

目次

第 1 章	はじめに	1
1.1	本書の目的	1
1.2	本書の対象読者	1
1.3	書式例	3
ruby および rails コードの書式例	3	
差分がある場合の書式例	3	
ターミナルコマンドの書式例	3	
ターミナルの実行結果の書式例	3	
1.4	サンプルコード	3
1.5	開発環境	3
Mac に開発環境をつくる	4	
Windows に開発環境をつくる	4	
Ruby と Rails をインストールする	6	
第 2 章	一番小さな Rails アプリづくり	7
2.1	一番小さな Rails アプリをつくる	7
アプリの作成と Welcome 画面	7	
コラム： rails new コマンドでやっていること	9	
rails g コマンドでページを作る	10	
コラム： タイムゾーンの設定	13	
2.2	Web アプリはどのように動作しているか	13
コラム： Web サーバ	16	
2.3	インターネットの向こう側とこちら側	16
2.4	今回つくった Rails アプリの動作まとめ	18
2.5	Rails での開発の進め方	18
2.6	Rails が生成するファイル	19
rails new コマンド	19	
rails g コマンド	20	
2.7	Rails アプリの処理の流れ	21
Routes	23	
コントローラ	26	
ビュー	27	
まとめ	28	
2.8	さらに学びたい場合は	29
第 3 章	CRUD の基礎と index アクション	31
3.1	CRUD 基礎	31

	アプリの作成	31
	scaffold コマンド	33
3.2	index アクション	36
	Routes	38
	コントローラ	41
	ビュー	42
3.3	まとめ	44
3.4	さらに学びたい場合は	46
第4章	new, create アクション	47
4.1	概略	47
	ステップ 1: new アクション（新規作成画面）	48
	ステップ 2: create アクション（画面なし）	48
4.2	new アクション	48
	Routes	51
	コントローラ	53
	ビュー	54
	リクエストを観察する	59
4.3	Create アクション	63
	新たなリクエスト	63
	Routes	64
	コントローラ	65
	パラメータ	66
	Strong Parameters	68
4.4	まとめ	69
4.5	さらに学びたい場合は	70
第5章	モデル	71
5.1	データの永続化	71
5.2	モデルの基本的な使い方 その 1 保存	74
5.3	モデルの基本的な使い方 その 2 読み込み	74
5.4	モデルの基本的な使い方 その 3 検索	75
5.5	実習 : rails console でモデルを使う	75
5.6	モデルの仕組み	76
	問: save や all といったメソッドが使えるのはなぜでしょうか?	76
	問: title や memo といった要素があることをどこで知るのでしょうか?	76
	データベース (DB) とは?	76
5.7	DB はいつ作られたのか?	77
5.8	マイグレーション (migration) ファイル	77
5.9	保存したあとの処理	81

5.10	まとめ	83
	scaffold で作られる model、migration	83
	モデルの使い方	86
5.11	さらに学びたい場合は	87
5.12	既存の DB テーブルにカラムを増やすには？	87
5.13	新しいモデルと migration を一緒に作るには？	89
5.14	rails g コマンドまとめ	89
5.15	scaffold でつくった既存テーブルへカラムを追加するには？	90
	books テーブルに string 型の author を加える	90
	migration から DB を作る	90
	view を修正	91
	controller を修正	92
	動作確認	93
第6章	Gem ライブリ	95
6.1	Gem ライブリ	95
6.2	Gem をインストールして利用する	95
6.3	Bundler と Gemfile	96
6.4	Gemfile に書かれた Gem のバージョンアップ	98
6.5	Gemfile を使って実行する	98
6.6	Gemfile でのバージョン指定	99
6.7	まとめ	99
第7章	画像アップロード機能の追加	100
7.1	画像情報を格納するための DB カラムを追加	100
7.2	carrierwave gem を追加	101
7.3	モデルの変更	102
7.4	コントローラの変更	102
7.5	ビューの修正	102
7.6	動作確認	104
7.7	まとめ	108
第8章	あとがき	109
8.1	さらに学びたい方への資料	109
	Git と GitHub でソースコードを管理する	109
	つくったアプリを公開する	109
	学び方の資料	109
	Rails を学ぶ資料	109
	Ruby を学ぶ資料	111
	HTML と CSS を学ぶ資料	111

Git を学ぶ資料	112
8.2 Ruby コミュニティ	112
8.3 謝辞とメッセージ	113
8.4 著者略歴	113

第1章 はじめに

Ruby は開発者を幸せにする言語。

もっと多くの開発者を Ruby で幸せにしたい。

- matz

ブラウザを使って買い物をする、SNS を読み書きする、料理のレシピを検索する。Web アプリが提供するサービスは私たちの生活に深く根ざしています。それらを使うだけではなくて、どうやって動いているか知りたい、自分でも作ってみたい。本書はその第一歩を踏み出すための資料です。

1.1 本書の目的

本書の目的の 1 つは Web アプリ作成のための便利なフレームワーク Ruby on Rails（以下、Rails）を使い、題材として写真や文書を投稿できるミニブログアプリを作成して動かし、その仕組みを学ぶことです。Web アプリがどういうものなのか、何ができるのか、どのような仕組みで動いているのかを説明していきます。できるだけ簡単なサンプルアプリを用意し、その仕組みを説明することで、Rails および Web アプリの基礎を説明していきます。

また、本書の目的のもう 1 つは、世の中にあるたくさんの情報を読むために前提となる基礎知識を身につけることです。Rails を学ぶための良い資料として、Rails ガイド や Rails チュートリアル があります。本書を読み進めると、これらの資料を読み進めるための基礎知識が身についてきます。実践的な Rails アプリを作るために必要なさまざまな資料を読むための入り口として本書はあります。

1.2 本書の対象読者

本書が対象としている読者は以下のようの方々です。

- プログラミングは初めてである
- Web プログラミングは初めてである
- Rails プログラミングは初めてである
- Rails プログラミングを講義で教えたい

プログラミングは初めての方も安心してください。プログラミングの世界へようこそ！ 本書は筆者が作成した一橋大学社会学部での半期分の講義資料をもとに書いています。講義では初めてプログラムを書く学生さんが多数派でした。学生さんたちがつまづいた箇所について補足をして、資料を加筆していました。プログラミングが初めての方に読んでいただけるように、丁寧に説明を

していきます。

プログラミングの世界、Web の世界は広大です。本書では HTML（ブラウザで表示するための言語）と CSS（HTML と一緒に使う、装飾するための言語）については最低限の説明のみにとどめています。学習したい方へ、巻末に HTML と CSS を学習する資料を載せています。もしもあなたがデザイナーであり、HTML コーディングはおてのもの、という場合は大きなアドバンテージがあります。本書の知識と組み合わせることで鮮やかな Web アプリを作ることができるようになるでしょう。

また、プログラミングが初めての方で、本書を読んで「プログラムの基礎をもっと学んでから挑戦したい」「プログラムを読んで理解はできるが、自分で考えて書くのが難しい」と感じた方は、筆者の別著書である「ゼロからわかる Ruby 超入門」（書籍情報はあとがきの章にあります）をお勧めします。

初めてプログラミングをする方へ向けた、Ruby プログラミングの入門書です。プログラミングの基礎である条件分岐、繰り返しの概念を図解を入れて丁寧に説明します。また、Rails で使う知識を広くカバーするように書いています。

もしもあなたが他の言語でプログラミングをした経験があれば、これも大きなアドバンテージです。特に JavaScript（ブラウザ上で動作するプログラミング言語）の経験がある方は、本書の知識をつかってインタラクティブで格好良い Web アプリを作ることができるようになるでしょう（本書では JavaScript については説明しません）。また、Ruby の経験がある方は、本書の内容を越えた範囲について、今までの知識をつかって早く成長することができるでしょう。

