The Rails Command Line

After reading this guide, you will know:

- **How to create a Rails application.**
- **How to generate models, controllers, database migrations, and unit tests.**
- **How to start a development server.**
- **How to experiment with objects through an interactive shell.**
- **How to profile and benchmark your new creation.**

Chapters

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

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`

All commands can run with `-h or --help` to list more information.

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.

```
$ rails new commandsapp
     create
     create  README.rdoc
     create  Rakefile
     create  config.ru
     create  .gitignore
     create  Gemfile
     create  app
     ...
     create  tmp/cache
     ...
        run  bundle install
```

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

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

```
$ cd commandsapp
$ bin/rails server
=> Booting WEBrick
=> Rails 4.0.0 application starting in development on
http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
[2013-08-07 02:00:01] INFO  WEBrick 1.3.1
[2013-08-07 02:00:01] INFO  ruby 2.0.0 (2013-06-27) [x86_64-
darwin11.2.0]
[2013-08-07 02:00:01] INFO  WEBrick::HTTPServer#start: pid=69680
port=3000
```

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`.

```
$ bin/rails server -e production -p 4000
```

The `-b` option binds Rails to the specified IP, by default it is 0.0.0.0. You can run a server as a daemon by passing a `-d` option.

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`.

```
$ bin/rails generate
Usage: rails generate GENERATOR [args] [options]

...
...

Please choose a generator below.

Rails:
  assets
```

```
    controller
    generator
    ...
    ...
```

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:

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`.

```
$ bin/rails generate controller
Usage: rails generate controller NAME [action action] [options]

...
...

Description:
    ...

    To create a controller within a module, specify the controller
name as a
    path like 'parent_module/controller_name'.

    ...

Example:
    `rails generate controller CreditCard open debit credit close`

    Credit card controller with URLs like /credit_card/debit.
        Controller: app/controllers/credit_card_controller.rb
        Test:       test/controllers/credit_card_controller_test.rb
        Views:      app/views/credit_card/debit.html.erb [...]
        Helper:     app/helpers/credit_card_helper.rb
```

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.

```
$ bin/rails generate controller Greetings hello
     create  app/controllers/greetings_controller.rb
      route  get "greetings/hello"
     invoke  erb
     create    app/views/greetings
     create    app/views/greetings/hello.html.erb
     invoke  test_unit
```

```
      create    test/controllers/greetings_controller_test.rb
      invoke  helper
      create    app/helpers/greetings_helper.rb
      invoke    test_unit
      create      test/helpers/greetings_helper_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/greetings.js.coffee
      invoke    scss
      create      app/assets/stylesheets/greetings.css.scss
```

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`):

```
class GreetingsController < ApplicationController
  def hello
    @message = "Hello, how are you today?"
  end
end
```

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

```
<h1>A Greeting for You!</h1>
<p><%= @message %></p>
```

Fire up your server using `rails server`.

```
$ bin/rails server
=> Booting WEBrick...
```

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

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.

```
$ bin/rails generate model
Usage:
  rails generate model NAME [field[:type][:index] field[:type]
[:index]] [options]

...
```

```
Active Record options:
      [--migration]           # Indicates when to generate migration
                              # Default: true

...

Description:
    Create rails files for model generator.
```

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.

```
$ bin/rails generate scaffold HighScore game:string score:integer
    invoke  active_record
    create    db/migrate/20130717151933_create_high_scores.rb
    create    app/models/high_score.rb
    invoke    test_unit
    create      test/models/high_score_test.rb
    create      test/fixtures/high_scores.yml
    invoke  resource_route
     route    resources :high_scores
    invoke  scaffold_controller
    create    app/controllers/high_scores_controller.rb
    invoke    erb
    create      app/views/high_scores
    create      app/views/high_scores/index.html.erb
    create      app/views/high_scores/edit.html.erb
    create      app/views/high_scores/show.html.erb
    create      app/views/high_scores/new.html.erb
    create      app/views/high_scores/_form.html.erb
    invoke    test_unit
    create      test/controllers/high_scores_controller_test.rb
    invoke    helper
    create      app/helpers/high_scores_helper.rb
    invoke      test_unit
    create        test/helpers/high_scores_helper_test.rb
    invoke    jbuilder
    create      app/views/high_scores/index.json.jbuilder
    create      app/views/high_scores/show.json.jbuilder
    invoke  assets
    invoke    coffee
    create      app/assets/javascripts/high_scores.js.coffee
    invoke    scss
    create      app/assets/stylesheets/high_scores.css.scss
    invoke  scss
   identical    app/assets/stylesheets/scaffolds.css.scss
```

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 `20130717151933_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.

```
$ bin/rake db:migrate
==  CreateHighScores: migrating
=============================================
-- create_table(:high_scores)
   -> 0.0017s
==  CreateHighScores: migrated (0.0019s)
======================================
```

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.

```
$ bin/rails server
```

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`.

