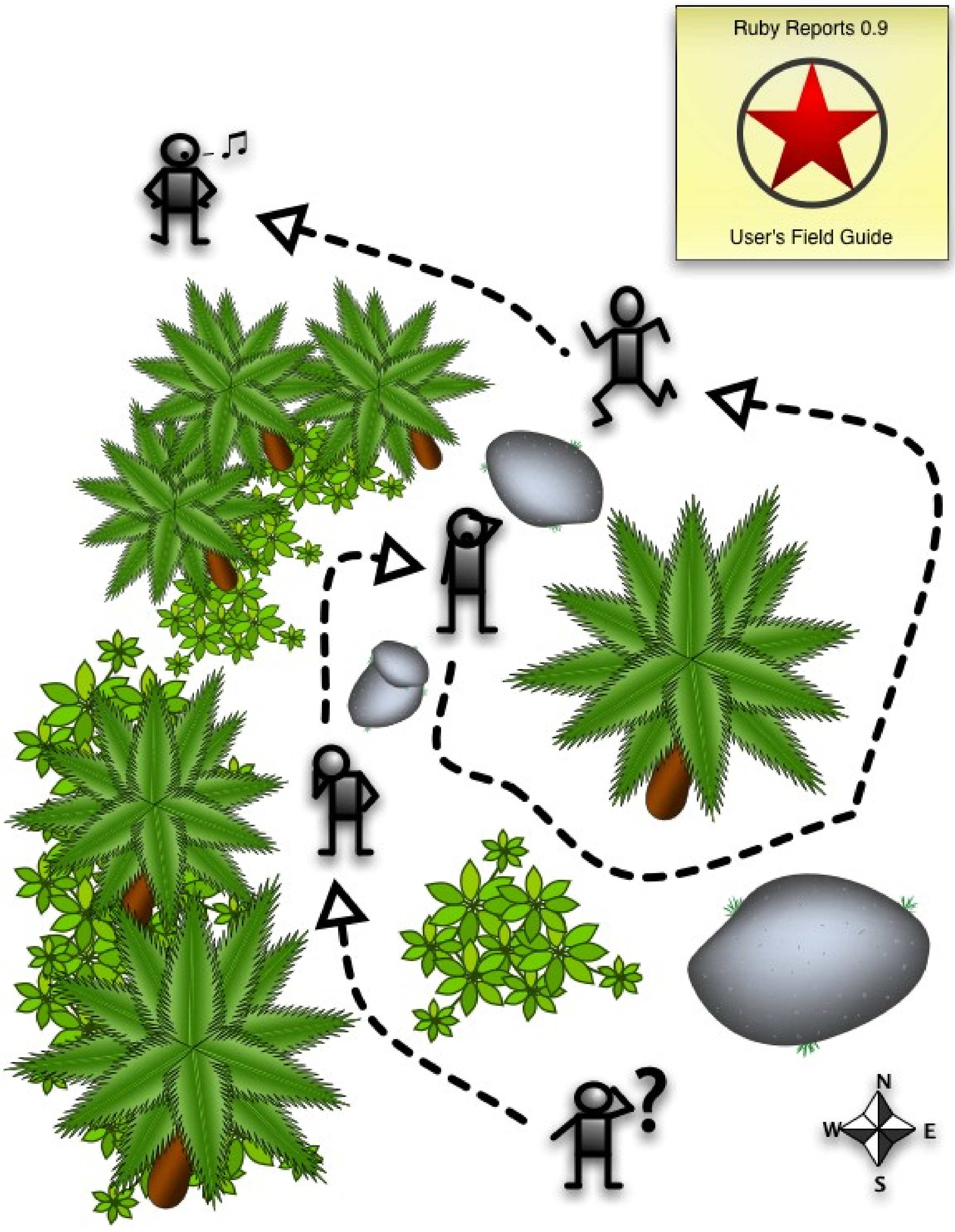

A guide Ruport's dev branch for kids who like to run with scissors.
(edit date: 2007.04.04)

First Steps

1. Build the latest development snapshot[1]

```
gem install ruport -y --source http://rubyreports.org
```

2. install mocha

```
gem install mocha
```

3. get ruport from trunk

```
svn co http://stonecode.svnrepository.com/svn/ruport/ruport/trunk ruport
```

4. make sure tests pass

```
rake test
```

5. Install Ruport Again[2]

```
rake gem

cd pkg

gem install ruport
```

Problems?

```
http://list.rubyreports.org
```

---

1. This grabs most of your dependencies and should get you pretty close to what trunk is running.
2. This installs the absolute latest code, which is what we'd like folks to run for our dev branch.