そして、Rails プログラミングを講義で教えたい方もいらっしゃると思います。本書は Web アプリがどういったものなのか手を動かしながら学ぶ構成になっています。大学で行った講義の資料を元に書いており、プログラミングが初めての学生さんらに向けて行った、1回 90 分の講義 10 回分の内容です。その後、3回の講義時間を使って自由に Web アプリを作る実習を行ったところ、彼ら彼女らは自分で考えた、実用的であったり、たのしさにあふれたりする、独創性のあるプログラムを作りあげて発表してくれました。本書はできるだけやさしい説明を心がけますが、Ruby のことやターミナルのこと、プログラミングでしか使わない記号など、つまづいたときにサポートの手を差し伸べられる状況を作つておくと順調に前へ進めるようになります。

一方で、本書の対象から外れるのは以下のようの方です。

- Rails の基礎はわかっている

本書は Rails の入門的な本やサイトを読んで難しいと感じた方に、今までの知識を埋められるように書いています。逆に言えば、他の入門書を読める力があれば、本書は易しすぎると感じるかもしれません。具体的には、Rails チュートリアル を読み進められるようであれば、本書から得られるものは少ないのではないかと思います。

1.3 書式例

本書では以下のようにコードやターミナルコマンドを表現します。

ruby および rails コードの書式例

```
puts "Hello world!"
```

差分がある場合の書式例

```
class HelloController < ApplicationController
  def index
-   @time = Time.current
+   @time = 1.day.ago
  end
end
```

-, +印が付いている場合はその行について変更を行うことを示します (-, +がついていない部分はそのままです)。-で始まる行を削除し、+で始まる行を追加してください。

ターミナルコマンドの書式例

```
$ rails s
```

先頭の\$マークはターミナルを表す印です。入力する必要はありません。

ターミナルの実行結果の書式例

```
=> Booting Puma
... (略)
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
```

1.4 サンプルコード

本書で使うサンプルコードは以下に置いてあります。

https://github.com/igaiga/rails_textbook_sample

1.5 開発環境

本書は2020年1月1日現在の最新バージョンであるRuby 2.7.0、Rails 6.0.2.1で動作確認をしています。

Webアプリのプログラムを書くための道具として、ターミナル、エディタ、ブラウザをつかいます。

ターミナルはコンピュータをコントロールするための道具です。自分で書いたプログラムをターミナルから実行することができます。ターミナルで Ruby および Rails のプログラムを実行するためには、開発環境を作る必要があります。最初に Mac と Windows の場合それぞれの環境構築方法を説明して、その後でどちらの環境でも共通である Ruby と Rails のインストール方法を説明します。

エディタはプログラムを入力する道具です。エディタは好みのものがあればそれを利用してください。特にない場合は Visual Studio Code をお勧めします。Visual Studio Code は無料で使うことができます。ライセンスは MIT ライセンスです。

ブラウザはつくった Web アプリへアクセスするための道具としてつかいます。お好きなものをご利用いただけますが、文中で説明につかわれている Chrome 使うと分かりやすいです。

もしもここに書いている内容でうまくいかない場合は、RailsGirls ガイド インストール・レシピ のページが新しい情報で更新されているので、こちらも参考にしてみてください。

- RailsGirls ガイド インストール・レシピ
 - <https://railsgirls.jp/install>

Mac に開発環境をつくる

開発環境づくりに必要な xcode ツールをインストールします。

```
$ xcode-select --install
```

Homebrew をインストールします。Homebrew は Mac でさまざまな便利なプログラムをインストール、管理するためのツールです。

```
$ /usr/bin/ruby -e "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install/master/install)"
```

Homebrew を使って rbenv をインストールします。rbenv は Ruby をインストールして管理するための道具です。

```
$ brew update
$ brew install rbenv
$ echo 'eval "$(rbenv init -)"' >> ~/.bash_profile
$ source ~/.bash_profile
```

これで Ruby と Rails をインストールするための準備ができました。つづいて「Ruby と Rails をインストールする」へ進んでください。

Windows に開発環境をつくる

Windows の場合は WSL (Windows Subsystem for Linux) と rbenv を利用することで開発環境をつくることができます。インストールした WSL 上で Linux 向けツールを利用して開発しま

す。WSL を利用するためには Windows10 以降が必要です。もしも Windows10 よりも前のバージョンを利用している場合や、WSL のインストールがうまく行かないときには、RailsGirls ガイド インストール・レシピ のページにある「Windows 用セットアップ（WSL が使えない方向け）」を参考にしてみてください。

WSL をインストールする方法は Windows Subsystem for Linux Installation Guide for Windows 10 のページに書かれていますので、こちらの手順でインストールしてください。また、RailsGirls ガイド インストール・レシピ のページにも同様の手順がスクリーンショットをつけて丁寧に説明されています。インストールする Linux を選ぶところでは好きなものを選んでもらって構いませんが、特になければ Ubuntu を選択することをお勧めします。

WSL のインストール後、起動したアプリケーションはターミナルあるいは Bash ウィンドウと呼びます。ターミナルにはユーザー名とパスワードの入力を促す画面が表示されていますので、適当なユーザー名（半角英数小文字のみ）とパスワードを入力しましょう。パスワードは 2 回入力する必要があり、画面には表示されませんが正しく入力されています。

ターミナルで以下のコマンドを実行してください。なお、最初のコマンドを実行する際にパスワードの入力を求められますが、先程ターミナルに入力したパスワードを入力してください。Windows のパスワードではないことに注意してください。`sudo dpkg-reconfigure tzdata` を実行するとタイムゾーンを設定する画面が出ますので、「Asia」を矢印キーで移動して選択して、Tab キーを押して Ok ボタンに移動して、Enter キーを押してください。つづいて都市を選択する以下の画面が起動するので、「Tokyo」を矢印キーで移動して選択して、Tab キーを押して Ok ボタンに移動して、Enter キーを押してください。

```
$ sudo apt update
$ sudo apt upgrade -y
$ sudo apt install autoconf bison build-essential libssl1.0-dev libyaml-dev libreadline-dev zlib1g-dev libncurses5-dev libffi-dev libgdbm-dev sqlite3 libsqlite3-dev nodejs-dev node-gyp npm -y
$ sudo npm install --global yarn
```

つづいて rbenv をインストールします。rbenv は Ruby をインストールして管理するための道具です。以下のコマンドを実行してください。

```
$ git clone https://github.com/rbenv/rbenv.git ~/.rbenv
$ git clone https://github.com/rbenv/ruby-build.git "$(rbenv root)"/plugins/ruby-build
$ echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
$ echo 'eval "$(rbenv init -)"' >> ~/.bashrc
$ source ~/.bashrc
```

これで Ruby と Rails をインストールするための準備ができました。つづいて「Ruby と Rails をインストールする」へ進んでください。

Ruby と Rails をインストールする

rbenv を使って Ruby をインストールします。Ruby のバージョンは、より新しいものがあればそちらを利用して構いません。rbenv からインストール可能な Ruby のバージョンは rbenv install -l で調べることができます。

```
$ rbenv install 2.7.0
```

インストールした Ruby を利用するように設定します。

```
$ rbenv global 2.7.0
```

以上で Ruby のインストールは完了です。続いて、インストールした Ruby を使って Rails をインストールします。

```
$ gem i rails
```

