Extracted from:

Agile Web Development with Rails 5.1

The Pragmatic Bookshelf

Raleigh, North Carolina

The Pragmatic Programmers

Agile Web Development with Rails 5.1

Sam Ruby
David Bryant Copeland
with Dave Thomas

Foreword by
David Heinemeier Hansson

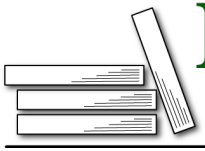
RAILS

Edited by Susannah Davidson Pfalzer

Agile Web Development with Rails 5.1

Sam Ruby
David Bryant Copeland
with Dave Thomas

The Pragmatic Bookshelf
Raleigh, North Carolina



Pragmatic Bookshelf

Our Pragmatic books, screencasts, and audio books can help you and your team create better software and have more fun. Visit us at *https://pragprog.com*.

For sales, volume licensing, and support, please contact *support@pragprog.com*.

For international rights, please contact *rights@pragprog.com*.

Printed in the United States of America.
ISBN-13: 978-1-68050-251-0
Encoded using the finest acid-free high-entropy binary digits.
Book version: B1.0—May 10, 2017

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 are 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 have seen most of these modules in action before. We will 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 are 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. 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 4.

```
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.
    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.
    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 is 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 have 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 6. 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.

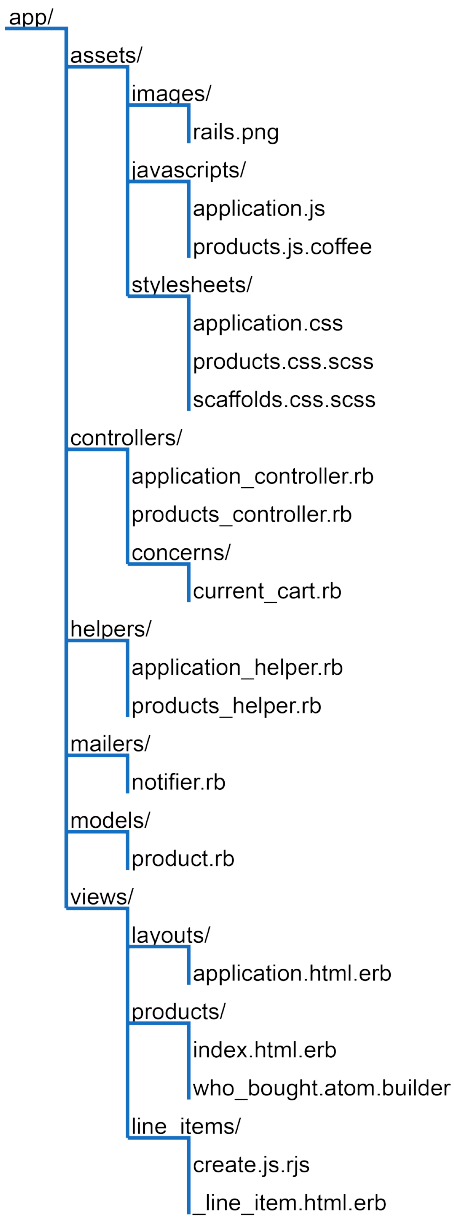
A Place for Our Tests

As we have seen in *Iteration B2: Unit Testing of Models*, on page ?, *Iteration C4: Functional Testing of Controllers*, on page ?, and *the (as yet) unwritten sec.integration.test*, 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 cannot 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 has 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:

rails51/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/...)
20170425000001
20170425000002
20170425000003
20170425000004
20170425000005
20170425000006
20170425000007
```

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