Ruby on Rails Guides: A Guide to The Rails Command Line [*1]

Rails comes with every command line tool you'll need to

- Create a Rails application
- Generate models, controllers, database migrations, and unit tests
- Start a development server
- Experiment with objects through an interactive shell
- Profile and benchmark your new creation

> *NOTE: This tutorial assumes you have basic Rails knowledge from reading the "Getting Started with Rails Guide".*

This Guide is based on Rails 3.0. Some of the code shown here will not work in earlier versions of Rails.

1 Command Line Basics

There are a few commands that are absolutely critical to your everyday usage of Rails. In the order of how much you'll probably use them are:

- `rails console`
- `rails server`
- `rake`
- `rails generate`
- `rails dbconsole`
- `rails new app_name`

Let's create a simple Rails application to step through each of these commands in context.

1.1 rails new

The first thing we'll want to do is create a new Rails application by running the `rails new` command after installing Rails.

You can install the rails gem by typing `gem install rails`, if you don't have it already. Follow the instructions in the *"Rails 3 Release Notes"*

Rails will set you up with what seems like a huge amount of stuff for such a tiny command!

[*1] Original: *http://guides.rubyonrails.org/command_line.html*

You've got the entire Rails directory structure now with all the code you need to run our simple application right out of the box.

1.2 rails server

The `rails server` command launches a small web server named WEBrick which comes bundled with Ruby. You'll use this any time you want to access your application through a web browser.

> *INFO: WEBrick isn't your only option for serving Rails. We'll get to that "later".*

With no further work, `rails server` will run our new shiny Rails app:

With just three commands we whipped up a Rails server listening on port 3000. Go to your browser and open *"http://localhost:3000"*, you will see a basic Rails app running.

You can also use the alias "s" to start the server: `rails s`.

The server can be run on a different port using the `-p` option. The default development environment can be changed using `-e`.

1.3 rails generate

The `rails generate` command uses templates to create a whole lot of things. Running `rails generate` by itself gives a list of available generators:

You can also use the alias "g" to invoke the generator command: `rails g`.

> *NOTE: You can install more generators through generator gems, portions of plugins you'll undoubtedly install, and you can even create your own!*

Using generators will save you a large amount of time by writing *boilerplate code*, code that is necessary for the app to work.

Let's make our own controller with the controller generator. But what command should we use? Let's ask the generator:

> *INFO: All Rails console utilities have help text. As with most *nix utilities, you can try adding* `--help` *or* `-h` *to the end, for example* `rails server --help`.

The controller generator is expecting parameters in the form of `generate controller ControllerName action1 action2`. Let's make a `Greetings` controller with an action of *hello*, which will say something nice to us.

What all did this generate? It made sure a bunch of directories were in our application, and created a controller file, a view file, a functional test file, a helper for the view, a javascript

file and a stylesheet file.

Check out the controller and modify it a little (in `app/controllers/greetings_controller.rb`):

Then the view, to display our message (in `app/views/greetings/hello.html.erb`):

Fire up your server using `rails server`.

Make sure that you do not have any "tilde backup" files in `app/views/(controller)`, or else WEBrick will *not* show the expected output. This seems to be a *bug* in Rails 2.3.0.

The URL will be *"http://localhost:3000/greetings/hello"*.

> *INFO: With a normal, plain-old Rails application, your URLs will generally follow the pattern of http://(host)/(controller)/(action), and a URL like http://(host)/(controller) will hit the index action of that controller.*

Rails comes with a generator for data models too.

> *NOTE: For a list of available field types, refer to the "API documentation" for the column method for the* `TableDefinition` *class.*

But instead of generating a model directly (which we'll be doing later), let's set up a scaffold. A *scaffold* in Rails is a full set of model, database migration for that model, controller to manipulate it, views to view and manipulate the data, and a test suite for each of the above.

We will set up a simple resource called "HighScore" that will keep track of our highest score on video games we play.

The generator checks that there exist the directories for models, controllers, helpers, layouts, functional and unit tests, stylesheets, creates the views, controller, model and database migration for HighScore (creating the `high_scores` table and fields), takes care of the route for the *resource*, and new tests for everything.

The migration requires that we *migrate*, that is, run some Ruby code (living in that `20100209025147_create_high_scores.rb`) to modify the schema of our database. Which database? The sqlite3 database that Rails will create for you when we run the `rake db:migrate` command. We'll talk more about Rake in-depth in a little while.

> *INFO: Let's talk about unit tests. Unit tests are code that tests and makes assertions about code. In unit testing, we take a little part of code, say a method of a model, and test its inputs and outputs. Unit tests are your friend. The sooner you make peace with the fact that your quality of life will drastically increase when you unit test your code, the better. Seriously. We'll make one in a moment.*

Let's see the interface Rails created for us.

Go to your browser and open *"http://localhost:3000/high_scores"*, now we can create new high scores (55,160 on Space Invaders!)

1.4 rails console

The `console` command lets you interact with your Rails application from the command line. On the underside, `rails console` uses IRB, so if you've ever used it, you'll be right at home. This is useful for testing out quick ideas with code and changing data server-side without touching the website.

You can also use the alias "c" to invoke the console: `rails c`.

If you wish to test out some code without changing any data, you can do that by invoking `rails console --sandbox`.

1.5 rails dbconsole

`rails dbconsole` figures out which database you're using and drops you into whichever command line interface you would use with it (and figures out the command line parameters to give to it, too!). It supports MySQL, PostgreSQL, SQLite and SQLite3.