確認のため、`rails -v` コマンドを実行してみましょう。

```
$ rails -v
Rails 6.0.2.1
```

このようにインストールした Rails のバージョンが表示されればインストール成功です。結果に表示されるバージョンの数字は、実行時の最新バージョンとなるため異なることがあります。

Rails は最新バージョンのものをつかって問題ありませんが、もしも問題が出て進められなくなったときには、`gem i rails -v 6.0.2.1` のようにバージョンを指定して、本書の内容と揃えることもできます。そのときには、以降のページで出てくる `rails new` アプリ名コマンドを実行するときに、つかうバージョンを指定し `rails _6.0.2.1_ new` アプリ名とします。

また、本書執筆時の最新バージョンである Ruby2.7.0 と Rails6.0.2.1 の組み合わせは、ターミナルで多くの Warning (警告) メッセージが出ます。次のバージョンへ向けての互換性に関する警告が多く、有用で大切なものではありますが、ほかの重要なメッセージが見づらくなる問題もあります。Ruby と Rails のバージョンをみなさんが実行するときの最新バージョンにすることで、Warning メッセージが少なくなっている可能性があります。できるだけ新しいバージョンを使うことをお勧めします。

ここまでで Ruby と Rails の開発環境構築ができました。

第2章 一番小さなRailsアプリづくり

ここではできるだけ小さい構成のRailsアプリを作つてみます。Railsアプリがどのように動作するのかの説明と、Railsによって作られたファイルがどのような役割なのか、機能と関連づけて説明していきます。

2.1 一番小さなRailsアプリをつくる

アプリの作成とWelcome画面

まずはターミナルを起動して、以下のコマンドを打つてみましょう。

```
mkdir my_web_apps  
cd my_web_apps
```

`mkdir` はフォルダを作成するコマンド、`cd` はターミナル内で現在のフォルダを移動するコマンドです。Windowsで普段使っている「コンピューター（エクスプローラー）」を起動するには、ターミナルで `explorer .` と打つと現在のフォルダを開くことができます。MacでFinderを開くにはターミナルから `open .` と打ちます。`mkdir` コマンドで作成したフォルダが存在することが確認できます。フォルダはターミナルからとエクスプローラー（MacではFinder）からどちらから作成しても違いはないので、どちらでも問題ありません。

それでは、アプリを作りましょう。最初に作るのは、ブラウザに"Hello world!"と表示させるアプリです。先ほど作成したフォルダ `my_web_apps` の下に新しいアプリを作つてみましょう。ターミナルを起動して以下のコマンドを打ちます。

```
rails new helloworld
```

実行には少し時間がかかります。たくさんのメッセージが表示されます。

```
$ rails new helloworld  
  create  
  create  README.md  
  create  Rakefile  
... (略)  
☒ Done in 3.83s.  
Webpacker successfully installed
```

ターミナルの画面にこのように"Webpacker successfully installed"と表示されれば成功です。メッセージ中"Done in 3.83s."の3.83sは所用時間なので、実行するごとに変化します。`rails new` コマンドはたくさんのフォルダとファイルを自動で作ります。`rails new` コマンドでやっていることはこの後のコラムで説明します。

もしもエラーなどで rails new コマンドが中断されたあとでリトライしたいときには、アプリのフォルダ、ここでは helloworld フォルダを削除してからもう一度実行してみてください。フォルダが残っていると問題が発生することがあります。

次は以下のコマンドを実行してみてください。rails s コマンドは web サーバを起動するコマンドで、s は server の略です。

```
cd helloworld  
rails s
```

うまく動作している場合は、以下のような表示になります（メッセージ中"Version 3.12.1"の数字は異なる場合があります）。

```
$ rails s  
=> Booting Puma  
=> Rails 6.0.0 application starting in development  
=> Run `rails server --help` for more startup options  
Puma starting in single mode...  
* Version 3.12.1 (ruby 2.6.5-p114), codename: Llamas in Pajamas  
* Min threads: 5, max threads: 5  
* Environment: development  
* Listening on tcp://localhost:3000  
Use Ctrl-C to stop
```

では、ブラウザを起動して以下の URL を入力してアクセスしてみましょう。

- <http://localhost:3000>



▲図2.1: welcome rails

これは、Railsが起動し、あなたのブラウザからのリクエストを受け付けて、表示している画面です。ここまでわずかな手順で、ブラウザでページを表示する機能を持つWebアプリをつくることができました。

ここで実行したコマンド `rails s` の `s` は `server` の略です。省略した `s` でも、省略せずに `server` でも、同じように動作します。

コラム： `rails new` コマンドでやっていること

`rails new` コマンドを実行すると、たくさんのメッセージが表示されました。このとき、何が行われているのでしょうか。最初に、Railsアプリとして動作するのに必要なファイルとフォルダを作成します。次に、利用するGemライブラリ（Rubyの便利なプログラム集）をダウンロードして利用可能にします。つづいて、`rails webpacker:install`, `yarn install` が実行されて、JavaScriptのライブラリをダウンロードして利用可能にします。

`rails new` コマンドがエラーで中断されたときには、表示されるエラーメッセージを手がかりに問題を解決する必要がありますが、これらの知識も役に立つかかもしれません。また、エラーで中断したあとにリトライしたいときには、これらの処理が途中のどこかで止まっていることがあるので、アプリのフォルダを削除してからもう一度実行してみてください。

rails g コマンドでページを作る

ひきつづき、以下のコマンドを入力してみましょう（メッセージ中"in process 9959"の数字は実行するごとに異なります）。rails server が起動している場合は、Ctrl-c (controlキーを押しながらcキー) で終了してからコマンドを打ってください。

```
rails g controller hello index
```

```
$ rails g controller hello index
Running via Spring preloader in process 9959
  create  app/controllers/hello_controller.rb
  route  get 'hello/index'
invoke  erb
create    app/views/hello
create    app/views/hello/index.html.erb
invoke  test_unit
create    test/controllers/hello_controller_test.rb
invoke  helper
create    app/helpers/hello_helper.rb
invoke  test_unit
invoke  assets
invoke    scss
create      app/assets/stylesheets/hello.scss
```

もしも rails g コマンドを打ち間違えて違うファイルをつくってしまったときは、打ち間違えたコマンドの g の部分を d にして再実行すると、rails g コマンドで作成したファイルをまとめて削除してくれます。たとえば、`rails g controller hell index` と hell を hello と打ち間違えた場合は、`rails d controller hell index` コマンドを実行することで間違えてつくったファイル群を削除することができます（ターミナルでカーソルキーの↑キーを押すと、さきほど入力した内容が出てくるので、それを利用して g を d に直すと楽に実行できます）。

再び rails server を起動させましょう。

```
rails s
```

```
$ rails s
=> Booting Puma
... (略)
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
```

ブラウザを使い、以下のURLへアクセスします。

- `http://localhost:3000/hello/index`

Hello#index

Find me in app/views/hello/index.html.erb

▲図2.2: hello/index

この画面が出れば、ここまで意図通りに動作しています。さきほど実行した rails g コマンドはこのページ、/hello/index を作成するものでした。どのような仕組みで動作しているかは後ほどまた説明しますので、今はこのページに "Hello world!" と表示させてみることにしましょう。

app/views/hello/index.html.erb ファイルをエディタで開いてみてください。以下のようないい處が修正されています。

```
<h1>Hello#index</h1>
<p>Find me in app/views/hello/index.html.erb</p>
```

これを以下のように変更して、ブラウザで同じ URL へアクセスしてみてください。-の行を削除して、+ の行を追加してください。先頭の-や + は入力しません。(rails s は起動したままで大丈夫です。もしも rails s を一度終了していた場合は、rails s コマンドでもう一度起動してからアクセスしてください)。

```
- <h1>Hello#index</h1>
- <p>Find me in app/views/hello/index.html.erb</p>
+ <p>Hello world!</p>
```

Hello world!

