

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 製のライブラリです。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}が使われています。

しかし、要件によっては、実は Rails が適切ではないケースもあるのではないかと考えています。

そんな時に、もし他のライブラリの事を知っていれば、そのライブラリを使う事も検討出来るでしょう。

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 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

本書は、既に Ruby、Rails を使った事がある人を対象としています。そのため、Ruby や Rails 自体の説明や、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

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

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

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。

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、名前を聞いた事がある方もいるでしょう。

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います。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

*¹ <https://github.com/jeremyevans>

*² Rails における Active Record 相当のライブラリ。O/R マッパーやデータベースのマイグレーションの為に機能が提供されている。 <http://sequel.jeremyevans.net/>

*³ ERB テンプレートを表示する為のテンプレートエンジン。 <https://github.com/jeremyevans/erubi>

*⁴ <http://roda.jeremyevans.net/>

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

*⁶ <http://sinatrarb.com/>

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」で触れます。

1.2 Hello World!

違いについて説明する前に、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ります。

まずは、Roda をインストールしましょう。なお、執筆時点で最新のバージョンである 3.6.0 を使用します。

```
$ gem install roda
Successfully installed roda-3.6.0
1 gem installed
```

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/> にアクセスしてみてください。"Hello World!"と表示されれば成功です。

コードについては、恐らくクラス名やメソッド名から何をしている処理なのかわかるかもしれませんが、簡単に説明します。

Roda

Roda のコアとなるクラス。このクラスを継承したクラスで処理を定義する必要があります。

route

ここで定義したブロックがリクエスト処理の際に実行されます。ブロックは `Rack::Request` のサブクラスのインスタンスで実行されます。

root

リクエストのパスが `/`、かつ、`GET` リクエストの場合に実行されるブロック。

freeze

アプリケーションの設定やミドルウェアを `freeze` する為のメソッド。`freeze` しておくことで、実行時に設定やミドルウェアが予期せず変更されてしまうことを検知出来ます。プロダクションやテスト環境では、`freeze` しておくことが推奨されています。

