

First step of Roda

y-yagi 著

2018-04-22 版 発行

はじめに

本書の内容

本書は、Roda^{*1}という Web アプリケーション用のライブラリについて紹介した本です。

Roda は Ruby 製のフレームワークです。Ruby で Web アプリケーション用のライブラリといえ
ば、一般的には Ruby on Rails(以降 Rails)^{*2}が使われる事が多いと思います。

実際に、Rails を使うのが適切なケースが殆どでしょう。しかし、Rails は多機能なフレームワ
ークな為、実は使用が適切ではないケースもあると思います。

そんな時に、もし他のフレームワークの事を知っていれば、そのフレームワークを使う事も検討
出来たと思います。

残念ながら、Roda 自体の知名度はあまり高くない印象です。本書を通じて Roda というフレーム
ワークの存在自体が少しでも広まる事を願っています。

対象読者

本書の対象は、既に Ruby、Rails を使った事がある人を想定しています。そのため、Ruby 書き
方や、Ruby に関連するツールについての説明は端折っています。予めご了承ください。

免責事項

本書に記載された内容は、情報の提供のみを目的としています。したがって、本書を用いた開発、
製作、運用は、必ずご自身の責任と判断によって行ってください。これらの情報による開発、製作、
運用の結果について、著者はいかなる責任も負いません。また、本書に記載されいてる情報は、執
筆時点でのものです。 時間経過によって情報が古くなっている可能性がありますのであらかじめご
了承ください

^{*1} <http://roda.jeremyevans.net/>

^{*2} <http://rubyonrails.org/>

第 1 章

Roda について知ろう

1.1 Roda とは

Roda は、Jeremy Evans([@jeremyevans](https://github.com/jeremyevans))^{*1}氏によって作られたライブラリです。Jeremy Evans 氏は、データベースツールキットの Sequel^{*2}や、テンプレートエンジンの Erubi^{*3}の作者でもあります。特に Erubi は、Rails 5.1 から Rails 標準のテンプレートエンジンになっており、名前を聞いた事ある方も多いかもしれません。

最初のバージョン (2014/07/30 リリース) から現在まで、Roda は Jeremy Evans 氏個人によって開発が続けられています。

HP^{*4}やソースコードの README^{*5}では、Roda の事を "Roda is a routing tree web toolkit" と紹介しています。

この紹介の通り、Roda はルーティング機能を提供するライブラリです。Model-View-Controller(以降 MVC) パターンにおける Controller 部分の機能のみを提供しており、Rails のように MVC 全ての機能を提供しているわけではありません。

Ruby 製で同様のライブラリだと、Sinatra^{*6}があります。実際、Roda は Sinatra を参考に作られており、HP 上には Sinatra との比較についての説明が記載されています。Sinatra と Roda の違いについては、1.3 Sinatra と Roda で触れたいと思います。

1.2 Hello World!

早速ですが、まずは Roda にさわってみましょう。ここでは、よくある「Hello, World!」と表示するだけのウェブアプリケーションを作ろうと思います。

まずは、Roda をインストールしましょう。

^{*1} <https://github.com/jeremyevans>

^{*2} <http://sequel.jeremyevans.net/>

^{*3} <https://github.com/jeremyevans/erubi>

^{*4} <http://roda.jeremyevans.net/>

^{*5} <https://github.com/jeremyevans/roda/blob/master/README.rdoc>

^{*6} <http://sinatrarb.com/>

```
$ gem install roda
```

これで準備は完了です。では、実際に Roda を使ってみましょう。

Roda は Rack ベースのライブラリです。そのため、Roda で Web アプリケーションを作るには、Rack アプリを起動する為の設定ファイルが必要になるため、その設定ファイルを作成しましょう。ファイル名は、デフォルトのファイル名である "config.ru" という名前にします。

リスト 1.1: config.ru

```
require "roda"

class App < Roda
  route do |r|
    r.root do
      "Hello World!"
    end
  end
end

run App.freeze.app
```

ファイルが作成出来たら、ファイルがあるのと同じディレクトリで "rackup" コマンドを実行して、アプリケーションを起動してみましょう。

```
$ rackup
```