▲図2.3: Hello world

"Hello world!" の文字が表示されましたか？ これで一番小さな Rails アプリができあがりました。ここへ、少しだけ Ruby のコードを書いて、現在時刻を表示する機能を追加してみましょう。

以下のように、+ の行を追加してください。

```
<p>Hello world!</p>
+ <p>現在時刻: <%= Time.current %></p>
```

Hello world!

現在時刻: 2019-12-30 01:28:22 UTC

▲図2.4: 現在時刻表示

表示されましたか？ ブラウザをリロードすると、現在時刻が更新される、アクセスしたそのときの時刻が表示されるアプリになりました。

ところで、現在時刻が9時間ずれていると思われた方もいるかと思います。これは異なるタイムゾーンで表示されているためです。この時刻はUTCタイムゾーンでの時刻です。UTCは協定世界時と呼ばれ、基準となるタイムゾーンとして使われています。日本での時刻はUTCよりも9時間早い時刻になります。プログラムを書き換えて、日本時間での現在時刻を表示させてみましょう。

```
- <p>現在時刻: <%= Time.current %></p>
+ <p>現在時刻: <%= Time.current.in_time_zone('Asia/Tokyo') %></p>
```

Hello world!

現在時刻: 2019-12-30 10:28:27 +0900

▲図2.5: 現在時刻表示

最後に、このままでもいいのですが、コードのロジックの部分をビューに書くのではなく、コントローラで書くことにしましょう。動作は同じまま、コードを書き換えます。

コントローラを次のように変更します。

`app/controllers/hello_controller.rb`

```
class HelloController < ApplicationController
  def index
+    @time = Time.current.in_time_zone('Asia/Tokyo')
  end
end
```

そして、ビューを変更します。

app/views/hello/index.html.erb

```
- <p>現在時刻: <%= Time.current.in_time_zone('Asia/Tokyo') %></p>
+ <p>現在時刻: <%= @time %></p>
```

これでブラウザからアクセスすると、先ほどと同じように現在時刻が表示されているかと思います。

次の節から、このアプリがどのように動作しているのかを説明していきます。

コラム：タイムゾーンの設定

今回の`Time.current.in_time_zone('Asia/Tokyo')`はこの場所で使う時刻だけを日本時間へ変更しました。この方法のほかに、アプリ全体でタイムゾーンを日本時間に設定する方法もあります。その場合は`config/application.rb`ファイル中に`config.time_zone = 'Asia/Tokyo'`と設定します。この方法の利点は、プログラムの中のあちこちで`in_time_zone('Asia/Tokyo')`を書かずに済み、`config/application.rb`ファイルの1カ所にまとめることができます。

2.2 Webアプリはどのように動作しているか

ここで、普段ブラウザからつかっているWebアプリがどのように動作しているかを見てみましょう。アドレス入力欄にURLを入力してエンターキーを押すと、「リクエスト」がURL先のサーバへ向けて飛んでいきます。たとえば`https://cookpad.com/`と入力した場合は、クックパッドのサーバへ向けてリクエストが飛んでいきます。リクエストは追って説明していきますが、ざっくりと「そのページを見たいという要求（リクエスト）」とイメージしてもらえばOKです。

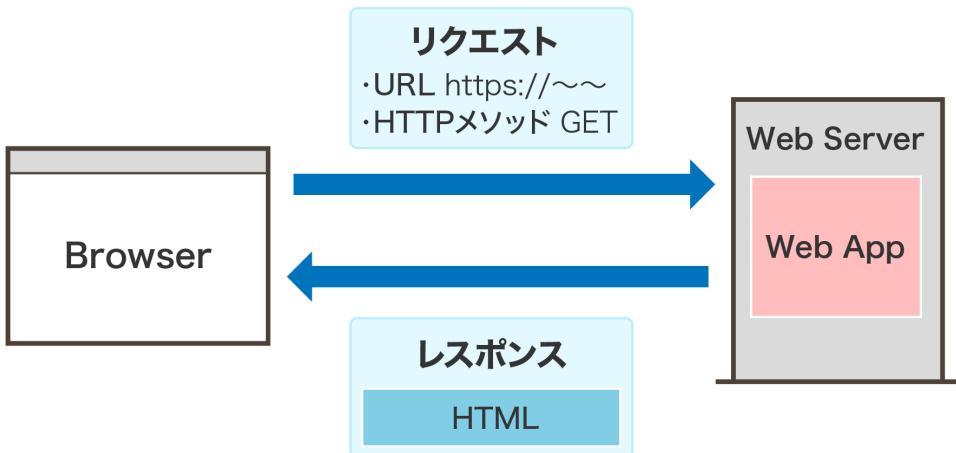
ブラウザにURLを入力してEnterを押すと、リクエストが飛ぶ



▲図2.6: リクエスト

Webサーバ上で動作しているWebアプリはリクエストを受け取ると、「レスポンス」としてHTMLで書かれたテキストを作成してブラウザへ返します。レスポンスは「Webアプリが返してきた情報群（HTMLで書かれた表示するための情報を含む）」とイメージできます。

レスポンスとしてHTMLが返ってくる



▲図2.7: レスポンス

ブラウザはHTMLを解釈して、私たちにとって見やすい、いつも見慣れたWebページを表示します。HTMLはHyperText Markup Languageのことで、Webページを記述するための言語です。

ブラウザはレスポンスで送られてきたHTMLを解釈して表示する



▲図2.8: HTMLを表示

HTMLはブラウザからも見ることができます。Chromeの場合は、どこかのサイト（たとえば <https://cookpad.com/>）へアクセスしたあと、右クリックメニューから「ページのソースを表示」を選ぶとHTMLで書かれたそのページを閲覧することができます。



▲図2.9: 右クリック

```

1 <!DOCTYPE html>
2 <html>
3 <head>
4 <meta charset='utf-8'>
5 <meta content='IE=edge' http-equiv='X-UA-Compatible'>
6 <script>
7   var __rendering_time = (new Date).getTime();
8 </script>
9
10 <link href='http://m.cookpad.com/' media='handheld' rel='alternate' type='text/html'>
11 <link href='https://assets.cpcdn.com/assets/device/apple-touch-icon.png?92b8bd477aedd34713e3d853583626f4d29101bcd7e6ceb52dcfe037b49e0988' rel='apple-touch-icon'>
12 <link href='/osd.xml' rel='search' title='クックパッド レシピ' type='application/opensearchdescription+xml'>
13 <title>レシピ検索No.1／料理レシピ載せるなら クックパッド</title><meta content="レシピ,簡単,料理,COOKPAD,くっくぱっど,recipe" name="keywords" /><meta content="日本最大の料理レシピサービス。317万品を超えるレシピ、作り方を検索できる。家庭の主婦が実際につくった簡単実用レシピが多い。利用者は5400万人。自分のレシピを公開することもできる。" name="description" /><meta name="robots" content="noarchive" /><meta name="google" content="nositelinkssearchbox" />
14
15
16 <link rel="stylesheet" media="all" href="https://assets.cpcdn.com/assets/libraries-bd21b9ab28067866195ed00c480186812ebb3701dcfb7ea90c638a3c6365fde8.css" />
17 <link rel="stylesheet" media="all" href="https://assets.cpcdn.com/assets/cookpad-df1570a45a7c41fe8125f37c695f77b215331d921d4ff6ac212eb8e40a49b25b.css" />
```

▲図2.10: HTML(抜粋)