1.3 Sinatra と Roda

色々と違いあるのですが、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られます。

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
```

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
    end
  end
end
```

```
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、と感じられる方もいるでしょう。それはそれで正しい意見です。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のではないかと筆者が考えています。

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 2 つの異なるライブラリの思想や設計に影響を受けて、今のような形になっています。

1.4 プラグインシステム

Roda の大きな特徴の 1 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していて、コアの機能は最小限になっています。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

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

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

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 1 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ライブラリの動作が高速になります。また、何か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ます。

なお、Roda 本体で提供しているプラグインについては、`doc@{fn}{document}` にまとまっています。Roda 本体だけでどういう機能が提供されているか気になる方は、一度チェックしてみてください。

1.5 パフォーマンス

Roda が HP 上でうたっている特徴の 1 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。中々強気な発言ですね。

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%

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

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 3 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 1 つ、TechEmpower 社がおこなっている 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に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 1 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。

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

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

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。

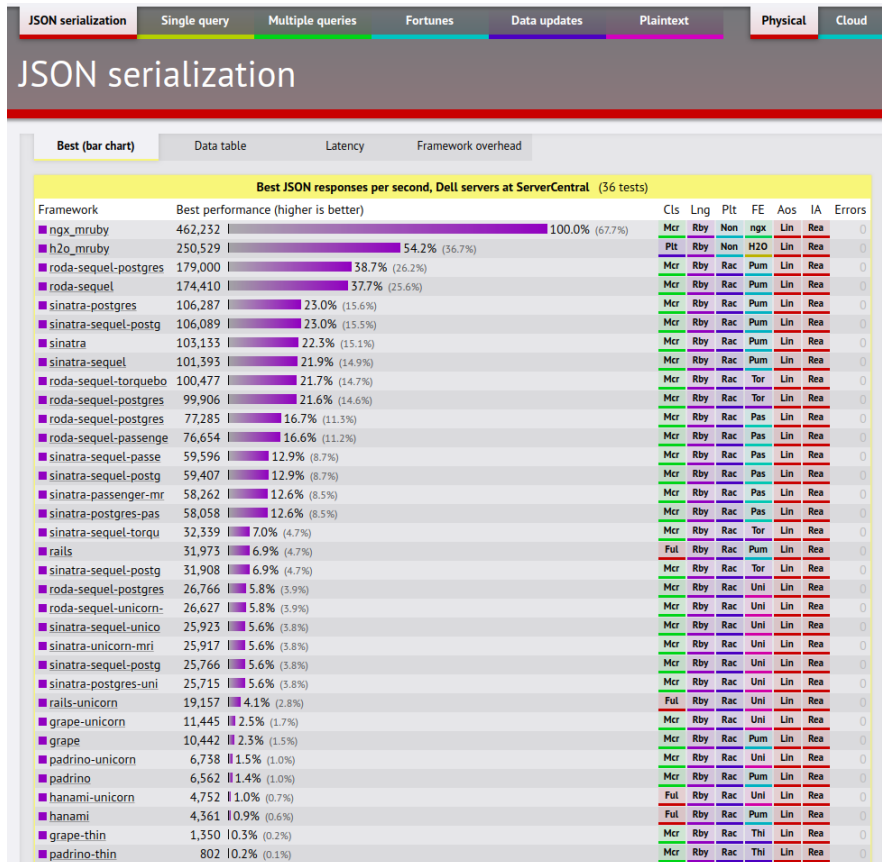


図 1.1: TechEmpower Framework Benchmarks(Ruby/JSON)

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

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。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 1 つ、データベースに対して 1 つ SQL を実行した場合の結果も見えてみましょう。

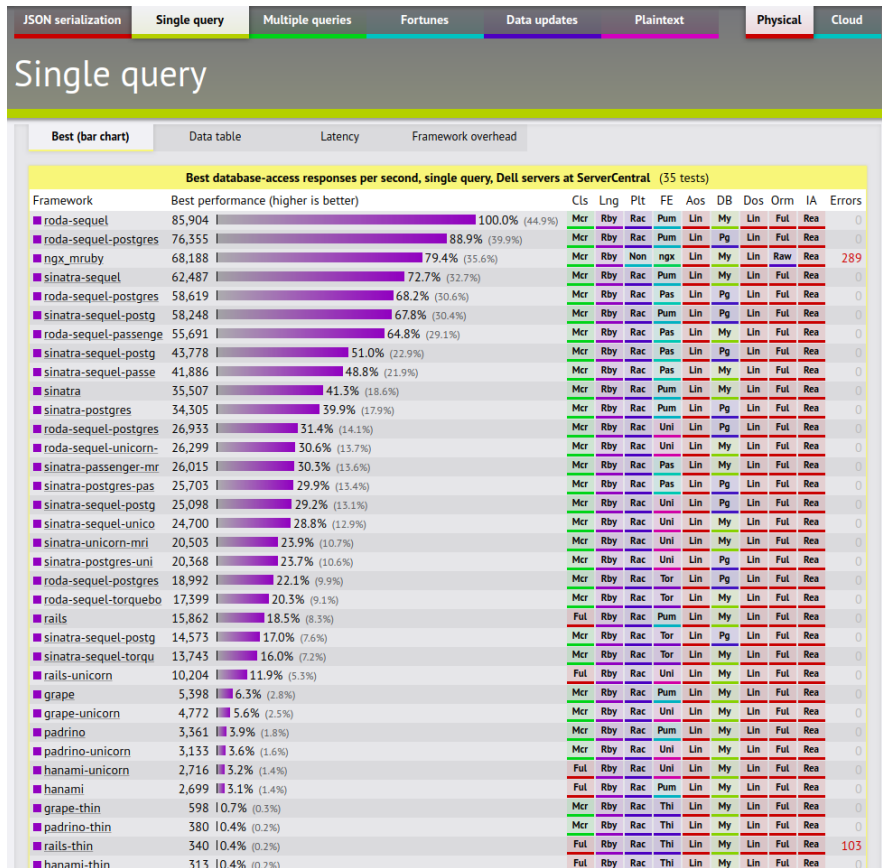


図 1.2: TechEmpower Framework Benchmarks(Ruby/Single query)

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きていますが、やはり Sinatra より速い、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。

コラム: Roda という名前について

Roda という名前は、どういう意図でつけられたのでしょうか？ 辞書を調べてみても、Roda という単語は存在しません。では何かの頭文字をとってつけたのかというと、そうでは無いようです。

同じ疑問を持った方がいて、Hacker News の、Roda に関するスレ (<https://news.ycombinator.com/item?id=8152403>) に、質問が記載され、それに対して Jeremy Evans 氏本人からの回答があります。回答は下記の通りです。

The name Roda comes from the Ys video game series, in which the Roda trees play a small role. I'm a huge Ys fan.

(<https://news.ycombinator.com/item?id=8155027> から引用)

という訳で、日本ファルコムのイースというゲームに出てくるロダの樹から引用したとのことです。ロダの樹は巨大な二本の老木です。恐らくルーティングが木構造になっていることから、連想したのだと思われます。因みに、今は流石になくなりましたが、昔は Roda の HP にイースのゲーム画面のスクリーンショットが貼られていたりしました。

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。なお、サンプルのコードが https://github.com/y-yagi/my_todo にあります。必要に応じて参考にしてください。

第2章 Roda でアプリケーションを作ってみよう 2.2 アプリケーション準備

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。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りません。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'
```