最後に、ブラウザを起動し、"http://localhost:9292/" にアクセスしてくみてください。"Hello World!" と表示されれば成功です。簡単ですね。

1.3 Sinatra と Roda

色々違いあるのですが、一番大きな違いとしてルーティングの書き方の違いがあげれます。

Sinatra はルーティングをフラットなリストで定義するのに対して、Roda は木構造で定義します。

リスト 1.2: sinatra

```
require 'sinatra/base'

class App < Sinatra::Base
  get '/' do
```

```
# TOP ページ
end

get '/todos/:id' do
  @todo = Todo[params[:id].to_i]
  # TODO 表示
end

post '/todos/:id' do
  @todo = Todo[params[:id].to_i]
  # TODO 更新
end

delete '/todos/:id' do
  @todo = Todo[params[:id].to_i]
  # TODO 削除
end
end
end
```

リスト 1.3: roda

```
require 'roda'

class App < Roda
  plugin :all_verbs
  route do |r|
    r.root do
      # TOP ページ
    end

    r.is 'todos', Integer do |todo_id|
      @todo = Todo[params[:id]]

      r.get do
        # TODO 表示
      end

      r.post do
        # TODO 更新
      end

      r.delete do
        # TODO 削除
      end
    end
  end
end
end
```

一見すると Sinatra の方が URL と処理のマッピングわかりやすく良いのでは、と思われる方もいると思います。それはそれで正しい意見だと思います。しかし、木構造になっている事で、ネームスペースや変数の定義を DRY にする事が出来ます。

これはパスが一階層しかないとあまり恩恵が無いかもしれませんが、パスの階層が深くなればなるだけ、よりわかりやすさに差が出てくると思います。

なお、ルーティングに木構造を使用するライブラリは Roda がはじめてではありません。Cuba^{*7} というライブラリがこの方法をとっており、実際 Roda は Cuba を fork して開発されています^{*8}。

Roda は、Cuba と Sinatra という二つの異なるライブラリの思想や設計に影響を受けて、今のようになっています。

1.4 プラグインシステム

Roda の大きな特徴の一つに、プラグインシステムがあります。Roda は、各種機能をプラグインとして提供するようにして、コアの機能は最小限になるようにしています。例えばテンプレートのレンダリング処理もデフォルトでは使用出来るようになっておらず、明示的にプラグインを利用する必要があります。

プラグインは plugin メソッドを対象のプラグインを読む込みことで出来るようになっています。先ほどのコード例だと、plugin :all_verbs がそうです。all_verbs プラグイン使用する事により、patch や delete のような HTTP リクエストメソッドと同じ名前のメソッドが使用出来るようになります。

これは Rails の原則の一つである"おまかせ"とは真逆で、使用したい料理を自分で明示的に選択する必要があります。そもそもどんな料理があるかがわからない最初のうちは、これは大変な作業かもしれません。

しかしこのシステムにより、不要な機能が読み込まれず、結果ライブラリの動作が高速になったり、何が問題があった際に確認すべきソースが少なくなったりと、少なくない恩恵があります。また、何か自分が独自に機能を追加したい場合も、プラグインとしてその機能を提供することで、簡単に機能を追加出来る、というメリットもあります。

なお、Roda 本体で提供しているプラグインについて、詳細に説明している資料は残念ながら今の所無いようです。しかし、API doc@{fn}{document}を見ると、どのような module があるかが一覧で確認出来るので、そこから詳細を見てみるのがおすすめです。

1.5 パフォーマンス

Roda が HP 上でうたっている特徴の一つに、パフォーマンスがあります。HP 上には、"Roda is the one of the fastest Ruby web frameworks."という記載もあります。中々つよきな発言ですね。

では、実際のパフォーマンスなどなののでしょうか。Roda、及び他のフレームワークのパフォーマンスについて触れているサイトがあるので、それを見てみましょう。

まずは、<https://github.com/luislavena/bench-micro> です。これは、Ruby の Web 用のマイクロフレームワークについてのベンチマークを行っているリポジトリです。アプリは"Hello World!"を返すだけの非常にシンプルなアプリになっています。なお、ベンチマーキングには、wrk^{*9}を使用

