Part III

Rails in Depth

In this chapter, you'll see:

- The directory structure of a Rails application
- Naming conventions
- Adding Rake tasks
- Configuration

CHAPTER 19

Finding Your Way Around Rails

Having survived our Depot project, you're now prepared to dig deeper into Rails. For the rest of the book, we'll go through Rails topic by topic (which pretty much means module by module). You've seen most of these modules in action before. We'll cover not only what each module does but also how to extend or even replace the module and why you might want to do so.

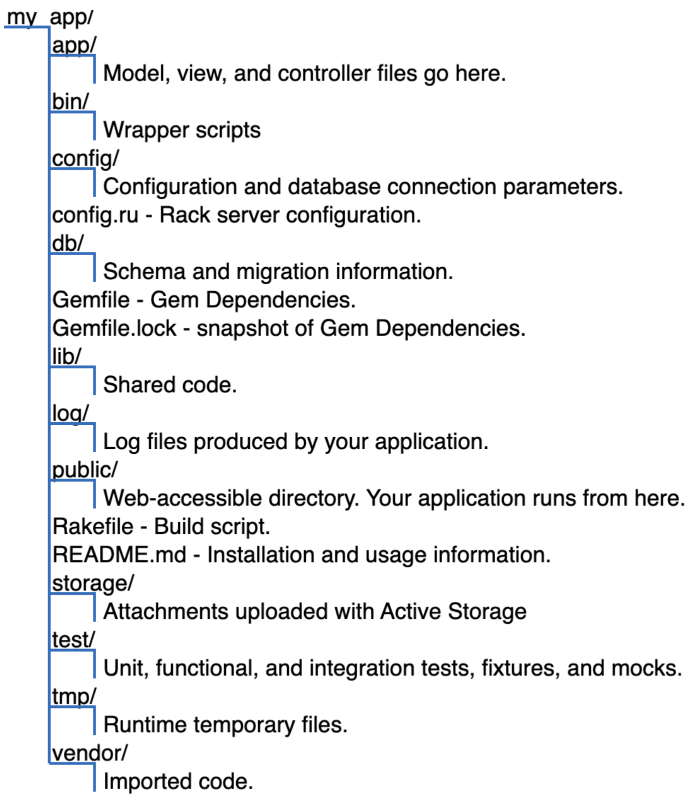
The chapters in Part III cover all the major subsystems of Rails: Active Record, Active Resource, Action Pack (including both Action Controller and Action View), and Active Support. This is followed by an in-depth look at migrations.

Then we're going to delve into the interior of Rails and show how the components are put together, how they start up, and how they can be replaced. Having shown how the parts of Rails can be put together, we'll complete this book with a survey of a number of popular replacement parts, many of which can be used outside of Rails.

We need to set the scene first. This chapter covers all the high-level stuff you need to know to understand the rest: directory structures, configuration, and environments.

Where Things Go

Rails assumes a certain runtime directory layout and provides application and scaffold generators, which will create this layout for you. For example, if we generate *my_app* using the command `rails new my_app`, the top-level directory for our new application appears as shown in the figure on page 292.



```
my_app/
  app/
    Model, view, and controller files go here.
  bin/
    Wrapper scripts
  config/
    Configuration and database connection parameters.
  config.ru - Rack server configuration.
  db/
    Schema and migration information.
  Gemfile - Gem Dependencies.
  Gemfile.lock - snapshot of Gem Dependencies.
  lib/
    Shared code.
  log/
    Log files produced by your application.
  public/
    Web-accessible directory. Your application runs from here.
  Rakefile - Build script.
  README.md - Installation and usage information.
  storage/
    Attachments uploaded with Active Storage
  test/
    Unit, functional, and integration tests, fixtures, and mocks.
  tmp/
    Runtime temporary files.
  vendor/
    Imported code.
```

Joe asks:

So, Where's Rails?