ここで、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。

minitest-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。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。

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

第 2 章 Roda でアプリケーションを作ってみよう 2.3 アプリケーション構成

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。

```
.  
  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 3 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

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ありません。`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 2.14: 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
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。

続いて、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 1 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

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

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。

2.4 テーブルの作成

ここでは、`Todo` を格納する為のテーブルを作成しましょう。

その前に、サンプルとして定義されているテーブルはもう不要なので、先に `dev_down` タスクを使ってテーブルを削除してしまいましょう。

```
$ rake dev_down
```

また、`models/model1.rb` も不要です。こちらも合わせて削除してしまいましょう。

```
$ rm models/model1.rb
```

では改めて、`Todo` を格納する為のテーブルを作成しましょう。

`Todo` の内容と、締切日を設定出来るテーブルを作成したいと思います。`migrate/001_tables.rb` を、下記の内容に変更してください。元々の記載されいる内容は全て削除してしまって大丈夫です。

リスト 2.5: `001_tables.rb`


```
Sequel.migration do
  change do
    create_table(:todos) do
      primary_key :id
      String :content, null: false
      DateTime :deadline
    end
  end
end
```

Rails のマイグレーションに慣れている方であれば、恐らく見ただけで何をしているか何となくわかるかとは思いますが、簡単に説明したいと思います。

まず、Sequel のマイグレーションは全て `Sequel.migration` ブロックの中で実行する必要があるため、`Sequel.migration` を使用しています。

次に、`up` / `down` どちらで実行されるかを定義します。ただ、リバーシブルなマイグレーションであれば、`change` を使うことで、自動で `up` / `down` 時に適切なメソッドを使用するようになります。`create_table` はリバーシブルなメソッド (`down` では `drop_table` が呼ばれる) 為、ここでは `change` を使用します。

最後に、`create_table` を使用して実際に作成したテーブルを作成します。`create_table` ブロックの中でカラムを定義します。カラムは、型、カラム名、オプションという順番で定義します。

これでテーブル作成のようなマイグレーションファイルが出来たので、再度 `rev_up` タスクを実行します。

```
$ rake dev_up
```

テーブルが出来たので、そのテーブルを操作する為の `model` も準備します。`models` ディレクトリ配下に、`todo.rb` というファイル名で `model` を作成します。ファイルの中身は下記の通りです。

リスト 2.15: `todo.rb`

```
class Todo < Sequel::Model
end
```

親クラスが `Sequel::Model` になっているだけで、Active Record を使っているのと変わらないですね。

これでテーブル、及び、そのテーブルを操作する為の model の準備が出来ました。次に進みましょう。

2.5 画面の作成

ここから実際に Tood 機能を作っていきます。今回作成するアプリケーションでは画面は一ページだけで、一ページで、Tood の表示・作成を出来るようにしたいと思います。

Todo の表示からいきたいと思います。

まずは、サーバ側のコードから対応していきましょう。app.rb を編集して、トップページで Todo の一覧を表示出来るようにします。

リスト 2.14: app.rb

```
r.root do
  @todos = Todo.all
  view 'index'
end
```

Sequel は Active Record 同様に `all` メソッドで全ての値を取得出来ます。これで Todo の一覧を取得出来るようになったので、view 側でその値を表示するようにしましょう。

リスト 2.12: view/index.rb

```
<h1>Todo</h1>

<table class='table'>
  <thead>
```

```
<tr>
  <th>内容</th>
  <th>期限</th>
  <th></th>
</tr>
</thead>

<tbody>
  <% @todos.each do |todo| %>
    <tr>
      <td><%= todo.content %></td>
      <td><%= todo.deadline %></td>
    </tr>
  <% end %>
</tbody>
</table>
```

普通に HTML ですね。これで一覧の表示が出来るようになりました。

続いて、Todo の登録を出来るようにしましょう。こちらもサーバ側のコードから修正しましょう。

"/todos"に対して POST メソッドでデータが送信されたら、送信されたデータを Todo として登録するようにします。t.root 配下に、下記コードを追加してください。

リスト 2.14: app.rb

```
r.post 'todos' do
  Todo.create(r.params['todo'])
  flash[:notice] = 'Todo を作成しました'
  r.redirect '/'
end
```

続けて view 側です。views/index.erb の一番下に、下記コードを追加してください。

リスト 2.12: view/index.rb