You can specify the environment in which the `console` command should operate.

```
$ bin/rails console staging
```

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

```
$ bin/rails console --sandbox
Loading development environment in sandbox (Rails 4.0.0)
```

```
Any modifications you make will be rolled back on exit
irb(main):001:0>
```

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 runner`

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

```
$ bin/rails runner "Model.long_running_method"
```

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.

```
$ bin/rails runner -e staging "Model.long_running_method"
```

1.7 `rails destroy`

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

You can also use the alias "d" to invoke the destroy command: `rails d`.

```
$ bin/rails generate model Oops
      invoke  active_record
      create    db/migrate/20120528062523_create_oops.rb
      create    app/models/oops.rb
      invoke    test_unit
      create      test/models/oops_test.rb
      create      test/fixtures/oops.yml
```

```
$ bin/rails destroy model Oops
      invoke  active_record
      remove    db/migrate/20120528062523_create_oops.rb
      remove    app/models/oops.rb
      invoke    test_unit
      remove      test/models/oops_test.rb
      remove      test/fixtures/oops.yml
```

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.

To get the full backtrace for running rake task you can pass the option `--trace` to command line, for example `rake db:create --trace`.

```
$ bin/rake --tasks
rake about              # List versions of all Rails frameworks and
the environment
rake assets:clean       # Remove compiled assets
rake assets:precompile  # Compile all the assets named in
config.assets.precompile
rake db:create          # Create the database from
config/database.yml for the current Rails.env
...
rake log:clear          # Truncates all *.log files in log/ to zero
bytes (specify which logs with LOGS=test,development)
rake middleware         # Prints out your Rack middleware stack
...
rake tmp:clear          # Clear session, cache, and socket files from
tmp/ (narrow w/ tmp:sessions:clear, tmp:cache:clear,
tmp:sockets:clear)
rake tmp:create         # Creates tmp directories for sessions,
cache, sockets, and pids
```

You can also use `rake -T` to get the list of tasks.

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.

```
$ bin/rake about
About your application's environment
Ruby version              1.9.3 (x86_64-linux)
RubyGems version          1.3.6
Rack version              1.3
Rails version             4.1.1
JavaScript Runtime        Node.js (V8)
Active Record version     4.1.1
Action Pack version       4.1.1
Action View version       4.1.1
Action Mailer version     4.1.1
Active Support version    4.1.1
Middleware                Rack::Sendfile, ActionDispatch::Static,
```

```
Rack::Lock, #
<ActiveSupport::Cache::Strategy::LocalCache::Middleware:0x007ffd131a7c8
Rack::Runtime, Rack::MethodOverride, ActionDispatch::RequestId,
Rails::Rack::Logger, ActionDispatch::ShowExceptions,
ActionDispatch::DebugExceptions, ActionDispatch::RemoteIp,
ActionDispatch::Reloader, ActionDispatch::Callbacks,
ActiveRecord::Migration::CheckPending,
ActiveRecord::ConnectionAdapters::ConnectionManagement,
ActiveRecord::QueryCache, ActionDispatch::Cookies,
ActionDispatch::Session::CookieStore, ActionDispatch::Flash,
ActionDispatch::ParamsParser, Rack::Head, Rack::ConditionalGet, Rack::E
Application root          /home/foobar/commandsapp
Environment               development
Database adapter          sqlite3
Database schema version   20110805173523
```

2.2 assets

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

2.5 notes

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

```
$ bin/rake notes
(in /home/foobar/commandsapp)
app/controllers/admin/users_controller.rb:
  * [ 20] [TODO] any other way to do this?
  * [132] [FIXME] high priority for next deploy

app/models/school.rb:
```

```
    * [ 13] [OPTIMIZE] refactor this code to make it faster
    * [ 17] [FIXME]
```