ここまでに説明してきた以下の2つが、ブラウザの主な仕事です。

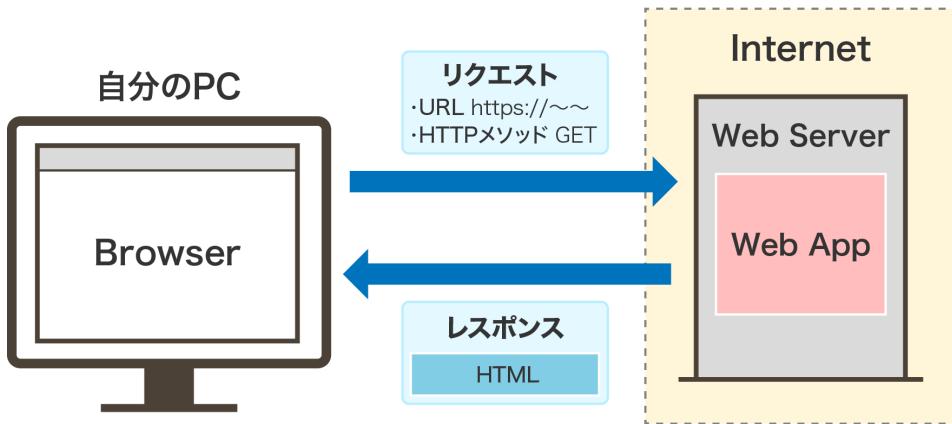
- リクエストをWebサーバへ投げる
- レスポンスで返ってきたHTMLを解釈して表示する

コラム：Webサーバ

Webサーバとはなにものなのでしょうか？Webサーバは「Webサービスを提供する場合に必要な共通の機能を提供するもの」と言えます。Webアプリはブラウザとのやりとりで必要な機能のうち、どのサービスでも使う機能はWebサーバに仕事をまかせ、自分のサービスで必要なオリジナルな機能を提供することに注力します。Railsで開発するときにはpumaというWebサーバを利用する設定になっていて、`rails s`を実行すると起動します。Webサービスを運用する場合は、nginxやApacheといった別の種類のWebサーバを組み合わせて使うことが多いです。

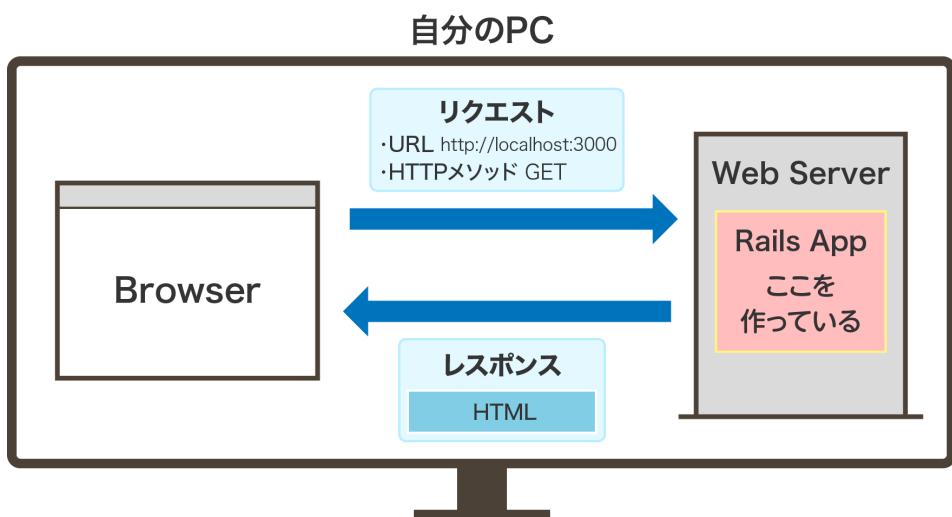
2.3 インターネットの向こう側とこちら側

ブラウザからWebサービスにアクセスするとき、通常、Webアプリはインターネット上にあります。ブラウザだけが自分のPCにあります。



▲図 2.11: インターネット上のサービスにアクセス

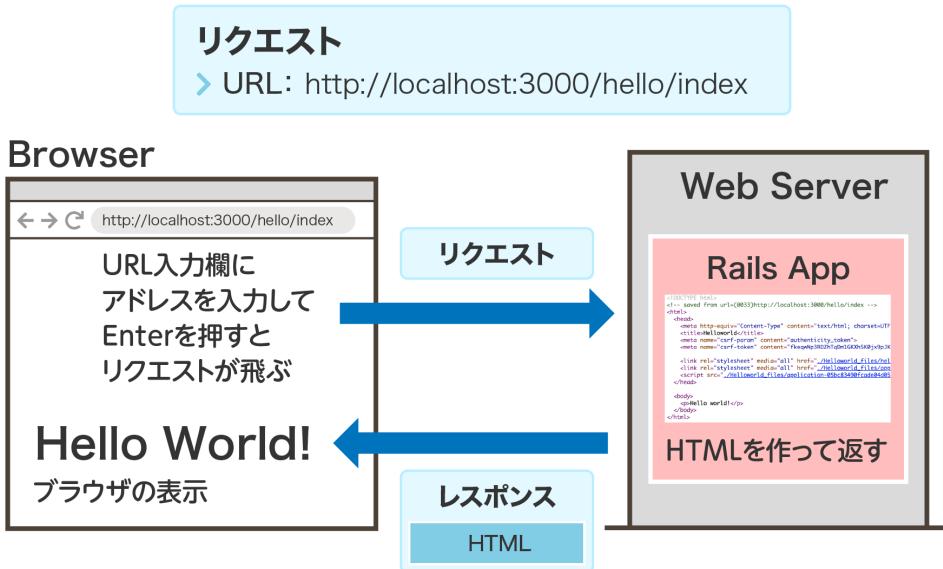
しかし、開発中は自分が作っているアプリをわざわざインターネット上へ置く必要はなく、自分のPCでWebアプリを動作させ、同じく自分のPCにあるブラウザからアクセス可能です。



▲図 2.12: 開発中は自分のPCでつくることができる

2.4 今回つくったRailsアプリの動作まとめ

今回つくったRailsアプリの動作を図にすると次のようにになります。



▲図2.13: 今回つくったRailsアプリの動作

ブラウザのURL欄にアドレスを入力してEnterを押すとリクエストが飛びます。リクエストを受け取ったRailsアプリはHTMLをつくり、レスポンスとして返します。レスポンスを受け取ったブラウザはHTMLを解釈し、画面に表示します。

2.5 Railsでの開発の進め方

Railsでの基本的な開発の進め方は以下の2つを繰り返すサイクルになります。

- ひな形になるファイル（ソースコードや設定ファイル）の生成
- つくっているアプリ用にファイルを変更、追記

実は、さきほどつくったアプリもこの手順で進めていました。

```
rails new helloworld
rails g controller hello index
```

これらのコマンドは「ひな形になるファイルの生成」を行っていました。そのあと、

app/views/hello/index.html.erbを編集して、

```
<p>Hello world!</p>
```

という内容に変更しました。このように、rails g コマンドなどでひな形となるファイルを生成し、それをそのアプリで使いたい形へ変えていく、Railsアプリ開発ではこれを繰り返してつくっていきます。

rails g コマンドはひな形を作成しますが、場合によってはこの手順を飛ばして、ゼロから手で書いても構いません。どちらの手順をつかっても、アプリをつくることが可能です。多くの場合、rails g コマンドを使った方が、楽につくれたり、ミスをしづらくなるので便利です。

