

Proverbs: My first app with Rails

Install rails 5.2

```
gem install rails -v 5.2.4
```

When you install the Rails framework, you also get a new command-line tool, `rails`, that's used to construct each new Rails application you write. It is installed along with the rails gem.

We're going to run that executable now to generate our app.

```
rails new Proverbs
```

This:

1. creates a new directory with the same name as the app, containing a whole bunch of files and directories inside it
2. downloads and installs some additional gems using the ruby gem manager "Bundler" (Gem Bundler helps you track and install the gems you need for your any Ruby project). By default some gems are added for you to work with. Gems are like Ruby libraries with some functionality, installed to satisfy specific needs.

Our Proverbs app is already set up to run and display a generic welcome page.

To test it, change into the new app directory (Proverbs) then type:

```
cd Proverbs
rails server
#or
rails s
```

This:

1. boots Puma, the web server that Rails 5 runs on top of.
2. starts our Rails 5.2.4.4 application (in development) on localhost port 3000

The important part, there is the URL at the end; `localhost:3000` is the address where

the server is running. Localhost means it is on the local machine, your own computer. 3000 is the port it is running on. Multiple servers can run on the same computer as long as they take different connections on different ports. A rails application that is in development takes connections on port 3000, by default.

To check what is running on the port 3000, before running any rails app:

```
lsof -i:3000
```

This will list any process listening to the port 3000.

To kill any process listening to this port using its PID:

```
kill -9 PID
```

When we copy the `localhost:3000` URL and paste it into our web browser, Rails will respond with the following welcome page. Our app is already running.



Yay! You're on Rails!



Rails version: 5.2.4.1

Ruby version: 2.7.0 (x86_64-linux)

Now that we know the server works, let's shut it back down for the time being. When Rails started, it also printed one other message, `Use Ctrl-C to stop` the server. Those are the keys to press when you want to shut down the web server. So return to your terminal window and press Ctrl+C to stop the server.

Rails directory structure

















When you use the rails to create your application, it creates the entire directory structure for the application. Rails knows where to find things it needs within this structure, so you don't have to provide any input.

Here is a top-level view of a directory tree created at the time of application creation. Except

for minor changes between releases, every Rails project will have the same structure, with the same naming conventions. This consistency gives you a tremendous advantage; you can quickly move between Rails projects without relearning the project's organization.

To understand this directory structure, let's use the Proverbs application created above using the simple `rails new Proverbs` command.

Now, if we check the Proverbs application root directory, we will find a directory structure as shown in the figure below.

Folders
 app
 bin
 config
 db
 lib
 log
 public
 storage
 test
 tmp
 vendor
Other
 config.ru
 Gemfile
 Gemfile.lock
 package.json
 Rakefile

Now let's explain the purpose of each directory

- `app` – It organizes your application components. It's got subdirectories that hold the view (views and helpers), controller (controllers), and the backend business logic

(models).

- `app/controllers` – The controllers subdirectory is where Rails looks to find the controller classes.
- `app/assets` - This directory holds an application's stylesheet files (i.e., css) within a subdirectory.
- `app/helpers` – The helpers subdirectory holds any helper classes used to assist the model, view, and controller classes. This helps to keep the model, view, and controller code small, focused, and uncluttered.
- `app/models` – The models subdirectory holds the classes that model and wrap the data stored in our application's database. In most frameworks, this part of the application can grow pretty messy, tedious, verbose, and error-prone. Rails makes it very simple!
- `app/view` – The views subdirectory holds the display templates to fill in with data from our application, convert to HTML, and return to the user's browser.
- `app/view/layouts` – Holds the template files for layouts to be used with views. This models the common header/footer method of wrapping views.
- `components` – This directory holds components, tiny self-contained applications that bundle model, view, and controller.
- `config` – This directory contains the small amount of configuration code that your application will need, including your database configuration (in `database.yml`), your Rails environment structure (`environment.rb`), and routing of incoming web requests (`routes.rb`). You can also tailor the behavior of the three Rails environments for test, development, and deployment with files found in the `environments` directory.
- `db` – Usually, your Rails application will have model objects that access relational database tables. You can manage the relational database with scripts you create and place in this directory. This is the parent directory for migration files.
- `doc` – Ruby has a framework, called RubyDoc, that can automatically generate documentation for code you create. You can assist RubyDoc with comments in your code. This directory holds all the RubyDoc-generated Rails and application documentation.
- `lib` – You'll put libraries here, unless they explicitly belong elsewhere (such as vendor libraries).
- `log` – Error logs go here. Rails creates scripts that help you manage various error

logs. You'll find separate logs for the server (server.log) and each Rails environment (development.log, test.log, and production.log).

- `public` – Like the public directory for a web server, this directory has web files that don't change, such as JavaScript files (public/javascripts), graphics (public/images), stylesheets (public/stylesheets), and HTML files (public).
- `script` – This directory holds scripts to launch and manage the various tools that you'll use with Rails. For example, there are scripts to generate code (generate) and launch the web server (server).
- `test` – The tests you write and those that Rails creates for you, all goes here. You'll see a subdirectory for mocks (mocks), unit tests (unit), fixtures (fixtures), and functional tests (functional).
- `tmp` – Rails uses this directory to hold temporary files for intermediate processing.
- `vendor` – Libraries provided by third-party vendors (such as security libraries or database utilities beyond the basic Rails distribution) go here.