```
<hr>

<h4>登録</h4>
<form action='/todos' method='post'>
  <%= csrf_tag %>
  <div class='form-group'>
    <input type='text' name='todo[content]' id='todo_content' required='true', placeholder=
内容'>
    <input type='datetime-local' name='todo[deadline]' id='todo_deadline'>
  </div>

  <input type='submit' name='commit' value='作成', class='btn btn-primary'>
</form>
```

Roda には Rails のような view ヘルパーメソッドはありません^{*3}。そのため、フォーム等も普通に HTML を記載する必要があります。

ただの HTML なので、特に説明はいらないかと思います。一点、CSRF 対策用のタグを `csrf_tag` メソッドを使用して生成しているので、そこだけ注意してください。

最後に、Todo の削除を出来るようにしましょう。

Rails であれば、`link_to 'Destroy', xxx, method: :delete` でおしまい、なのですが、残念ながら view ヘルパーメソッドがない Roda ではそんな簡単にはいきません。多少手間であるのですが、削除用の view を追加し、そのページに削除ボタンを表示する、というアプローチをとりたいと思います。まずはサーバ側です。

リスト 2.14: `app.rb`

```
r.on 'todo', Integer do |id|
  @todo = Todo[id]

  r.get do
    view 'delete'
  end
end
```

^{*3} Jeremy Evans 氏が form 用のライブラリを作成しており、そちらを使用すると、メソッドを使用して form を作成する事も可能です。 <https://github.com/jeremyevans/forme>

```
r.post 'destroy' do
  @todo.destroy
  flash[:notice] = 'Todo を削除しました'
  r.redirect '/'
end
end
```

削除ページの表示、及び、実際に削除する処理の2つルーティングを追加しています。どちらも `/todo/:id` というパスから始まるようにするために、まず `on` メソッドでブランチ (ここから `"todo"` パスのルーティングになります、という宣言) を作成します。また、パラメータとして `Todo` の ID が必須になるので、パラメータに `Integer` の値がくることを宣言するように、`on` メソッドの引数に `Integer` を指定しています。

あとは、`/todo/:id` に `GET` メソッドでアクセスしたら削除のページを表示するよう、`/todo/:id/destroy` に `POST` メソッドでアクセスしたら実際に削除処理を行うようにそれぞれしています。サーバ側の処理はこれで完了です。

続けて `view` 側です。まずは、`views/index.erb` に削除ページへのリンクを追加したいと思います。`Todo` の一覧を表示している `table` に、以下のように `a` タグを追加してください。

リスト 2.12: `view/index.rb`

```
<tbody>
  <% @todos.each do |todo| %>
    <tr>
      <td><%= todo.content %></td>
      <td><%= todo.deadline %></td>
      <td><a href='<%= "/todo/#{todo.id}" %>'>削除</a></td>
    </tr>
  <% end %>
</tbody>
```

最後に、削除ページの `view` を追加します。`delete.erb` という名前のファイルを、`view` ディレクトリ配下に追加してください。

リスト 2.13: `view/delete.rb`

```

<h4>削除</h4>
<form action=' <%= "/todo/#{@todo.id}/destroy" %>' method='post'>
  <%= csrf_tag %>
  <table class="table table-bordered table-striped">
    <tr class="string required">
      <td><label class="label-before" for="todo_content">内          容
    </label></td>
      <td><span><%= @todo.content %></span></td>
    </tr>
    <tr class="string required">
      <td><label class="label-before" for="todo_deadline">期      限
    </label></td>
      <td><span><%= @todo.deadline %></span></td>
    </tr>
  </table>
  <input type='submit' name='delete' value='          削
除', class='btn btn-danger'>
</form>

```

これで削除も出来るようになりました。

2.6 JSON 出力

ここまでで、簡単な Todo アプリケーションが動くようになりました。実際に動くアプリを書いてみての感想は色々あるかと思いますが、やはり Rails に比べると view を書くのが大変、と思われた方が多いのでは無いでしょうか？ 筆者もそう思います。

Sinatra の代替として使う分には良いかもしれませんが、やはり view を書くのが大変だと、Rails の代わりに使いたい、と思う方は多くはいらっしゃらないと思います。

そこで、筆者が個人的に Roda が活躍出来るのではないかと考えているのは、API 開発です。

昨今、ブラウザや JavaScript ライブラリの進化に伴い、サーバ側は API を返すだけで view は全て JavaScript を実装する、というアプリケーションも多くあります。また、スマートフォンアプリケーション用に API が必要な場合も、view は不要になることがあるでしょう。

そのような、単純に JSON でやりとりするだけの API サーバが必要な場合、

Roda が Rails の代わりに使える可能性はあると筆者は考えています。

もちろん、Railsの方が開発はしやすいでしょう。Railsに慣れていればなおのことだと思います。Rodaを使うことで、その慣れによる開発効率は失われてしましますが、代わりに性能という恩恵を受けることが出来ます。

「1.5 パフォーマンス」で触れた TechEmpower 社による JSON レスポンスのベンチマーク結果によると、Roda + Sequel の組み合わせは Rails の実に 5 倍以上の性能を出しています。

勿論、これはあくまでベンチマークであり、実際のアプリケーションでそこまでの差異が出るとは限りません。しかし Roda を使うことで速くなる可能性は間違いなくあるでしょう。最初 Rails で開発していたが、性能が期待通りにならなかった、という状況に変わる、というようなケースがありましたら、是非 Roda についても検討してみてください。

という訳で、前置きが長くなりましたが、ここでは、簡単な JSON 出力機能を追加してみようと思います。とはいえ、JSON のプラグインが提供されているので、そのプラグインの設定をするだけです。

まず、下記のように、JSON プラグインの読み込み処理を app.rb に追加しましょう。

リスト 2.14: app.rb