2.6 Railsが生成するファイル

rails new コマンド

では、Railsはどのようなファイルを生成するのでしょうか。最初の rails new コマンドを実行したとき、以下のように create ... という表示がずっとされたと思います。rails が生成したファイルとフォルダの名前を表示していたのです。

```
$ rails new helloworld
create
create README.md
create Rakefile
create config.ru
create .gitignore
create Gemfile
create app
create app/assets/config/manifest.js
create app/assets/javascripts/application.js
create app/assets/javascripts/cable.js
create app/assets/stylesheets/application.css
create app/channels/application_cable/channel.rb
create app/channels/application_cable/connection.rb
create app/controllers/application_controller.rb
create app/helpers/application_helper.rb
... (略)
```

これらのファイル群によって、rails new をしただけで（何もコードを書かなくても）Webアプリとして動作します。たくさんのファイルがつくられていますね。Railsアプリの基本的なフォルダとファイル群を書いたものが次の図です。いきなりすべてを説明するのは難しいので、順番に説明していきます。役割ごとにフォルダが分かれていますが、それぞれの役割についてはこのあと説明していきます。



▲図2.14: Railsアプリの基本的なフォルダ・ファイル群（一部抜粋）

rails g コマンド

次に実行した rails g コマンドで作られたファイルを見てみましょう。

```
rails g controller hello index
```

```
$ rails g controller hello index
Running via Spring preloader in process 50811
create  app/controllers/hello_controller.rb
  route  get 'hello/index'
invoke  erb
create    app/views/hello
create    app/views/hello/index.html.erb
invoke  test_unit
create    test/controllers/hello_controller_test.rb
invoke  helper
create    app/helpers/hello_helper.rb
invoke  test_unit
invoke  assets
invoke    coffee
create      app/assets/javascripts/hello.coffee
invoke    scss
create      app/assets/stylesheets/hello.scss
```

ここで実行した rails g controller コマンドは、URL のパスが /hello/index であるページを表示するためのファイル群を作ります。g は generate の略です。rails g controller の後ろの hello と

index が、生成するページのパスを指定していることが分かります。ほかにもいくつかの generate コマンドが用意されているので、またあとで説明します。

もしも、コマンドを間違えて生成したファイルをまとめて削除したい場合は、g を d に替えたコマンドを実行すると、まとめて削除することができます。d は destroy の略です。

ここで生成、追記されたファイルのうち、特に重要なのは以下の3つのファイルです。

- app/controllers/hello_controller.rb
- app/views/hello/index.html.erb
- config/routes.rb

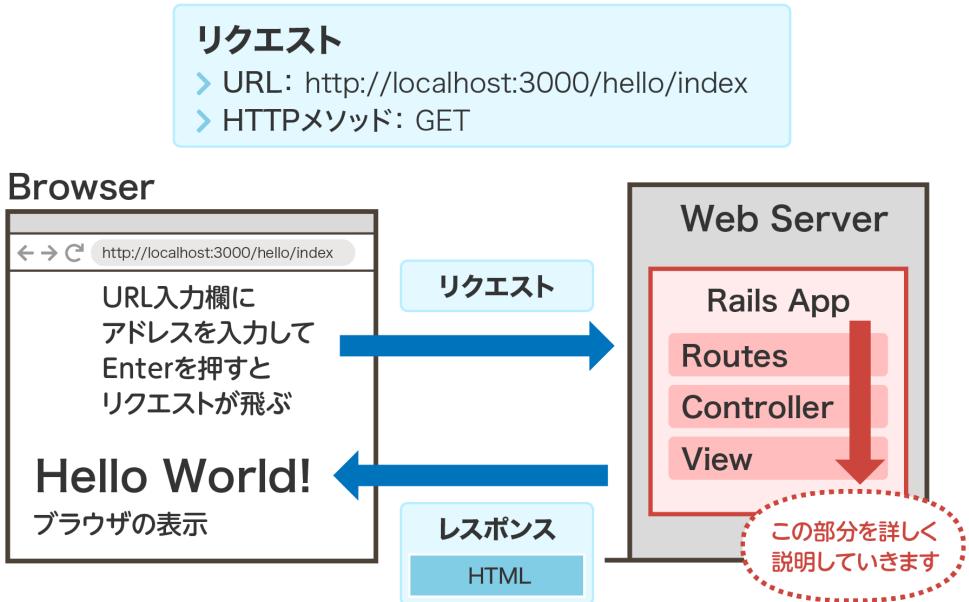


▲図 2.15: rails g controller hello index コマンドで生成されるファイル

これらのファイルがどのような働きをしているのかを、次の節で Rails がリクエストを受けてからレスポンスを返すまでの基本的な処理の順序を追いかけながら説明していきます。

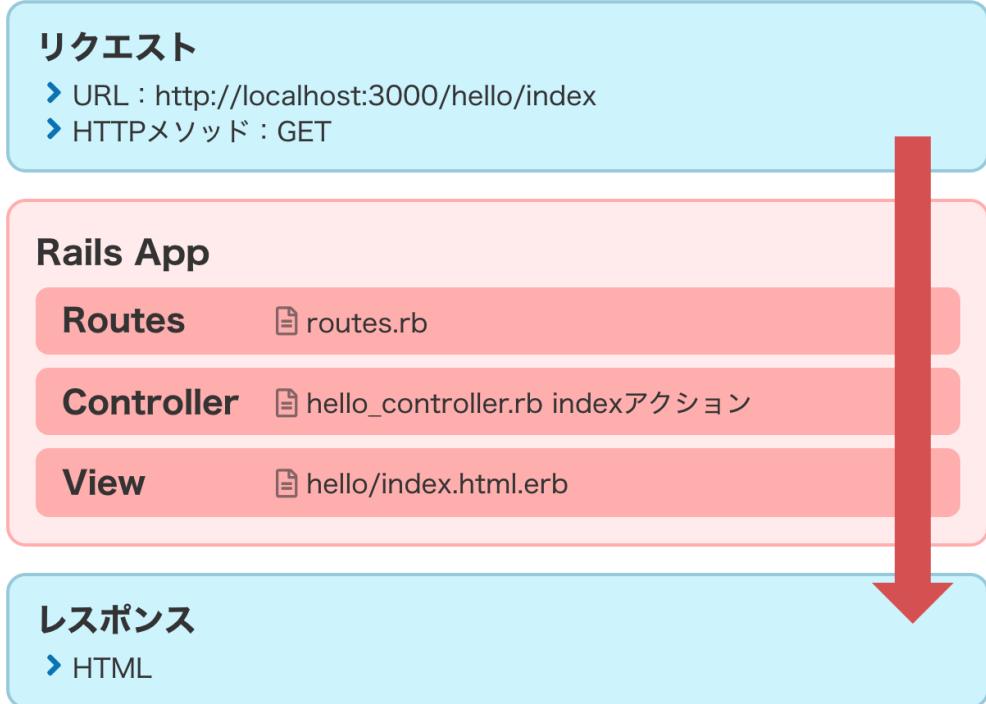
2.7 Rails アプリの処理の流れ

ブラウザからアクセスしたときに、Rails アプリはどのように動作しているのでしょうか？ 次の図は、今回つくった Rails アプリの動作を示したものです。前に出てきた動作の図から少し詳しくしています。



▲図2.16: Railsアプリの動作

ここでは、この図のRails Appの部分について詳しく説明していきます。対象箇所を抜き出したのが次の図です。



▲図 2.17: Rails アプリがリクエストを受けてレスポンスを返すまで

リクエストを受けたRailsアプリは、Routes, Controller, Viewの3つのコードを通過します。それぞれの場所で処理を行い、レスポンスとしてHTMLを生成して返します。各所ではそれぞれ仕事の分担が決まっていて、自分の担当の処理を行います。Routes, Controller, Viewでそれぞれどのような処理が行われているのか、順に見ていきましょう。