^{*7} <http://roda.jeremyevans.net/>

^{*8} Roda の GitHub のリポジトリをみると、"forked from sovereign/cuba"と表示されているのが確認できます

^{*9} <https://github.com/wg/wrk>

しています。

結果は下記の通りです。

リスト 1.4: <https://github.com/luislavena/bench-micro> から引用

Requests/sec		
Framework	Requests/sec	% from best
rack	15839.64	100.00%
watts	13585.65	85.77%
syro	13306.01	84.00%
roda	11776.27	74.35%
cuba	11501.96	72.62%
rack-response	11480.11	72.48%
hobbit	11140.22	70.33%
hobby	10606.80	66.96%
hanami-router	10247.61	64.70%
newark	8896.35	56.17%
rambutan	8466.83	53.45%
plezi	8240.07	52.02%
rackstep	8187.32	51.69%
rack-app	7473.52	47.18%
rails-metal	6598.65	41.66%
flame	5454.46	34.44%
sinatra	3977.30	25.11%
grape	2937.03	18.54%
rails-api	1211.33	7.65%

一位は Rack です。昨今のフレームワークは Rack をベースにしている為、素の Rack が一番速いのは当然ですね。ついで、watts^{*10}、syro^{*11}ときて、roda、という順位になっています。

上位三つと比べると少々劣りますが、それでも十分速いのではないのでしょうか。なお、比較対象にあげられている Sinatra と比べると、3 倍近い性能を出しています。

もう一つ、TechEmpower がおこなっている Web Framework Benchmarks(<https://www.techempower.com/benchmarks/>)を見てみましょう。

これは Ruby に限らず、様々なプログラミング言語の Web アプリケーションのフレームワークのパフォーマンス比較を行い、その結果を表示しているサイトです。ベンチマークは定期的を取得するようになっており、最新の結果 (Round 15 / 2018-02-14) では、26 のプログラミング言語の 158 ものフレームワークがベンチマークの対象になっています。

ベンチマークは複数のパターン (JSON を返す、データベースに対して一つ SQL を実行する、等々)、データベースも複数パターン (MySQL、PostgreSQL、MongoDB、等々)、アプリケーションサーバも当然複数パターン (Puma、unicorn、等々) で結果を取得するようになっており、大変多岐に渡っています。

結果の全てを説明するのは紙面の都合上無理なので、ここでは Ruby の結果だけみていきましょう。なお、色々面白い結果になっていると思うので、是非ともサイト上で結果の詳細を見てみる