One of the interesting aspects of Rails is how componentized it is. From a developer's perspective, you spend all your time dealing with high-level modules such as Active Record and Action View. There's a component called Rails, but it sits below the other components, silently orchestrating what they do and making them all work together seamlessly. Without the Rails component, not much would happen. But at the same time, only a small part of this underlying infrastructure is relevant to developers in their day-to-day work. We'll cover the parts that *are* relevant in the rest of this chapter.

Let's start with the text files in the top of the application directory:

- `config.ru` configures the Rack Webserver Interface, either to create Rails Metal applications or to use Rack Middlewares in your Rails application. These are discussed further in the Rails Guides.[1]

1. http://guides.rubyonrails.org/rails_on_rack.html

- Gemfile specifies the dependencies of your Rails application. You've already seen this in use when the bcrypt-ruby gem was added to the Depot application. Application dependencies also include the database, web server, and even scripts used for deployment.

  Technically, this file isn't used by Rails but rather by your application. You can find calls to the Bundler[2] in the config/application.rb and config/boot.rb files.

- Gemfile.lock records the specific versions for each of your Rails application's dependencies. This file is maintained by Bundler and should be checked into your repository.

- Rakefile defines tasks to run tests, create documentation, extract the current structure of your schema, and more. Type rake -T at a prompt for the full list. Type rake -D task to see a more complete description of a specific task.

- README contains general information about the Rails framework.

Let's look at what goes into each directory (although not necessarily in order).

A Place for Our Application

Most of our work takes place in the app directory. The main code for the application lives below the app directory, as shown in the figure on page 294. We'll talk more about the structure of the app directory as we look at the various Rails modules such as Active Record, Action Controller, and Action View in more detail later in the book.

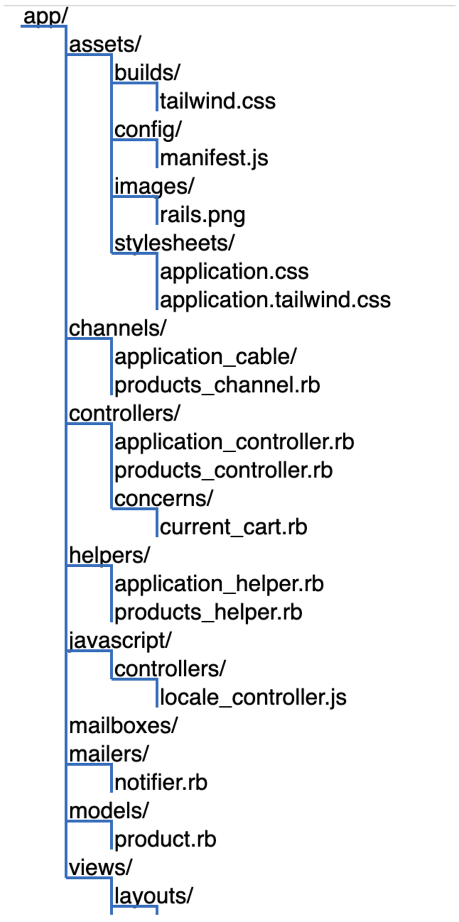
A Place for Our Tests

As we've seen in Iteration B2: Unit Testing of Models, on page 89, Iteration C4: Functional Testing of Controllers, on page 108, and Iteration G3: Testing Our JavaScript Functionality, on page 183, Rails has ample provisions for testing your application, and the test directory is the home for all testing-related activities, including fixtures that define data used by our tests.

A Place for Supporting Libraries

The lib directory holds application code that doesn't fit neatly into a model, view, or controller. For example, you may have written a library that creates PDF receipts that your store's customers can download. These receipts are sent directly from the controller to the browser (using the send_data() method). The code that creates these PDF receipts will sit naturally in the lib directory.

2. https://github.com/bundler/bundler



The lib directory is also a good place to put code that's shared among models, views, or controllers. Maybe you need a library that validates a credit card number's checksum, that performs some financial calculation, or that works out the date of Easter. Anything that isn't directly a model, view, or controller should be slotted into lib.