Routes

Routesは「リクエストのURLとHTTPメソッド」に応じて次に処理を行う先を決めるのが仕事です。URLは前に出てきましたが、HTTPメソッドとは何でしょうか？

- リクエスト
 - URL : http://localhost:3000/hello/index
 - HTTPメソッド : GET

リクエストを構成する要素のうち、重要なものがURLとHTTPメソッドです。URLはインターネット上の住所を表し、アクセスする先を指定するものです。もう一方のHTTPメソッドは、そのURLに対して「何をするか」を指示するものです。ブラウザのアドレス欄へURLを入力しEnterを押すと、HTTPメソッド"GET"でリクエストが飛びます。GETは「ページを取得する」指示です。GETのほかにもHTTPメソッドはいくつかあり、Railsアプリでよく使うものは4つ

です。GET以外のHTTPメソッドは次の章以降で説明していきます。

まとめると、リクエストは「URL」でアクセス先を、「HTTPメソッド」でなにをするかを指示します。

最初に、Routesは「リクエストのURLとHTTPメソッド」に応じて次に処理を行う先を決めるのが仕事だと書きました。RailsではRoutesの処理が終わると、次はコントローラのアクションへ処理が移ります。アクションとはコントローラに書かれているメソッドのうち、publicなメソッドのことです。「Routesから呼び出せるメソッド」と言っても良いでしょう。ここでのメソッドはRubyでのメソッドのことで、処理を集めて名前をつけたものです。HTTPメソッドとは別のものです。RoutesはリクエストのURLとHTTPメソッドから、処理の進み先であるコントローラのアクションを決定します。RoutesはリクエストとControllerのアクションとの対応表と言えます。

リクエスト

- URL : `http://localhost:3000/hello/index`
- HTTPメソッド : GET

routesはリクエストのURLとHTTPメソッドに応じて次の処理先を決める

○○コントローラ の □□アクション

▲図 2.18: Routes

では、対応表であるRoutes表を見て見ましょう。rails serverを起動させて`/rails/info/routes`へアクセスしてみてください。Routes表の見方を説明したのが次の図です。

<http://localhost:3000/rails/info/routes>

Helper	HTTP Verb	Path	Controller#Action
Path / Url	Path Match		
hello_index_path	GET	/hello/index(.:format)	hello#index
	HTTPメソッド	URL(のパスの部分)	Controllerのアクション

リクエスト

- › URL : <http://localhost:3000/hello/index>
- › HTTPメソッド : GET



HelloControllerのindexアクション

▲図 2.19: Routes 表

表中の "HTTP Verb" が HTTP メソッドです。"Path" (パス) は URL の後半部分に相当します。URL が "http://localhost:3000/hello/index" である場合、パスは "/hello/index" になります。(表示されたパスの後半部分の "(.:format)" は省略できる記述で、レスポンスで返すフォーマットを指定するための機能です。省略した場合は HTML を返すのが普通です。)

右端の "Controller#Action" が処理の移るコントローラとアクションを示しています。ここでは "hello#index" と書かれていますが、#より左側がコントローラ名、右側がアクション名です。この場合は、「HelloController の index アクション」を示しています。

この対応表を解釈すると、「リクエストの HTTP メソッドが "GET"、パスが "/hello/index" のとき、次の処理は "HelloController" の "index" アクションになる」となります。

ここで表示された Routes は config/routes.rb ファイルから生成されています。このファイルには次のような記述があります。

```
get 'hello/index'
```

これが Routes のコード部分で、この 1 行からさきほど説明した対応表が生成されています。「パス "hello/index" への GET でのアクセスで HelloController の index アクションが呼ばれる」という文です。Routes の書き方はまた追って説明していきます。

Routes についてまとめると、「Routes はリクエストのパスと HTTP メソッドによって次の処理

先であるコントローラとアクションを決める対応表」となります。

それでは、次の処理が行われるコントローラみてみましょう。

コントローラ

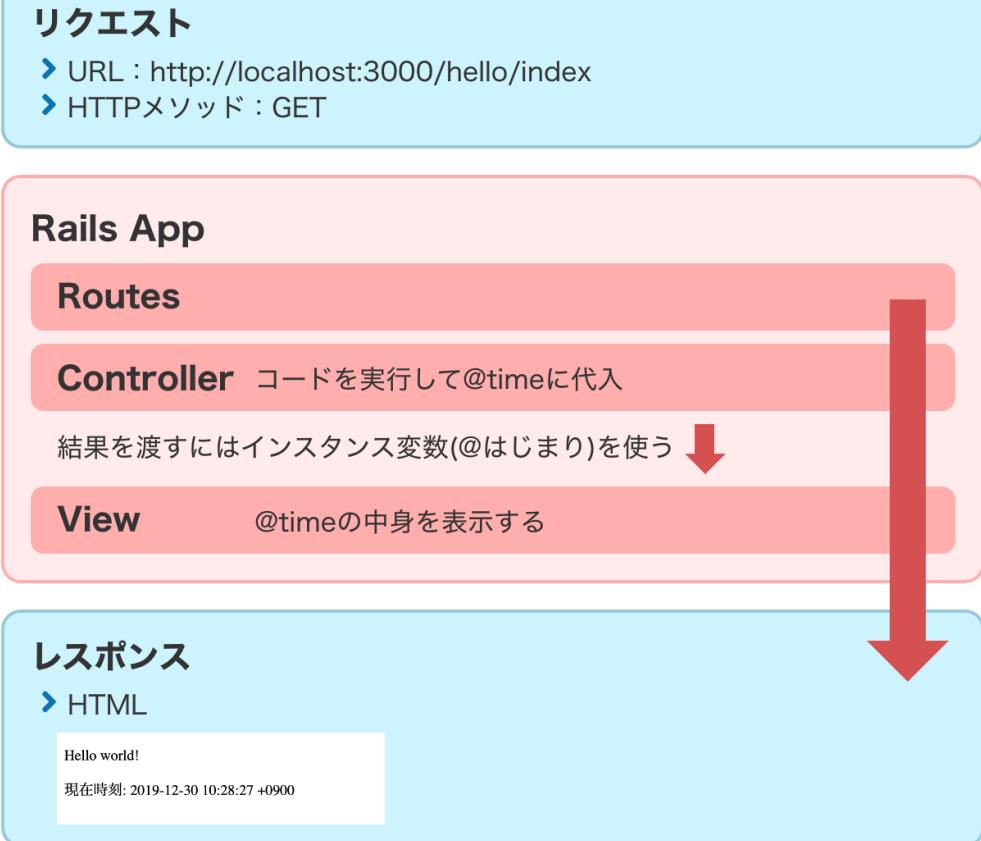
コントローラではさまざまな処理を行い、次のビューでつかうデータをつくって渡します。コントローラのファイルは `app/controllers/` へ置きます。さきほどの Routes で次の処理先として指定された HelloController は、`app/controller/hello_controller.rb` というファイルです。

```
class HelloController < ApplicationController
  def index
    @time = Time.current.in_time_zone('Asia/Tokyo')
  end
end
```

HelloController の `index` アクションが呼び出されます。`def index` から `end` までが `index` アクションです。コントローラにある `public` なメソッドをアクションと呼びます。アクションは Routes から次の処理先として指示される可能性があります。

この `index` アクションでは `@time` というインスタンス変数に現在時刻を代入しています。アクションの中のプログラム、ここでは `@time = Time.current.in_time_zone('Asia/Tokyo')` は、インデント（字下げ）されて書かれます。