^{*10} リソース指向のフレームワーク <https://github.com/pete/watts>

^{*11} ルータイブラリ <http://soveran.github.io/syro/>

ことをおすすめします。

まずは、JSON レスポンスの結果です。

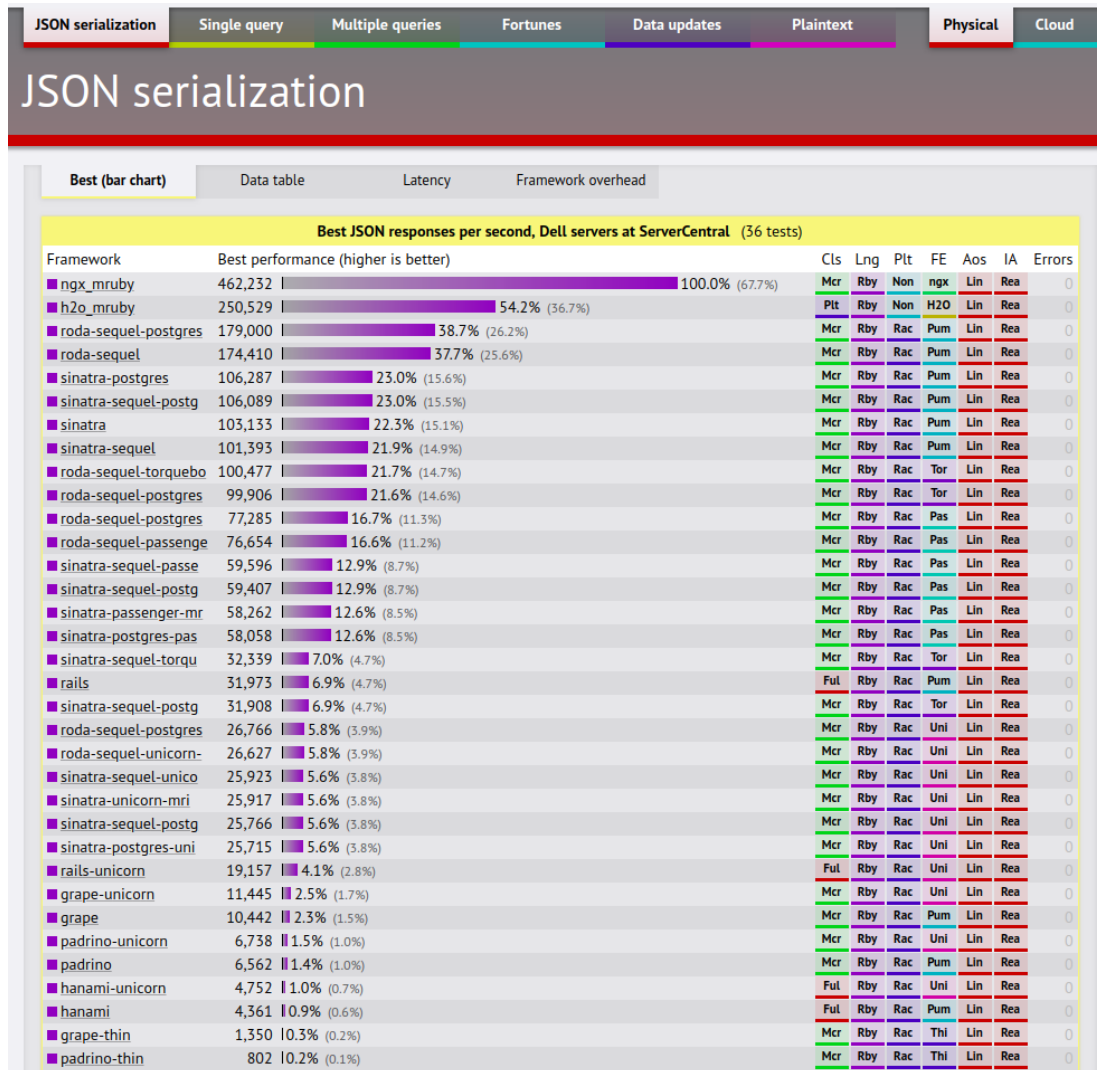


図 1.1: TechEmpower Framework Benchmarks(Ruby/JSON)

mruby と CRuby の結果を混ぜるのはそもそもどうなのだろう、という意見もあるかと思いますが、そこは一旦置いておいてやはり mruby + HTTP サーバが圧倒的に速いですね。mruby について、CRuby では roda + Sequel + PostgreSQL の組み合わせが三位になっています。ついで四位も roda + Sequel(こちらはデータベースが MySQL) です。なお、どちらもアプリケーションサーバは Puma です。そしてその次に Sinatra + PostgreSQL の組み合わせが来ています。先ほどの bench-micro ほどの差は出ていないですが、それでも 1.7 倍程度高速なようです。