Don't feel that you have to stick a bunch of files directly into the lib directory. Feel free to create subdirectories in which you group related functionality under lib. For example, on the Pragmatic Programmer site, the code that generates receipts, customs documentation for shipping, and other PDF-formatted documentation is in the directory lib/pdf_stuff.

In previous versions of Rails, the files in the lib directory were automatically included in the load path used to resolve require statements. This is now an option that you need to explicitly enable. To do so, place the following in config/application.rb:

```
config.autoload_paths += %W(#{Rails.root}/lib)
```

Once you have files in the lib directory and the lib added to your autoload paths, you can use them in the rest of your application. If the files contain classes or modules and the files are named using the lowercase form of the class or module name, then Rails will load the file automatically. For example, we might have a PDF receipt writer in the file receipt.rb in the directory lib/pdf_stuff. As long as our class is named PdfStuff::Receipt, Rails will be able to find and load it automatically.

For those times where a library can't meet these automatic loading conditions, you can use Ruby's require mechanism. If the file is in the lib directory, you can require it directly by name. For example, if our Easter calculation library is in the file lib/easter.rb, we can include it in any model, view, or controller using this:

```
require "easter"
```

If the library is in a subdirectory of lib, remember to include that directory's name in the require statement. For example, to include a shipping calculation for airmail, we might add the following line:

```
require "shipping/airmail"
```

A Place for Our Rake Tasks

You'll also find an empty tasks directory under lib. This is where you can write your own Rake tasks, allowing you to add automation to your project. This isn't a book about Rake, so we won't elaborate, but here's a simple example.

Rails provides a Rake task to tell you the latest migration that's been performed. But it may be helpful to see a list of *all* the migrations that have been performed. We'll write a Rake task that prints the versions listed in the schema_migration table. These tasks are Ruby code, but they need to be placed into files with the extension .rake. We'll call ours db_schema_migrations.rake:

rails7/depot_u/lib/tasks/db_schema_migrations.rake

```
namespace :db do
  desc "Prints the migrated versions"
  task :schema_migrations => :environment do
    puts ActiveRecord::Base.connection.select_values(
      'select version from schema_migrations order by version' )
  end
end
```

We can run this from the command line just like any other Rake task:

```
depot> bin/rails db:schema_migrations
(in /Users/rubys/Work/...)
20221207000001
20221207000002
20221207000003
20221207000004
20221207000005
20221207000006
20221207000007
```

Consult the Rake documentation at https://github.com/ruby/rake#readme for more information on writing Rake tasks.

A Place for Our Logs

As Rails runs, it produces a bunch of useful logging information. This is stored (by default) in the log directory. Here you'll find three main log files, called development.log, test.log, and production.log. The logs contain more than just trace lines; they also contain timing statistics, cache information, and expansions of the database statements executed.

Which file is used depends on the environment in which your application is running (and we'll have more to say about environments when we talk about the config directory in A Place for Configuration, on page 297).

A Place for Static Web Pages

The public directory is the external face of your application. The web server takes this directory as the base of the application. In here you place *static* (in other words, unchanging) files, generally related to the running of the server.

A Place for Script Wrappers

If you find it helpful to write scripts that are launched from the command line and perform various maintenance tasks for your application, the bin directory is the place to put wrappers that call those scripts.

This directory also holds the Rails script. This is the script that's run when you run the rails command from the command line. The first argument you pass to that script determines the function Rails will perform:

console
: Allows you to interact with your Rails application methods.

dbconsole
: Allows you to directly interact with your database via the command line.

destroy

Removes autogenerated files created by generate.

generate

A code generator. Out of the box, it will create controllers, mailers, models, scaffolds, and web services. Run generate with no arguments for usage information on a particular generator; here's an example:

```
bin/rails generate migration
```

new

Generates Rails application code.

runner

Executes a method in your application outside the context of the Web. This is the noninteractive equivalent of rails console. You could use this to invoke cache expiry methods from a cron job or handle incoming email.

server

Runs your Rails application in a self-contained web server, using the web server listed in your Gemfile, or WEBrick if none is listed. We've been using Puma in our Depot application during development.

