**DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.**

# 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.

---

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

## 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:

- `bin/rails console`
- `bin/rails server`
- `bin/rails test`
- `bin/rails generate`
- `bin/rails db:migrate`
- `bin/rails db:create`
- `bin/rails routes`
- `bin/rails dbconsole`
- `rails new app_name`

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

```
$ rails --help
Usage: rails COMMAND [ARGS]

The most common rails commands are:
 generate    Generate new code (short-cut alias: "g")
 console     Start the Rails console (short-cut alias: "c")
 server      Start the Rails server (short-cut alias: "s")
 ...

All commands can be run with -h (or --help) for more information.

In addition to those commands, there are:
 about                              List versions of all Rails ...
 assets:clean[keep]                 Remove old compiled assets
 assets:clobber                     Remove compiled assets
 assets:environment                 Load asset compile environment
 assets:precompile                  Compile all the assets ...
 ...
 db:fixtures:load                   Loads fixtures into the ...
```

```
 db:migrate                          Migrate the database ...
 db:migrate:status                   Display status of migrations
 db:rollback                         Rolls the schema back to ...
 db:schema:cache:clear               Clears a db/schema_cache.yml file
 db:schema:cache:dump                Creates a db/schema_cache.yml file
 db:schema:dump                      Creates a database schema file (either
db/schema.rb or db/structure.sql ...
 db:schema:load                      Loads a database schema file (either
db/schema.rb or db/structure.sql ...
 db:seed                             Loads the seed data ...
 db:version                          Retrieves the current schema ...
 ...
 restart                             Restart app by touching ...
 tmp:create                          Creates tmp directories ...
```

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

### `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.

INFO: You can install the rails gem by typing `gem install rails`, if you don't have it already.

```
$ rails new commandsapp
     create
     create  README.md
     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.

If you wish to skip some files or components from being generated, you can append the following arguments to your `rails new` command:

| Argument | Description |
| --- | --- |
| `--skip-git` | Skip .gitignore file |
| `--skip-keeps` | Skip source control .keep files |
| `--skip-action-mailer` | Skip Action Mailer files |
| `--skip-action-mailbox` | Skip Action Mailbox gem |

| | |
|---|---|
| `--skip-action-text` | Skip Action Text gem |
| `--skip-active-record` | Skip Active Record files |
| `--skip-active-job` | Skip Active Job |
| `--skip-active-storage` | Skip Active Storage files |
| `--skip-action-cable` | Skip Action Cable files |
| `--skip-asset-pipeline` | Skip Asset Pipeline |
| `--skip-javascript` | Skip JavaScript files |
| `--skip-hotwire` | Skip Hotwire integration |
| `--skip-jbuilder` | Skip jbuilder gem |
| `--skip-test` | Skip test files |
| `--skip-system-test` | Skip system test files |
| `--skip-bootsnap` | Skip bootsnap gem |

### `bin/rails server`

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

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

```
$ cd commandsapp
$ bin/rails server
=> Booting Puma
=> Rails 7.0.0 application starting in development
=> Run `bin/rails server --help` for more startup options
Puma starting in single mode...
* Version 3.12.1 (ruby 2.5.7-p206), codename: Llamas in Pajamas
* Min threads: 5, max threads: 5
* Environment: development
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
```

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.

INFO: You can also use the alias "s" to start the server: `bin/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 localhost. You can run a server as a daemon by passing a `-d` option.

### `bin/rails generate`

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

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

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

...
...

Please choose a generator below.

Rails:
  assets
  channel
  controller
  generator
  ...
  ...
```

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 `bin/rails server --help`.

```
$ bin/rails generate controller
Usage: bin/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:
    `bin/rails generate controller CreditCards open debit credit close`

    Credit card controller with URLs like /credit_cards/debit.
        Controller: app/controllers/credit_cards_controller.rb
        Test:       test/controllers/credit_cards_controller_test.rb
        Views:      app/views/credit_cards/debit.html.erb [...]
        Helper:     app/helpers/credit_cards_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
```

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 `bin/rails server`.

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

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.

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

...

ActiveRecord options:
      [--migration], [--no-migration]        # Indicates when to generate migration
                                             # Default: true

...

Description:
    Generates a new model. Pass the model name, either CamelCased or
    under_scored, and an optional list of attribute pairs as arguments.

...
```

NOTE: For a list of available field types for the `type` parameter, refer to the API documentation for the add_column method for the `SchemaStatements` module. The `index` parameter generates a corresponding index for the column.

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/20190416145729_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
    create      test/system/high_scores_test.rb
    invoke    helper
    create      app/helpers/high_scores_helper.rb
    invoke      test_unit
    invoke    jbuilder
    create      app/views/high_scores/index.json.jbuilder
    create      app/views/high_scores/show.json.jbuilder
    create      app/views/high_scores/_high_score.json.jbuilder
```

The generator creates the model, views, controller, **resource** route, and database migration (which creates the `high_scores` table) for HighScore. And it adds tests for those.