もう一つ、データベースに対して一つ SQL を実行した場合の結果も見してみましょう。

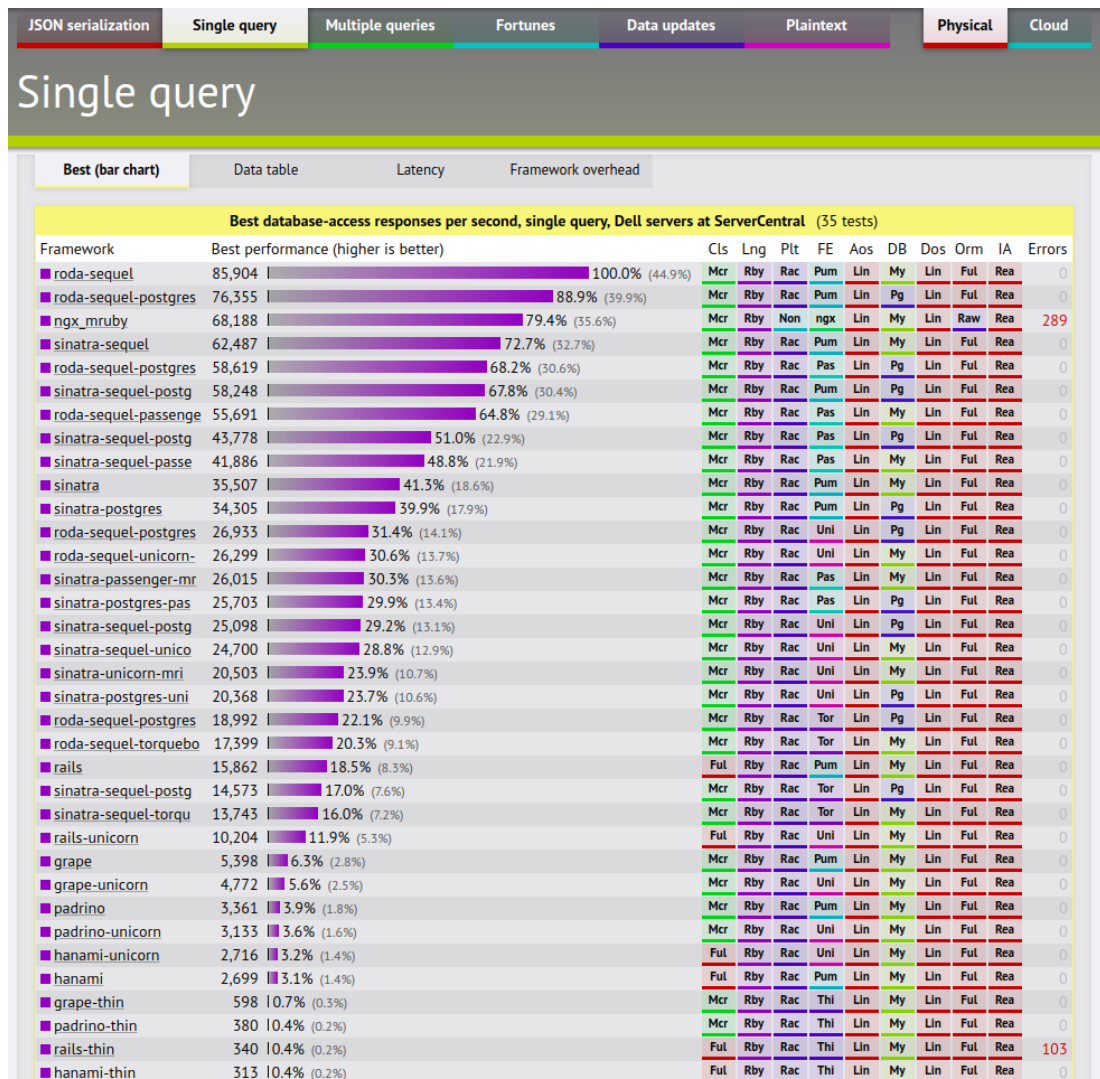


図 1.2: TechEmpower Framework Benchmarks(Ruby/Single query)

roda + Sequel の組み合わせが一位、二位という結果になりました。JSON の結果より差は出なくなっていますが、やはり Sinatra よりは速い、という結果になっています。これらの結果を見る限り、"Roda is the one of the fastest Ruby web frameworks" というのは、そこまで大げさではないように思えますね。

第2章

Roda でアプリケーションを作ってみよう

2.1 前準備

今回のアプリケーションではデータベースには PostgreSQL を使用します。バージョンは 9.2 以上であれば OK です。各自環境に合わせた方法でインストールをお願いします。

PostgreSQL のインストールが完了したら、アプリケーションで使用するデータベースとユーザーを作成しましょう。下記コマンドを PostgreSQL のユーザで実行してください。

```
$ createuser -U postgres my_todo
$ createdb -U postgres -O my_todo my_todo_development
```

これで前準備完了です。

2.2 アプリケーション準備

実際に Roda を使って簡単な Todo アプリケーションを作ってみましょう。

Roda は Rails における `rails new` コマンドのようなスケルトンを作成する為の仕組みがありません。全て自分でセットアップする必要があります。