Five Ways to not get lost

1. Use the mailing list

(http://list.rubyreports.org)

2. Use #ruport on Freenode

(Gregory - <sandal> | Michael - <mikem836>)

3. Keep an eye on the Trac timeline

http://stonecode.svnrepository.com/ruport/trac.cgi/timeline

(RSS feed available, see bottom of page at the above link)

4. Watch the Blog

http://blog.rubyreports.org/

5. Use edge docs from trunk

```
rake rdoc
```

`(or for latest released snapshot: http://api.rubyreports.org/edge)`

Users of Ruport 0.8, Beware!

Dev has pulled away from 0.8.0 by >200 changesets.
Here are some highlights

- `Format::Plugin` is now `Formatter`
- `options` and `layout` now combined just as `options`
- `MyRenderer.add_format(self, :pdf)` becomes `renders :pdf, :for => MyRenderer`
- `Renderer::Helpers` now default, and part of `Renderer` class
- New Renderers: `Renderer::Row, Renderer::Group, Renderer::Grouping`
- There are lots of new helpers in `Formatter::PDF`
- There are formatting hooks for `Report` via `renders_with`
- your `Formatter` objects can now implement multiple formats
- New data structures: `Data::Group, Data::Grouping`
- Removed Classes/Modules: `Groupable, Taggable, Renderer::Graph, Format::SVG, Format::XML, Renderer::TableHelpers`
- Rope has a whole bunch of new Rake tasks,
- including an **acts_as_reportable** installer and a renderer generator
- Rope now supports hooks for describing your directory layout and provides shortcuts like **data_dir/** , **output_dir/** etc.
- There are IO hooks in the formatting system via `:io`
- Many of the 'old features' of Ruport will be revived in **ruport-util** (as of 2007.04.04, this includes invoice, graph, and report_manager)
- CSV input and output has many improvements by exposing more hooks to FasterCSV

Whatchu' Talkin' Bout, Willis?

A random smattering of Ruport 0.9 examples

== Grouping, and using acts_as_reportable standalone ==

(The full source for this project, along with a MySQL dump is available via SVN at: http://stonecode.svnrepository.com/svn/ruport_shards/tattle_aar/)

AAR Config (in lib/init.rb):

```
require "active_record"
require "vendor/plugins/acts_as_reportable/lib/acts_as_reportable"
ActiveRecord::Base.send :include, Ruport::Reportable
require "data/models"
ActiveRecord::Base.establish_connection(
    :adapter  => 'mysql',
    :host     => 'localhost',
    :username => 'root',
    :database => 'tattle')
```

Report File:

```
require "lib/init"
class Platforms < Ruport::Report
  renders_with Ruport::Renderer::Grouping

  def generate

    table = Report.report_table(:all,
      :only       => %w[host_os rubygems_version ruby_version user_key],
      :conditions => "user_key is not null and user_key <> ''",
      :group      => "host_os, rubygems_version, ruby_version, user_key")
    grouping = Grouping(table, :by => "host_os")
    rubygems_versions = Table(%w[platform rubygems_version count])
    grouping.each do |name,group|
      Grouping(group, :by => "rubygems_version").each do |vname,group|
        rubygems_versions << { "platform"         => name,
                               "rubygems_version" => vname,
                               "count"            => group.length }
      end
    end
    sorted_table = rubygems_versions.sort_rows_by { |r| -r.count }
    g = Grouping(sorted_table, :by => "platform")
  end
end
if __FILE__ == $0
  Platforms.to_csv :show_table_headers => false, :io => STDOUT
end
```