```
plugin :json, classes: [Array, Hash, Sequel::Model]
```

classes には JSON に変換する対象のクラスを指定します。デフォルトは Array と Hash です。今回は Sequel で取得した結果も変換対象にしたいので、対象に Sequel::Model も追加しています。

続いて、model 側の修正です。

リスト 2.15: todo.rb

```
class Todo < Sequel::Model
  plugin :json_serializer
end
```

Sequel も Roda と同様にプラグインシステムがあります^{*4}。JSON のシリアライズ処理用のプラグインも提供されているので、そのプラグインを読み込むようにしましょう。

最後に、ルーティングの追加です。app.rb に追加しても良いのですが、折角 multi_route プラグインが読み込まれているので、JSON の出力処理については別のファイルに定義しようと思います。まず、routes/prefix1.rb を適切な名前に変更しましょう。JSON を出力するためのルーティングなので、"api" という名前にします。

```
$ mv routes/prefix1.rb routes/api.rb
```

ファイルにルーティングを追加しましょう。/api/todos にアクセスしたら、JSON が出力されるようにしましょう。

リスト 2.16: routes/api.rb

```
class MyTodo
  route 'api' do |r|
    r.get 'todos' do
      Todo.all
    end
  end
end
```

これで完成です。<http://localhost:9292/api/todos> にアクセスして、JSON が出力されることを確認してください。

^{*4} 正確には、Sequel のプラグインシステムが先にあって、Roda のプラグインシステムは Sequel のプラグインシステムを元に作られています。

付録 A

Roda の樹のもっと上へ

ここでは、Roda についてもっと知る為のコンテンツについてご紹介します。

ドキュメント

Roda のドキュメントは、 <http://roda.jeremyevans.net/documentation.html> にまとまっています。API doc へのリンクだけではなく、プラグインの一覧や発表資料へのリンク等もまとまっていますので、まずはこのページを見るのがおすすめです。

ソースコード

詳細な挙動を確認したくなったら、やはりソースコードを見るのが確実です。Roda のコアのファイル (roda.rb) は 1000 行程度しか無いので、割とすんなり読めるでしょう。

Giftsmas

ここからは Roda で作られたアプリケーションについてご紹介します。まずは、Giftsmas(<https://github.com/jeremyevans/giftsmas>) というアプリケーションです。これは Jeremy Evans 氏本人が作成したアプリケーションで、ギフトを管理する為のアプリケーション (どんなイベントがあって、誰に何のギフトを送ったのか等) です。

model が 4 つ、メインのコードも約 250 行程度しか無い小さなアプリケーション

付録 A Roda の樹のもっと上へ

ンです。ですが、Jeremy Evans 氏本人が作成しているということもあり、Roda 及び Sequel を使った実際のアプリケーションがどのようなものになるのかの参考になるでしょう。

Kontena

最後にもう 1 つ。Kontena(<https://github.com/kontena/kontena/tree/master/server>) という、Docker コンテナを管理する為のアプリケーションです。

こちらは Roda + DB に MongoDB という構成です。model も routes のファイルも 40 以上という中々重厚なアプリケーションです。中規模程度のアプリケーションを Roda を使って本当に作る事が出来るのかの参考になるでしょう。

First step of Roda

2018 年 4 月 22 日 初版第 1 刷 発行

著 者 y-yagi