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.

```
$ bin/rake notes:fixme
(in /home/foobar/commandsapp)
app/controllers/admin/users_controller.rb:
  * [132] high priority for next deploy

app/models/school.rb:
  * [ 17]
```

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`.

```
$ bin/rake notes:custom ANNOTATION=BUG
(in /home/foobar/commandsapp)
app/models/post.rb:
  * [ 23] Have to fix this one before pushing!
```

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

By default, `rake notes` will look in the `app`, `config`, `lib`, `bin` and `test` directories. If you would like to search other directories, you can provide them as a comma separated list in an environment variable `SOURCE_ANNOTATION_DIRECTORIES`.

```
$ export SOURCE_ANNOTATION_DIRECTORIES='spec,vendor'
$ bin/rake notes
(in /home/foobar/commandsapp)
app/models/user.rb:
  * [ 35] [FIXME] User should have a subscription at this point
spec/models/user_spec.rb:
  * [122] [TODO] Verify the user that has a subscription works
```

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`

A good description of unit testing in Rails is given in **A Guide to Testing Rails Applications**

Rails comes with a test suite called Minitest. Rails owes its stability to the use of tests. The tasks available

in 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:` namespaced tasks will help you clear and create the `Rails.root/tmp` directory:

- `rake tmp:cache:clear` clears `tmp/cache`.
- `rake tmp:sessions:clear` clears `tmp/sessions`.
- `rake tmp:sockets:clear` clears `tmp/sockets`.
- `rake tmp:clear` clears all the three: cache, sessions and sockets.
- `rake tmp:create` creates tmp directories for sessions, cache, sockets, and pids.

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.

2.10 Custom Rake Tasks

Custom rake tasks have a `.rake` extension and are placed in `Rails.root/lib/tasks`. You can create these custom rake tasks with the `bin/rails generate task` command.

```
desc "I am short, but comprehensive description for my cool task"
task task_name: [:prerequisite_task, :another_task_we_depend_on] do
  # All your magic here
  # Any valid Ruby code is allowed
end
```

To pass arguments to your custom rake task:

```
task :task_name, [:arg_1] => [:pre_1, :pre_2] do |t, args|
  # You can use args from here
end
```

You can group tasks by placing them in namespaces:

```
namespace :db do
  desc "This task does nothing"
  task :nothing do
    # Seriously, nothing
  end
end
```

Invocation of the tasks will look like:

```
$ bin/rake task_name
$ bin/rake "task_name[value 1]" # entire argument string should be
quoted
$ bin/rake db:nothing
```

If your need to interact with your application models, perform database queries and so on, your task should depend on the `environment` task, which will load your application code.

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:

```
$ mkdir gitapp
$ cd gitapp
$ git init
Initialized empty Git repository in .git/
$ rails new . --git --database=postgresql
      exists
      create  app/controllers
      create  app/helpers
...
...
      create  tmp/cache
      create  tmp/pids
      create  Rakefile
add 'Rakefile'
      create  README.rdoc
add 'README.rdoc'
      create  app/controllers/application_controller.rb
add 'app/controllers/application_controller.rb'
      create  app/helpers/application_helper.rb
...
      create  log/test.log
add 'log/test.log'
```

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:

```
$ cat config/database.yml
# PostgreSQL. Versions 8.2 and up are supported.
#
# Install the pg driver:
#   gem install pg
# On OS X with Homebrew:
#   gem install pg -- --with-pg-config=/usr/local/bin/pg_config
# On OS X with MacPorts:
#   gem install pg -- --with-pg-
config=/opt/local/lib/postgresql84/bin/pg_config
# On Windows:
#   gem install pg
#       Choose the win32 build.
#       Install PostgreSQL and put its /bin directory on your path.
#
# Configure Using Gemfile
# gem 'pg'
#
development:
  adapter: postgresql
  encoding: unicode
  database: gitapp_development
  pool: 5
  username: gitapp
  password:
...
...
```

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

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.

Feedback

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

Please contribute if you see any typos or factual errors. To get started, you can read our **documentation contributions** section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check **Edge Guides** first to verify if the issues are already fixed or not on the master branch. 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**.