== Custo m Records, Grouping, and processing XML data with HPricot ==

```
%w[rubygems ruport hpricot open-uri].each { |lib| require lib }

class TicketCountRenderer < Ruport::Renderer

 include AutoRunner

 required_option :timeline_uri
  option :days
  options { |o| o.days = 7 }

 stage :summary

 class TicketStatus < Ruport::Data::Record

   def closed
      title =~ /Ticket.+(\w+ closed)/ ? 1 : 0
    end

   def opened
      title =~ /Ticket.+(\w+ created)|(\w+ reopened)/ ? 1 : 0
    end

 end

 def retrive_base_data
    uri = options.timeline_uri << "?wiki=on&milestone=on&ticket=on&changeset=on"+
     "&max=10000&daysback=#{options.days-1}&format=rss"
    feed = Hpricot(open(uri))
    table = Table([:title, :date], :record_class => TicketStatus) do |table|
      (feed/"item").each do |r|
         title = (r/"title").innerHTML
         next unless title =~ /Ticket.*(created|closed)/
         table <<  { :title => title,
                     :date  => Date.parse((r/"pubdate").innerHTML) }
       end
    end
    @grouping = Grouping(table,:by => :date)
  end

 def calculate_sums
    @summary = Table(:date, :opened,:closed)
    @grouping.each do |date,group|
      opened = group.sigma { |r| r.opened  }
      closed = group.sigma { |r| r.closed  }
      @summary << { :date => date, :opened => opened, :closed => closed }
    end
    @summary = @summary.sort_rows_by { |r| r.date }
  end

 def run
    retrive_base_data
    calculate_sums
    options.summary = @summary
  end
end
```

```
class TicketCountFormatter < Ruport::Formatter

 renders [:html,:text,:csv,:pdf], :for => TicketCountRenderer
  opt_reader :summary

 def build_summary
    output << summary.as(format)
  end

end

timeline = "http://dev.rubyonrails.org/timeline"
puts TicketCountRenderer.render_text(:timeline_uri => timeline, :days => 9)
```

== *Simple PDF Output Tweaking* ==

```
class NotesTable < Ruport::Renderer
  stage :notes_table
end
class NotesTablePDF < Ruport::Formatter::PDF
  renders :pdf, :for => NotesTable
  def build_notes_table
    options.paper_orientation = :landscape
    pdf_writer.start_page_numbering(750,5,10,:center,"<PAGENUM>")
    draw_table data, :paper_orientation => :landscape,
                     :font_size => 8,
                     :maximum_width => 700,
                     :heading_font_size => 10
    pad(15) { add_text "Total: $#{options.total}" }
    output << pdf_writer.render
  end
end
```

Rope is your friend. (From the Ruport Doc Effort)

Overview

Rope provides you with a number of simple utilities that script away much of your boilerplate code, and also provide useful tools for development, such as automatic test running and a way to check your SQL queries for validity. Additionally, you'll get logging for free and you can share a common configuration file between applications in the same project.

Though each tool on it's own isn't complicated, having them all working together can be a major win.

The Basics

Starting a new rope project

```
$ rope labyrinth
creating directories..
  labyrinth/test
  labyrinth/config
  labyrinth/output
  labyrinth/data
  labyrinth/lib
  labyrinth/lib/reports
  labyrinth/lib/renderers
  labyrinth/templates
  labyrinth/sql
  labyrinth/log
  labyrinth/util
creating files..
  labyrinth/lib/reports.rb
  labyrinth/lib/helpers.rb
  labyrinth/lib/renderers.rb
  labyrinth/lib/init.rb
  labyrinth/config/ruport_config.rb
  labyrinth/util/build
  labyrinth/util/sql_exec
  labyrinth/Rakefile
```

Once this is complete, you'll have a large number of mostly empty folders laying around, along with some helpful tools at your disposal.

utilities

- build : A tool for generating reports and formatting system extensions
- sql_exec: A simple tool for getting a result set from a SQL file (possibly with ERb)
- Rakefile: Script for project automation tasks. Includes a default test runner, in installer for acts_as_reportable, and an alternative interface to the build script

directories

- test : unit tests stored here can be auto-run
- config : holds a configuration file which is shared across your applications
- reports : when reports are autogenerated, they are stored here
- renderers : autogenerated formatting system extensions are stored here
- templates : erb templates may be stored here
- sql : SQL files can be stored here, which are pre-processed by erb
- log : The logger will automatically store your logfiles here by default
- util : contains rope related tools

Generating a `Report` definition with rope

```
$ ./util/build report ghosts
report file: lib/reports/ghosts.rb
test file: test/test_ghosts.rb
class name: Ghosts

$ rake
(in /home/sandal/labyrinth)
/usr/bin/ruby -Ilib:test "/usr/lib/ruby/gems/1.8/gems/rake-
0.7.1/lib/rake/rake_test_loader.rb" "test/test_ghosts.rb"
Loaded suite /usr/lib/ruby/gems/1.8/gems/rake-0.7.1/lib/rake/rake_test_loader
Started
F
Finished in 0.001119 seconds.

1) Failure:
test_flunk(TestGhosts) [./test/test_ghosts.rb:6]:
Write your real tests here or in any test/test_* file.

1 tests, 1 assertions, 1 failures, 0 errors
rake aborted!
Command failed with status (1): [/usr/bin/ruby -Ilib:test "/usr/lib/ruby/ge...]

 (See full trace by running task with --trace)
```

You can now edit lib/reports/ghosts.rb as needed and write tests for it in test/test_ghosts.rb without having to hook everything up.