A Place for Temporary Files

It probably isn't a surprise that Rails keeps its temporary files tucked in the tmp directory. You'll find subdirectories for cache contents, sessions, and sockets in here. Generally these files are cleaned up automatically by Rails, but occasionally if things go wrong, you might need to look in here and delete old files.

A Place for Third-Party Code

The vendor directory is where third-party code lives. You can install Rails and all of its dependencies into the vendor directory.

If you want to go back to using the system-wide version of gems, you can delete the vendor/cache directory.

A Place for Configuration

The config directory contains files that configure Rails. In the process of developing Depot, we configured a few routes, configured the database, created an initializer, modified some locales, and defined deployment instructions. The rest of the configuration was done via Rails conventions.

Before running your application, Rails loads and executes config/environment.rb and config/application.rb. The standard environment set up automatically by these files includes the following directories (relative to your application's base directory) in your application's load path:

- The app/controllers directory and its subdirectories
- The app/models directory
- The vendor directory and the lib contained in each plugin subdirectory
- The directories app, app/helpers, app/mailers, and app/*/concerns

Each of these directories is added to the load path only if it exists.

In addition, Rails will load a per-environment configuration file. This file lives in the environments directory and is where you place configuration options that vary depending on the environment.

This is done because Rails recognizes that your needs, as a developer, are very different when writing code, testing code, and running that code in production. When writing code, you want lots of logging, convenient reloading of changed source files, in-your-face notification of errors, and so on. In testing, you want a system that exists in isolation so you can have repeatable results. In production, your system should be tuned for performance, and users should be kept away from errors.

The switch that dictates the runtime environment is external to your application. This means that no application code needs to be changed as you move from development through testing to production. When starting a server with the bin/rails server command, we use the -e option:

```
depot> bin/rails server -e development
depot> bin/rails server -e test
depot> bin/rails server -e production
```

If you have special requirements, such as if you favor having a *staging* environment, you can create your own environments. You'll need to add a new section to the database configuration file and a new file to the config/environments directory.

What you put into these configuration files is entirely up to you. You can find a list of configuration parameters you can set in the Configuring Rails Applications guide.[3]

3. http://guides.rubyonrails.org/configuring.html

Naming Conventions

Newcomers to Rails are sometimes puzzled by the way it automatically handles the naming of things. They're surprised that they call a model class Person and Rails somehow knows to go looking for a database table called people. In this section, you'll learn how this implicit naming works.

The rules here are the default conventions used by Rails. You can override all of these conventions using configuration options.

Mixed Case, Underscores, and Plurals

We often name variables and classes using short phrases. In Ruby, the convention is to have variable names where the letters are all lowercase and words are separated by underscores. Classes and modules are named differently: there are no underscores, and each word in the phrase (including the first) is capitalized. (We'll call this *mixed case*, for fairly obvious reasons.) These conventions lead to variable names such as order_status and class names such as LineItem.

Rails takes this convention and extends it in two ways. First, it assumes that database table names, such as variable names, have lowercase letters and underscores between the words. Rails also assumes that table names are always plural. This leads to table names such as orders and third_parties.

On another axis, Rails assumes that files are named using lowercase with underscores.

Rails uses this knowledge of naming conventions to convert names automatically. For example, your application might contain a model class that handles line items. You'd define the class using the Ruby naming convention, calling it LineItem. From this name, Rails would automatically deduce the following:

- The corresponding database table will be called line_items. That's the class name, converted to lowercase, with underscores between the words, and pluralized.
- Rails would also know to look for the class definition in a file called line_item.rb (in the app/models directory).

Rails controllers have additional naming conventions. If our application has a store controller, then the following happens:

- Rails assumes the class is called StoreController and that it's in a file named store_controller.rb in the app/controllers directory.