You can also use the alias "db" to invoke the dbconsole: `rails db`.

1.6 rails plugin

The `rails plugin` command simplifies plugin management. Plugins can be installed by name or their repository URLs. You need to have Git installed if you want to install a plugin from a Git repo. The same holds for Subversion too.

1.7 rails runner

`runner` runs Ruby code in the context of Rails non-interactively. For instance:

You can also use the alias "r" to invoke the runner: `rails r`.

You can specify the environment in which the `runner` command should operate using the `-e` switch.

1.8 rails destroy

Think of `destroy` as the opposite of `generate`. It'll figure out what generate did, and undo it.

2 Rake

Rake is Ruby Make, a standalone Ruby utility that replaces the Unix utility 'make', and uses a 'Rakefile' and `.rake` files to build up a list of tasks. In Rails, Rake is used for common administration tasks, especially sophisticated ones that build off of each other.

You can get a list of Rake tasks available to you, which will often depend on your current directory, by typing `rake --tasks`. Each task has a description, and should help you find the thing you need.

2.1 about

`rake about` gives information about version numbers for Ruby, RubyGems, Rails, the Rails subcomponents, your application's folder, the current Rails environment name, your app's database adapter, and schema version. It is useful when you need to ask for help, check if a security patch might affect you, or when you need some stats for an existing Rails installation.

2.2 assets

You can precompile the assets in `app/assets` using `rake assets:precompile` and remove compiled assets using `rake assets:clean`.

2.3 db

The most common tasks of the `db:` Rake namespace are `migrate` and `create`, and it will pay off to try out all of the migration rake tasks (`up`, `down`, `redo`, `reset`). `rake db:version` is useful when troubleshooting, telling you the current version of the database.

More information about migrations can be found in the *"Migrations"* guide.

2.4 doc

The `doc:` namespace has the tools to generate documentation for your app, API documentation, guides. Documentation can also be stripped which is mainly useful for slimming your codebase, like if you're writing a Rails application for an embedded platform.

- `rake doc:app` generates documentation for your application in `doc/app`.
- `rake doc:guides` generates Rails guides in `doc/guides`.
- `rake doc:rails` generates API documentation for Rails in `doc/api`.

- `rake doc:plugins` generates API documentation for all the plugins installed in the application in `doc/plugins`.
- `rake doc:clobber_plugins` removes the generated documentation for all plugins.

2.5 notes

`rake notes` will search through your code for comments beginning with FIXME, OPTIMIZE or TODO. The search is only done in files with extension `.builder`, `.rb`, `.rxml`, `.rhtml` and `.erb` for both default and custom annotations.

If you are looking for a specific annotation, say FIXME, you can use `rake notes:fixme`. Note that you have to lower case the annotation's name.

You can also use custom annotations in your code and list them using `rake notes:custom` by specifying the annotation using an environment variable `ANNOTATION`.

> *NOTE: When using specific annotations and custom annotations, the annotation name (FIXME, BUG etc) is not displayed in the output lines.*

2.6 routes

`rake routes` will list all of your defined routes, which is useful for tracking down routing problems in your app, or giving you a good overview of the URLs in an app you're trying to get familiar with.

2.7 test

> *INFO: A good description of unit testing in Rails is given in "A Guide to Testing Rails Applications"*

Rails comes with a test suite called Test::Unit. It is through the use of tests that Rails itself is so stable, and the slew of people working on Rails can prove that everything works as it should.

The `test:` namespace helps in running the different tests you will (hopefully!) write.

2.8 tmp

The `Rails.root/tmp` directory is, like the *nix /tmp directory, the holding place for temporary files like sessions (if you're using a file store for files), process id files, and cached actions. The `tmp:` namespace tasks will help you clear them if you need to if they've become

overgrown, or create them in case of deletions gone awry.

2.9 Miscellaneous

- `rake stats` is great for looking at statistics on your code, displaying things like KLOCs (thousands of lines of code) and your code to test ratio.
- `rake secret` will give you a pseudo-random key to use for your session secret.
- `rake time:zones:all` lists all the timezones Rails knows about.

3 The Rails Advanced Command Line

More advanced use of the command line is focused around finding useful (even surprising at times) options in the utilities, and fitting those to your needs and specific work flow. Listed here are some tricks up Rails' sleeve.

3.1 Rails with Databases and SCM

When creating a new Rails application, you have the option to specify what kind of database and what kind of source code management system your application is going to use. This will save you a few minutes, and certainly many keystrokes.

Let's see what a `--git` option and a `--database=postgresql` option will do for us:

We had to create the *gitapp* directory and initialize an empty git repository before Rails would add files it created to our repository. Let's see what it put in our database configuration:

It also generated some lines in our database.yml configuration corresponding to our choice of PostgreSQL for database.

> *NOTE: The only catch with using the SCM options is that you have to make your application's directory first, then initialize your SCM, then you can run the* `rails new` *command to generate the basis of your app.*

3.2 server with Different Backends

Many people have created a large number of different web servers in Ruby, and many of them can be used to run Rails. Since version 2.3, Rails uses Rack to serve its webpages, which means that any webserver that implements a Rack handler can be used. This includes WEBrick, Mongrel, Thin, and Phusion Passenger (to name a few!).

> *NOTE: For more details on the Rack integration, see "Rails on Rack".*

To use a different server, just install its gem, then use its name for the first parameter to `rails server`:

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"*.