Apart from these directories, there will be several files available in the Proverbs directory:

- `Rakefile` – This file is similar to Unix Makefile, which helps with building, packaging and testing the Rails code.
- `Gemfile` – This file includes a list of all gems that you want to include in the project.

Creating our first resource

That welcome page is cool, but it's time to create some pages that are specific to our proverbs app. And we're going to do it using **JUST TWO terminal commands**. You're about to witness the power of convention over configuration.

We will first start by creating a Rails resource.

A rails "resource" is a type of object that you want users to be able to:

- **create** instances of
- **read** data for
- **update** data for, and
- **delete** instances of when they don't want them anymore.

These are the famous **CRUD** operations.

One of the most important resource for our Proverbs app is the: **Proverb** We need to be able to create new proverb, read existing proverb data like the english quotes and their

translations, update or change that data, and delete proverbs when needed.

A **proverb** is defined using the following attributes: *english*, *translation* and *active*. Rails implements the MVC design pattern and uses the three main components **model**, **view** and **controller** to perform these CRUD operations.

It is very common to create each component separately, as we usually like to have extra control over the model, view and controller we create. But, for now we will cheat a bit and create all three at once using a **Rails scaffold**.



rails scaffold is the Rails generator that lets you create a model, view, and controller for a resource simultaneously.

So let's create our first scaffold now. In your terminal, check to make sure you're at your command prompt and make sure there is no rails server running and then type the following:

```
rails generate scaffold Proverb english:string translation:string active:boolean
```

- `rails` rails is the command we are running here
- `generate` subcommand will generate some source code files for us. (remember we ran the `server` subcommand earlier)
- `scaffold` is the type of code we want to generate.
- `Proverb` is the model we want to create a scaffold for.
- `english:string translation:string active:boolean` : attributes

describing the object Proverb. To create the right type of database field, we need to specify the type of data the attribute will hold. This could be a string, an integer, a date, or any type of value the database supports.

After this, we'll be able to create Ruby objects with a class of Proverb, save them to the database and load them in again and display them in our browser.

This command creates a whole bunch of files for you. Among these file, a **migration file** is created. It has a specific name and stored under *db/migrate* in the rails folder tree:

```
db/migrate/today's date\_create_proverb.rb
```

This migration file is responsible for creating the proverb table in the database, in which we will store all the proverbs data that will be collected through the system via the creation of Proverb objects.

But that table hasn't been created yet.

We first runing `rails server` , then reload the browser, an error is thrown

ActiveRecord::PendingMigrationError

Migrations are pending. To resolve this issue, run: `bin/rake db:migrate RAILS_ENV=development`

Extracted source (around line #393):

```
391
392   def check_pending!(connection = Base.connection)
393     raise ActiveRecord::PendingMigrationError if ActiveRecord::Migrator.needs_migration?(connection)
394   end
395
396   def load_schema_if_pending!
```

Rails.root: /home/tarepsh/workspace/sample_app

[Application Trace](#) | [Framework Trace](#) | [Full Trace](#)

Request

Parameters:

None

>>

This is the `ActiveRecord::PendingMigrationError` error.

To fix this, we need to run the migration to create the table here on our development system, and it'll be run again later on your production system to create the same table there.

```
rails db:migrate
```

This is what we get.

```
20200205082634 CreateProverbs: migrating
===== -- create_table(:proverbs)
```


-> 0.0012s

== 20200205082634 CreateProverbs: migrated (0.0013s) =====

- The `CreateProverbs: migrating` shows that the migration is running.
- Below, you will see `create_table(:proverbs)`, and that's the table being created.

Using

The proverbs table has an automatically generated **id** column that's used to look records up, in addition to the ones we specified in the migration.

Migrations for new tables have a call to the `timestamps` method by default, which sets up the **created_at** and **updated_at** columns, which store the date and time a record was created or updated.

Now, if we reload the page, we will get the home page shown successfully. But this only the welcome page, common to all applications. But, we would like to see our `proverb` resource we just created.

In order to access the owner resource, we need to specify a path to the resource. we will add `/proverbs` to the end of the URL.

To check the list of possible paths, we check the routes file using

```
rails routes
```

All the routes created in the application will be displayed.

So, the question is: How are all these details displayed.

Rails receives a `get` request for the `proverbs` path. This request is sent to the `index` method in the ProverbController class.

I want to get a list of the proverbs: I pass it this URL

`http://localhost:3000/proverbs` and I get a list of the proverbs, as you can see. The dispatcher calls the ProverbsController and asks it to execute the action `index`, after looking at the routing table that basically maps the HTTP Verb and the URI pattern into a `controller#action` combination.

So If I want to get all the proverbs, all I need to do is to pass a `GET HTTP Verb` and pass the pattern `proverbs` in the URL. This tells the dispatcher that the request should go to this controller/action combination: `proverbs#index`.

```
# in the controller
def index
  @proverbs=Proverb.all
end
```

In the index action: `@proverbs` : is the instance variable, this bucket or container in which I will store this data in.

And here the Controller is asking the Proverb model to run this `all` method: `Proverb.all`.

THE MODEL: Models are always upper case and singular, and our controllers will be in plural and upper case So Proverb is a model, we will go to the Proverb model and run the `all` Ruby method.

NOW BEFORE GOING TO THE MODEL PROVERB: RUN `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.

```
rails console
```

And type in `Proverb.all`

One thing I would like you to notice, that each time I do `Proverb.all` , it generates some SQL.

Also, you can test the following:

```
Proverb.first
Proverb.last
Proverb.find(5)
p= Proverb.first
p.active=false
p.save
```