The migration requires that we **migrate**, that is, run some Ruby code (the `20190416145729_create_high_scores.rb` file from the above output) to modify the schema of our database. Which database? The SQLite3 database that Rails will create for you when we run the `bin/rails db:migrate` command. We'll talk more about that command below.

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

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. Please visit the testing guide for an in-depth look at unit testing.

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!)

### `bin/rails console`

The `console` command lets you interact with your Rails application from the command line. On the underside, `bin/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.

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

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

```
$ bin/rails console -e staging
```

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

```
$ bin/rails console --sandbox
Loading development environment in sandbox (Rails 7.1.0)
Any modifications you make will be rolled back on exit
irb(main):001:0>
```

#### The app and helper objects

Inside the `bin/rails console` you have access to the `app` and `helper` instances.

With the `app` method you can access named route helpers, as well as do requests.

```
irb> app.root_path
=> "/"

irb> app.get _
Started GET "/" for 127.0.0.1 at 2014-06-19 10:41:57 -0300
...
```

With the `helper` method it is possible to access Rails and your application's helpers.

```
irb> helper.time_ago_in_words 30.days.ago
=> "about 1 month"

irb> helper.my_custom_helper
=> "my custom helper"
```

### `bin/rails dbconsole`

`bin/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 (including MariaDB), PostgreSQL, and SQLite3.

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

If you are using multiple databases, `bin/rails dbconsole` will connect to the primary database by default. You can specify which database to connect to using `--database` or `--db`:

```
$ bin/rails dbconsole --database=animals
```

### `bin/rails runner`

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

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

INFO: You can also use the alias "r" to invoke the runner: `bin/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"
```

You can even execute ruby code written in a file with runner.

```
$ bin/rails runner lib/code_to_be_run.rb
```

### `bin/rails destroy`

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

INFO: You can also use the alias "d" to invoke the destroy command: `bin/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
```

### `bin/rails about`

`bin/rails 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/rails about
About your application's environment
Rails version             7.0.0
Ruby version              2.7.0 (x86_64-linux)
RubyGems version          2.7.3
Rack version              2.0.4
JavaScript Runtime        Node.js (V8)
Middleware:               Rack::Sendfile, ActionDispatch::Static,
ActionDispatch::Executor, ActiveSupport::Cache::Strategy::LocalCache::Middleware,
Rack::Runtime, Rack::MethodOverride, ActionDispatch::RequestId,
ActionDispatch::RemoteIp, Sprockets::Rails::QuietAssets, Rails::Rack::Logger,
ActionDispatch::ShowExceptions, WebConsole::Middleware,
```

```
ActionDispatch::DebugExceptions, ActionDispatch::Reloader,
ActionDispatch::Callbacks, ActiveRecord::Migration::CheckPending,
ActionDispatch::Cookies, ActionDispatch::Session::CookieStore,
ActionDispatch::Flash, Rack::Head, Rack::ConditionalGet, Rack::ETag
Application root          /home/foobar/commandsapp
Environment               development
Database adapter          sqlite3
Database schema version   20180205173523
```

### `bin/rails assets:`

You can precompile the assets in `app/assets` using `bin/rails assets:precompile`, and remove older compiled assets using `bin/rails assets:clean`. The `assets:clean` command allows for rolling deploys that may still be linking to an old asset while the new assets are being built.

If you want to clear `public/assets` completely, you can use `bin/rails assets:clobber`.

### `bin/rails db:`

The most common commands of the `db:` rails namespace are `migrate` and `create`, and it will pay off to try out all of the migration rails commands (`up`, `down`, `redo`, `reset`). `bin/rails 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.

### `bin/rails notes`

`bin/rails notes` searches through your code for comments beginning with a specific keyword. You can refer to `bin/rails notes --help` for information about usage.

By default, it will search in `app`, `config`, `db`, `lib`, and `test` directories for FIXME, OPTIMIZE, and TODO annotations in files with extension `.builder`, `.rb`, `.rake`, `.yml`, `.yaml`, `.ruby`, `.css`, `.js`, and `.erb`.

```
$ bin/rails notes
app/controllers/admin/users_controller.rb:
  * [ 20] [TODO] any other way to do this?
  * [132] [FIXME] high priority for next deploy

lib/school.rb:
  * [ 13] [OPTIMIZE] refactor this code to make it faster
  * [ 17] [FIXME]
```

#### Annotations

You can pass specific annotations by using the `--annotations` argument. By default, it will search for FIXME, OPTIMIZE, and TODO. Note that annotations are case sensitive.