とはいえ、ゼロからセットアップをするのは、慣れないうちは大変です。幸い、作者である Jeremy Evans 氏が Roda/Sequel を使用したアプリケーションを作成する為のスケルトンアプリケーションを公開してくれているので、今回はそちらを使用しようと思います。なお、今回は、そのスケルトンに合わせて O/R マッパーに Sequel を使用します。Sequel の使い方については、適時説明を入れていきますので、Sequel が使った事が無くても大丈夫です。

スケルトンアプリケーションは <https://github.com/jeremyevans/roda-sequel-stack> にありますので、まずは、`git clone` してアプリケーションをダウンロードしましょう。

```
$ git clone https://github.com/jeremyevans/roda-sequel-stack.git
```

今回作成するアプリケーションは"MyTodo"という名前にしようと思いますので、ディレクトリ名も"my_todo"に変更しましょう。

```
$ mv roda-sequel-stack my_todo
$ cd my_todo
```

このスケルトンアプリケーションでは、アプリケーションサーバが指定されておらず、使用されるアプリケーションサーバがユーザの環境に依存するようになっています。今回は Puma を使用したいと思います。Gemfile の rack-unreloader の下に puma を追加してください。

リスト 2.1: Gemfile

```
source 'https://rubygems.org'

gem 'rack_csrf'
gem 'sass'
gem 'tilt', '>= 2'
gem 'erubi'
gem 'roda', '>= 3'
gem 'sequel', '>= 5'
gem 'sequel_pg'
gem 'rack-unreloader'
gem 'puma'

gem 'capybara'
gem 'minitest', '>= 5.7.0'
gem 'minitest-hooks', '>= 1.1.0'
```

ついでなので、使用している gem で、あまり普段使用しないであろう gem について簡単に説明したいと思います。

rack_csrf

CSRF 対策用 Rack ミドルウェア。Roda の csrf プラグインはこのライブラリを使用して実装されています。

tilt

各種テンプレートエンジンへの汎用インターフェイスを提供してくれるライブラリ。Rack のテンプレートレンダリングプラグインはこのライブラリを使用して実装されています。

sequel_pg

Sequel の行フェッチ処理を C で実装したライブラリ。これを使うことにより、素の Sequel より SELECT 処理が高速に行えるようになっています。Jeremy Evans 氏が作成した公式の

ライブラリです。なお、PostgreSQL 版のみ存在し、MySQL 版はありません。

rack-unreloader

アプリケーションのファイルが更新された際に、更新されたファイルを再読み込みしてくれるライブラリ。このライブラリにより、ファイルを変更した際にいちいちアプリケーションサーバを再起動する必要がなくなります。

mintiest-hooks

Minitest で `around/around(:all) hooks` を使用出来るようにする為のライブラリです。なお、Jeremy Evans 氏は普段 Minitest^{*1}を使っているようで、このスケルトンアプリケーションでも Minitest を使うようになっています。

puma を追加したら、`bundle install` で gem のインストールを行いましょう。

```
$ bundle install
```

最後に、`setup` 用の rake タスクを実行します。このタスクでは、データベースの接続先や、cookie のキーを引数に指定されたアプリケーション名に書き換えてくれます。

```
$ rake "setup[MyTodo]"
```

これで準備完了です。

では、アプリケーションを起動してみましょう。`dev_up` タスクでマイグレーションの実行、及び、`rackup` コメントでアプリケーションを起動してください。

```
$ rake dev_up
$ rackup
```

`http://localhost:9292/` にブラウザでアクセスして、`"Hello World!"`が表示されれば、起動成功です。

2.3 アプリケーション構成

ここでは、スケルトンアプリケーションの構成についてみていきましょう。
ファイル構成は下記のようになっているかと思います。

^{*1} Roda や Sequel、Erubi 自体のテストが Minitest で書かれています。

```
.
  Gemfile
  Gemfile.lock
  README.rdoc
  Rakefile
  app.rb
  assets
    css
      app.scss
  config.ru
  db.rb
  migrate
    001_tables.rb
  models
    model1.rb
  models.rb
  routes
    prefix1.rb
  spec
    minitest_helper.rb
    model
      model1_spec.rb
      spec_helper.rb
    web
      prefix1_spec.rb
      spec_helper.rb
  views
    index.erb
    layout.erb
```