- Rails also looks for a helper module named StoreHelper in the file store_helper.rb located in the app/helpers directory.
- It will look for view templates for this controller in the app/views/store directory.
- It will by default take the output of these views and wrap them in the layout template contained in the file store.html.erb or store.xml.erb in the directory app/views/layouts.

All these conventions are shown in the following tables.

Model Naming	
Table	line_items
File	app/models/line_item.rb
Class	LineItem

Controller Naming	
URL	http://../store/list
File	app/controllers/store_controller.rb
Class	StoreController
Method	list
Layout	app/views/layouts/store.html.erb

View Naming	
URL	http://../store/list
File	app/views/store/list.html.erb (or .builder)
Helper	module StoreHelper
File	app/helpers/store_helper.rb

There's one extra twist. In normal Ruby code you have to use the require keyword to include Ruby source files before you reference the classes and modules in those files. Since Rails knows the relationship between filenames and class names, require isn't normally necessary in a Rails application. The first time you reference a class or module that isn't known, Rails uses the naming conventions to convert the class name to a filename and tries to load that file behind the scenes. The net effect is that you can typically reference (say) the name of a model class, and that model will be automatically loaded into your application.

Grouping Controllers into Modules

So far, all our controllers have lived in the app/controllers directory. It's sometimes convenient to add more structure to this arrangement. For example, our store might end up with a number of controllers performing related but disjoint administration functions. Rather than pollute the top-level namespace, we might choose to group them into a single admin namespace.

David says:

Why Plurals for Tables?

Because it sounds good in conversation. Really. "Select a Product from products." And "Order has_many :line_items."

The intent is to bridge programming and conversation by creating a domain language that can be shared by both. Having such a language means cutting down on the mental translation that otherwise confuses the discussion of a *product description* with the client when it's really implemented as *merchandise body*. These communications gaps are bound to lead to errors.

Rails sweetens the deal by giving you most of the configuration for free if you follow the standard conventions. Developers are thus rewarded for doing the right thing, so it's less about giving up "your ways" and more about getting productivity for free.

Rails does this using a simple naming convention. If an incoming request has a controller named (say) admin/book, Rails will look for the controller called book_controller in the directory app/controllers/admin. That is, the final part of the controller name will always resolve to a file called *name*_controller.rb, and any leading path information will be used to navigate through subdirectories, starting in the app/controllers directory.

Imagine that our program has two such groups of controllers (say, admin/*xxx* and content/*xxx*) and that both groups define a book controller. There'd be a file called book_controller.rb in both the admin and content subdirectories of app/controllers. Both of these controller files would define a class named BookController. If Rails took no further steps, these two classes would clash.

To deal with this, Rails assumes that controllers in subdirectories of the directory app/controllers are in Ruby modules named after the subdirectory. Thus, the book controller in the admin subdirectory would be declared like this:

```ruby
class Admin::BookController < ActionController::Base
  # ...
end
```

The book controller in the content subdirectory would be in the Content module:

```ruby
class Content::BookController < ActionController::Base
  # ...
end
```

The two controllers are therefore kept separate inside your application.

The templates for these controllers appear in subdirectories of app/views. Thus, the following is the view template corresponding to this request:

```
http://my.app/admin/book/edit/1234
```

And it will be in this file:

```
app/views/admin/book/edit.html.erb
```

You'll be pleased to know that the controller generator understands the concept of controllers in modules and lets you create them with commands such as this:

```
myapp> bin/rails generate controller Admin::Book action1 action2 ...
```

What We Just Did

Everything in Rails has a place, and we systematically explored each of those nooks and crannies. In each place, files and the data contained in them follow naming conventions, and we covered that too. Along the way, we filled in a few missing pieces:

- We added a Rake task to print the migrated versions.
- We showed how to configure each of the Rails execution environments.

Next up are the major subsystems of Rails, starting with the largest, Active Record.