPhoto@gmail.com>