```
$ bin/rails notes --annotations FIXME RELEASE
app/controllers/admin/users_controller.rb:
  * [101] [RELEASE] We need to look at this before next release
```

```
  * [132] [FIXME] high priority for next deploy

lib/school.rb:
  * [ 17] [FIXME]
```

#### Tags

You can add more default tags to search for by using `config.annotations.register_tags`. It receives a list of tags.

```
config.annotations.register_tags("DEPRECATEME", "TESTME")
```

```
$ bin/rails notes
app/controllers/admin/users_controller.rb:
  * [ 20] [TODO] do A/B testing on this
  * [ 42] [TESTME] this needs more functional tests
  * [132] [DEPRECATEME] ensure this method is deprecated in next release
```

#### Directories

You can add more default directories to search from by using `config.annotations.register_directories`. It receives a list of directory names.

```
config.annotations.register_directories("spec", "vendor")
```

```
$ bin/rails notes
app/controllers/admin/users_controller.rb:
  * [ 20] [TODO] any other way to do this?
  * [132] [FIXME] high priority for next deploy

lib/school.rb:
  * [ 13] [OPTIMIZE] Refactor this code to make it faster
  * [ 17] [FIXME]

spec/models/user_spec.rb:
  * [122] [TODO] Verify the user that has a subscription works

vendor/tools.rb:
  * [ 56] [TODO] Get rid of this dependency
```

#### Extensions

You can add more default file extensions to search from by using `config.annotations.register_extensions`. It receives a list of extensions with its corresponding regex to match it up.

```
config.annotations.register_extensions("scss", "sass") { |annotation| /\/\/\s*(#
{annotation}):?\s*(.*)$/ }
```

```
$ bin/rails notes
app/controllers/admin/users_controller.rb:
  * [ 20] [TODO] any other way to do this?
  * [132] [FIXME] high priority for next deploy

app/assets/stylesheets/application.css.sass:
  * [ 34] [TODO] Use pseudo element for this class

app/assets/stylesheets/application.css.scss:
  * [  1] [TODO] Split into multiple components

lib/school.rb:
  * [ 13] [OPTIMIZE] Refactor this code to make it faster
  * [ 17] [FIXME]

spec/models/user_spec.rb:
  * [122] [TODO] Verify the user that has a subscription works

vendor/tools.rb:
  * [ 56] [TODO] Get rid of this dependency
```

#### `bin/rails routes`

`bin/rails 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.

#### `bin/rails test`

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

Rails comes with a test framework called minitest. Rails owes its stability to the use of tests. The commands available in the `test:` namespace helps in running the different tests you will hopefully write.

#### `bin/rails tmp:`

The `Rails.root/tmp` directory is, like the *nix /tmp directory, the holding place for temporary files like process id files and cached actions.

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

- `bin/rails tmp:cache:clear` clears `tmp/cache`.
- `bin/rails tmp:sockets:clear` clears `tmp/sockets`.
- `bin/rails tmp:screenshots:clear` clears `tmp/screenshots`.
- `bin/rails tmp:clear` clears all cache, sockets, and screenshot files.
- `bin/rails tmp:create` creates tmp directories for cache, sockets, and pids.

### Miscellaneous

- `bin/rails initializers` prints out all defined initializers in the order they are invoked by Rails.
- `bin/rails middleware` lists Rack middleware stack enabled for your app.
- `bin/rails 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.

- `bin/rails secret` will give you a pseudo-random key to use for your session secret.
- `bin/rails time:zones:all` lists all the timezones Rails knows about.

### 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] => [:prerequisite_1, :prerequisite_2] do |task, args|
  argument_1 = args.arg_1
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/rails task_name
$ bin/rails "task_name[value 1]" # entire argument string should be quoted
$ bin/rails "task_name[value 1,value2,value3]" # separate multiple args with a comma
$ bin/rails db:nothing
```

NOTE: If you 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.

## 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.

### 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.md
add 'README.md'
      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 9.3 and up are supported.
#
# Install the pg driver:
#   gem install pg
# On macOS with Homebrew:
#   gem install pg -- --with-pg-config=/usr/local/bin/pg_config
# On macOS 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'
#
default: &default
  adapter: postgresql
  encoding: unicode
  # For details on connection pooling, see Rails configuration guide
  # https://guides.rubyonrails.org/configuring.html#database-pooling
  pool: <%= ENV.fetch("RAILS_MAX_THREADS") { 5 } %>
```

```
development:
  <<: *default
  database: gitapp_development
...
...
```

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.