Rope’s Autogenerated Configuration File

Basic Details

roped projects will automatically make use of the configuration details in config/ruport_config.rb , which can be used to set up database connections, Ruport’s mailer, and other project information.

The default file is shown below.

```
require "ruport"
# For details, see Ruport::Config documentation
Ruport.configure { |c|
  c.source :default, :user     => "root",
                     :dsn      =>  "dbi:mysql:mydb"
  c.log_file "log/ruport.log"
}
```

In order for the `source` attribute to be useful, you’ll need to tweak it to have the right dsn for your database, check the DBI docs for that. You can of course, add additional sources as needed.

`Report#log` and `Ruport.log` will use the logfile you specify, which is set to `ruport.log` by default.

Special file layouts.

A handy feature rope provides is dynamic directory shortcuts. It adds the following methods to `Report` (via lib/init.rb) for your convenience:

- data_dir
- template_dir
- query_dir
- output_dir

Here is a simple example of using them.

```
load_csv data_dir/"foo.csv"
```

You can override the directories these shortcuts point to via your config file.

```
Ruport.configure { |c|
  c.source :default, :user     => "root",
                     :dsn      =>  "dbi:mysql:mydb"
  c.log_file "log/ruport.log"
  # these will be used by default, but you can override them in your Reports
  c.data_dir = "/usr/share/queries"
  c.output_dir = "/var/www/public_html"
}
```

Custom rendering with rope generators.

By Example

```
$ rope my_reverser
$ cd my_reverser
$ rake build renderer=reverser
```

Edit test/test_reverser.rb to look like the code below:

```
require "test/unit"
require "lib/renderers/reverser"
class TestReverser < Test::Unit::TestCase
  def test_reverser
    assert_equal "baz", Reverser.render_text("zab")
  end
end
```

Now edit lib/renderers/reverser.rb to look like this:

```
require "lib/init"
class Reverser < Ruport::Renderer
  stage :reverser
end
class ReverserFormatter < Ruport::Formatter
  renders :text, :for => Reverser
  def build_reverser
    output << data.reverse
  end
end
```

The tests should pass. You can now generate a quick report using this renderer

```
$ rake build report=reversed_report
```

Edit test/test_reversed_report.rb as such:

```
require "test/unit"
require "lib/reports/reversed_report"
class TestReversedReport < Test::Unit::TestCase
  def test_reversed_report
    report = ReversedReport.new
    report.message = "hello"
    assert_equal "olleh", report.to_text
  end
end
```

edit lib/reports/reversed_report.rb as below and run the tests.

```
  require "lib/init"
  class ReversedReport < Ruport::Report
    renders_with Reverser
    attr_accessor :message
    def generate
      message
    end
  end
```

ActiveRecord integration the lazy way.

```
$ rake install_aar
```

This will install the plugin in vendor/plugins/acts_as_reportable and set up
the necessary commands in app/init.rb to hook everything together. Note you need
svn installed for this to work.

Setup details

Change the following code in lib/init.rb to match your config information

```
ActiveRecord::Base.establish_connection(
  :adapter  => 'mysql',
  :host     => 'localhost',
  :username => 'name',
  :password => 'password',
  :database => 'mydb'
)
```

You need to add the proper call to your models.

```
class MyModel < ActiveRecord::Base
  acts_as_reportable
end
```

For more complex stuff, check the appropriate acts_as_reportable / ActiveRecord docs

You Gots What We Need:

- Feedback on overall usefulness of Ruport
- Questions about how to do different kinds of things
- Beautiful API ideas
- Bug Reports and Feature Requests
- Contributions such as documentation and patches

We will happily help you use Ruport in your projects. This helps drive the code, and exposes problems so we can fix them.

If you'd like to help, just catch up with us on IRC or the Ruport mailing list.

Happy Hacking!

Ruby Reports Users Field Guide