主要なファイルについて詳細をみていきましょう。

Rakefile

まずは、Rakefile です。

リスト 2.2: Rakefile

```
# Migrate

migrate = lambda do |env, version|
  ENV['RACK_ENV'] = env
  require_relative 'db'
  require 'logger'
  Sequel.extension :migration
  DB.loggers << Logger.new($stdout)
  Sequel::Migrator.apply(DB, 'migrate', version)
end

desc "Migrate test database to latest version"
task :test_up do
  migrate.call('test', nil)
end
```

```
desc "Migrate test database all the way down"
task :test_down do
  migrate.call('test', 0)
end

desc "Migrate test database all the way down and then back up"
task :test_bounce do
  migrate.call('test', 0)
  Sequel::Migrator.apply(DB, 'migrate')
end

desc "Migrate development database to latest version"
task :dev_up do
  migrate.call('development', nil)
end

desc "Migrate development database to all the way down"
task :dev_down do
  migrate.call('development', 0)
end

desc "Migrate development database all the way down and then back up"
task :dev_bounce do
  migrate.call('development', 0)
  Sequel::Migrator.apply(DB, 'migrate')
end

desc "Migrate production database to latest version"
task :prod_up do
  migrate.call('production', nil)
end

# Shell

irb = proc do |env|
  ENV['RACK_ENV'] = env
  trap('INT', "IGNORE")
  dir, base = File.split(FileUtils::RUBY)
  cmd = if base.sub!(/\Aruby/, 'irb')
    File.join(dir, base)
  else
    "#{FileUtils::RUBY} -S irb"
  end
  sh "#{cmd} -r ./models"
end

desc "Open irb shell in test mode"
task :test_irb do
  irb.call('test')
end

desc "Open irb shell in development mode"
task :dev_irb do
  irb.call('development')
end

desc "Open irb shell in production mode"
```

```
task :prod_irb do
  irb.call('production')
end

# Specs

spec = proc do |pattern|
  sh "#{FileUtils::RUBY} -e 'ARGV.each{|f| require f}' #{pattern}"
end

desc "Run all specs"
task default: [:model_spec, :web_spec]

desc "Run model specs"
task :model_spec do
  spec.call('./spec/model/*_spec.rb')
end

desc "Run web specs"
task :web_spec do
  spec.call('./spec/web/*_spec.rb')
end
```

Rakefile には、大きく分けて三つのタスクが定義されています。データベースマイグレーション用のタスク、コンソール用のタスク、および、テスト用のタスクです。

まず、データベースマイグレーション用のタスクです。Sequel では、データベースマイグレーション用の Rake タスク (Rails における `db:migrate` や `db:rollback` が提供されていない為、自前でタスクを準備しています。環境名 + `up` / `down` でタスクが実行出来るようになっています。例えば、`development` 環境の `migrate` を実行したい場合は、`dev_up` を実行すれば OK です。なお、`down` タスクは、`db:rollback` と異なり、最初のマイグレーションまで `rollback` されてしまうので、その点だけご注意ください。

次に、コンソール用のタスクです。コードを見て頂くとわかるとおり、基本的には `irb` を実行しているだけです。`RACK_ENV` を指定しているのと、`models` 配下のファイルを読み込むようになっているので、`irb` 起動後、すぐ `model` が使用出来るようになっています。

最後に、テスト用のタスクです。`model` 用のテストと `web` (Capybara を使ったテスト) 用のテストでタスクが別れています。なお、`"spec"` という名前が少し紛らわしいかもしれませんが、Gemfile の箇所でも説明した通り、テストは Minitest を使うようになっています。Minitest の BDD 風にテストを書くためのライブラリである、`minitest/spec` を使っている為、`"spec"` という名前になっています。

今回は紙面の都合上、テストについての説明は省いていますが、Roda も Sequel もテストの為の特別な機能を提供している訳ではないので、Minitest、Capybara の機能だけを使用してテストは書かれています。そのため、Minitest を RSpec に置き換えても、特に問題無くテストは書けます。

config.ru

続いています、`config.ru` です。

最初に書いた Hello World を表示するアプリケーションでは、config.ru にメインとなるコードを記載しましたが、このスケルトンアプリケーションでは、メインのコードは app.rb に記載するようにして、config.ru ではそのファイルを読み込むようになっています。

リスト 2.3: config.ru

```
dev = ENV['RACK_ENV'] == 'development'

if dev
  require 'logger'
  logger = Logger.new($stdout)
end

require 'rack/unreloader'
Unreloader = Rack::Unreloader.new(subclasses: %w'Roda Sequel::Model',
                                  logger: logger, reload: dev){ MyTodo }

require_relative 'models'
Unreloader.require('app.rb'){ 'MyTodo' }
run(dev ? Unreloader : MyTodo.freeze.app)
```

ここでは、development 環境用に特別な処理として、STDOUT にログを出すようにしているのと、rack-unreloader を使ってコードが変更されたら再読み込みされるようにしています。

app.rb

最後に、app.rb をいき見てみましょう。このファイルにメインの処理が記載されています。

リスト 2.4: app.rb

```
require_relative 'models'

require 'roda'
require 'tilt/sass'

class MyTodo < Roda
  plugin :default_headers,
    'Content-Type'=>'text/html',
    'Content-Security-Policy'=>"default-src 'self';
    style-src 'self' https://maxcdn.bootstrapcdn.com;",
    #'Strict-Transport-Security'=>'max-age=16070400;',
    # Uncomment if only allowing https:// access
    'X-Frame-Options'=>'deny',
    'X-Content-Type-Options'=>'nosniff',
    'X-XSS-Protection'=>'1; mode=block'

  # Don't delete session secret from environment in development mode as it breaks reloading
  session_secret = ENV['RACK_ENV'] == 'development' ? ENV['MY_TODO_SESSION_SECRET']
    : ENV.delete('MY_TODO_SESSION_SECRET')
  use Rack::Session::Cookie,
    key: '_MyTodo_session',
    #secure: ENV['RACK_ENV'] != 'test', # Uncomment if only allowing https:// access
    :same_site=>:lax, # or :strict if you want to disallow linking into the site
```



```
secret: (session_secret || SecureRandom.hex(40))

plugin :csrf
plugin :flash
plugin :assets, css: 'app.scss', css_opts: {style: :compressed, cache: false},
        timestamp_paths: true
plugin :render, escape: true
plugin :multi_route

Unreloader.require('routes'){

  route do |r|
    r.assets
    r.multi_route

    r.root do
      view 'index'
    end
  end
end
```

まずは、default_headers プラグインを使用してヘッダーを指定しています。指定しているのは、Content-Security-Policy や X-Frame-Options 等の、主にセキュリティに関するヘッダーです。

次に、session の指定、及び、cookie を使用出来るようにする為に、Rack::Session::Cookie ミドルウェアを Rack のスタックに追加しています。

次に、各種プラグインのロード処理です。csrf は CSRF 対策、assets は asset ファイル (CSS 及び JavaScript) のレンダリング、render はテンプレートのレンダリングの為のプラグインです。

flash は、リクエスト間でデータを保持する為の仕組みです。Rails の flash^{*2}と同様の機能です。

multi_route は、複数の名前付きルートを作成出来るようにするプラグインです。通常、ルーティングの大本の定義である root は一つしか定義出来ません。しかし、multi_route プラグインを使用する事により、この root を複数定義するように出来ます。これにより、例えば、API に関するルーティングは別ファイルにする、ということが可能になっています。このスケルトンアプリケーションでは、routes ディレクトリ配下にルーティングファイルが配置されるようになっています。そのため、その次の行で、rack-unreloader で routes ディレクトリのファイルが reload 対象になるようにしています。

route からが実際のルーティングの定義です。まず、assets、及び、multi_route メソッドを呼び出して、asset ファイルのレンダリング、複数 route の読み込みが行われるようにしています。

最後に、view 'index' で、トップページにアクセスした際に、index テンプレートの中身が表示されるようにしています。

^{*2} <http://api.rubyonrails.org/classes/ActionDispatch/Flash.html>

First step of Roda

2018 年 4 月 22 日 初版第 1 刷 発行
著 者 y-yagi