変数は名札のようなもので、代入したものをあとから使えるように名前をつける仕組みです。変数のうち、`@`はじまりの変数のことをインスタンス変数といいます。インスタンス変数を使うと、コントローラから、このあとの処理先であるビューへ情報を伝えることができます。ちなみに、`@`はじまりではない変数はローカル変数と呼ばれ、このメソッド（アクション）を抜けると役目を終えて使えなくなります。つまり、ローカル変数へ代入してもビューから使うことはできません。ビューで使うためには、`@`はじまりのインスタンス変数を利用します。



▲図 2.20: インスタンス変数を使うとビューへ情報を渡すことができる

次にどのビューへ処理が進むかはコントローラで指定することができますが、今回は何も指定がありません。指定がないときは、コントローラおよびアクションと同名のビューを選択します。今回は HelloController の index アクションなので、次の処理先であるビューのファイルは app/views/hello/index.html.erb になります。

コントローラについてまとめてみましょう。コントローラではさまざまな処理を行い、次のビューに処理を渡します。ビューへ情報を伝えるためには@はじまりのインスタンス変数へ代入しておきます。次の処理先となるビューはコントローラで指定することができますが、指定していないときはコントローラおよびアクションと同名のビューを選択します。

それでは次の処理が行われるビューを見てみましょう。

ビュー

ビューでは HTML など、ユーザーの目に届く部分を作ります。作られた HTML は、レスポンスとしてブラウザへ送られます。

ビューのファイルである `index.html.erb` は、HTML のもとになるファイルです。ブラウザで表示させるための言語 HTML をそのまま書くことができます。さらに、普通の HTML との違いとして、Ruby のコードを実行した結果を埋め込むこともできます。このようなファイルをテンプレートと呼びます。ここでは `erb` という種類のテンプレートが使われています。書かれている HTML と、埋め込まれた Ruby のコードを実行した結果をあわせて HTML を作ります。では、ビューのファイルを見てみましょう。

`app/views/hello/index.html.erb`

```
<p>現在時刻: <%= @time %></p>
```

HTML の `p` タグがあります。その中に HTML ではない `<%=` と `%>` というタグがあります。これが Ruby のコードを実行するためのタグです。ここではその中にある `@time` が実行されます。`@time` インスタンス変数にはコントローラで実行された現在時刻 `Time.current.in_time_zone('Asia/Tokyo')` の結果が代入されているので、これが HTML へ埋め込まれます。

ビューで作られた HTML は、Rails がその他の加工を加えてレスポンスとして送出され、ブラウザに表示されます。作られた HTML はブラウザで「ページのソースを表示」機能をつかって確認することができます。

Hello world!

現在時刻: 2019-12-30 10:28:27 +0900

▲図 2.21: ブラウザからビューがつくった HTML を確認

ビューについての動作をまとめてみましょう。ビューは HTML など、ユーザーの目に届く部分をつくります。`erb` はテンプレートと呼ばれる HTML のもとになるファイルで、Ruby のコードを実行した結果を埋め込むことができます。ビューで使う情報をコントローラから送ってもらうときは、インスタンス変数を使います。作られた HTML は、Rails がその他の加工を加えてレスポンスとして送出します。

まとめ

最小構成の Rails アプリをつくり、リクエストを受けてレスポンスを返すまでの動作について説明しました。ポイントをまとめると以下のようになります。

- Routes、コントローラ、ビューの順番で処理を行い、HTML を作ってブラウザへレスポン

スを返す

- RoutesはリクエストのURLとHTTPメソッドに応じて、処理をするコントローラとアクションを決める対応表
- コントローラはさまざまな処理を行い、ビューでつかう情報を@はじめのインスタンス変数を使って渡す
- ビューはテンプレート書かれたRubyのコードを実行して埋め込み、HTMLを作る

リクエスト

- URL: http://localhost:3000/hello/index
- HTTPメソッド: GET

Rails App

Routes routes.rb

Controller hello_controller.rb indexアクション

View hello/index.html.erb

レスポンス

- HTML

▲図2.22: Railsアプリがリクエストを受けてレスポンスを返すまで

2.8 さらに学びたい場合は

- Railsガイド: Railsのルーティング
 - Routesについての詳しい説明です。
- Railsガイド: Action Controllerの概要
 - コントローラについての詳しい説明です。
- Railsガイド: Action Viewの概要
 - ビューについての詳しい説明です。

- Rails ガイド: レイアウトとレンダリング
 - コントローラから次のビューを指定する方法や、ビューを整理して書くための情報などが説明されています。

第3章 CRUDの基礎とindexアクション

Web アプリには基本となる 4 つの機能があります。ページの新規作成 (Create)、表示 (Read)、更新 (Update)、削除 (Destroy) です。それぞれの頭文字を取って CRUD と呼ばれています。大きな Web アプリも、この CRUD を基礎として作られます。

ここでは CRUD 機能を持った Rails アプリを作り、その動作について解説します。

3.1 CRUD 基礎

アプリの作成

今回も最初にアプリを作ります。本のタイトルとメモを管理する簡易なアプリです。以前の章で作成した `my_web_apps` フォルダの下に新しいアプリを作ります。ターミナルを起動して以下のコマンドを打ちます（メッセージ中の"Done in 3.61s."の数字は所用時間なので、毎回異なります）。

```
cd my_web_apps
rails new books_app
```

```
$ rails new books_app
      create
... (略)
Done in 3.61s.
Webpacker successfully installed
```

次に、以下のコマンドを実行します（メッセージ中"in process 52142"、"20160702024137"、"-> 0.0013s"などの数字は実行するごとに異なります）。

```
cd books_app
rails g scaffold book title:string memo:text
rails db:migrate
rails s
```

```
$ rails g scaffold book title:string memo:text
Running via Spring preloader in process 20280
  invoke  active_record
  create    db/migrate/20191012005727_create_books.rb
...
  invoke  scss
  create    app/assets/stylesheets/scaffolds.scss

$ rails db:migrate
== 20191012005727 CreateBooks: migrating =====
```

```
-- create_table(:books)
  -> 0.0034s
== 20191012005727 CreateBooks: migrated (0.0035s) =====

$ rails s
=> Booting Puma
... (略)
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
```

ブラウザを起動して以下の URL を入力してアクセスしてみます。

- <http://localhost:3000/books>

Books

Title Memo

New Book

▲図 3.1: 起動した画面

こんな画面が表示されましたでしょうか。ここから、New Book リンクをクリックして先へ進み、title 欄と memo 欄を記入し Create Book ボタンを押してみてください。

Book was successfully created.

Title: RubyとRailsの学習ガイド

Memo: Rails関連技術地図とそれらの学習資料の紹介

Edit | Back

▲図 3.2: データを入力して Create Book ボタン

ここから Back ボタンを押すと最初の /books のページに戻りますが、先ほど入力したデータが表示されていることが分かるかと思います。

Books

Title	Memo	Show	Edit	Destroy
RubyとRailsの学習ガイド	Rails関連技術地図とそれらの学習資料の紹介	Show	Edit	Destroy
New Book				

▲図 3.3: 入力したデータが表示される

さらにデータを追加したり、他のリンクも操作してみてください。

ここで作成したアプリは CRUD の各機能、すなわち、新規作成、表示、更新、削除を持っています。以降で、1つずつその動作を説明していきます。

scaffold コマンド

rails g scaffold コマンドは新規作成、表示、更新、削除、各機能を一度に作る機能です。つまり、scaffold を使うと CRUD 機能を持ったページを作成することができます。

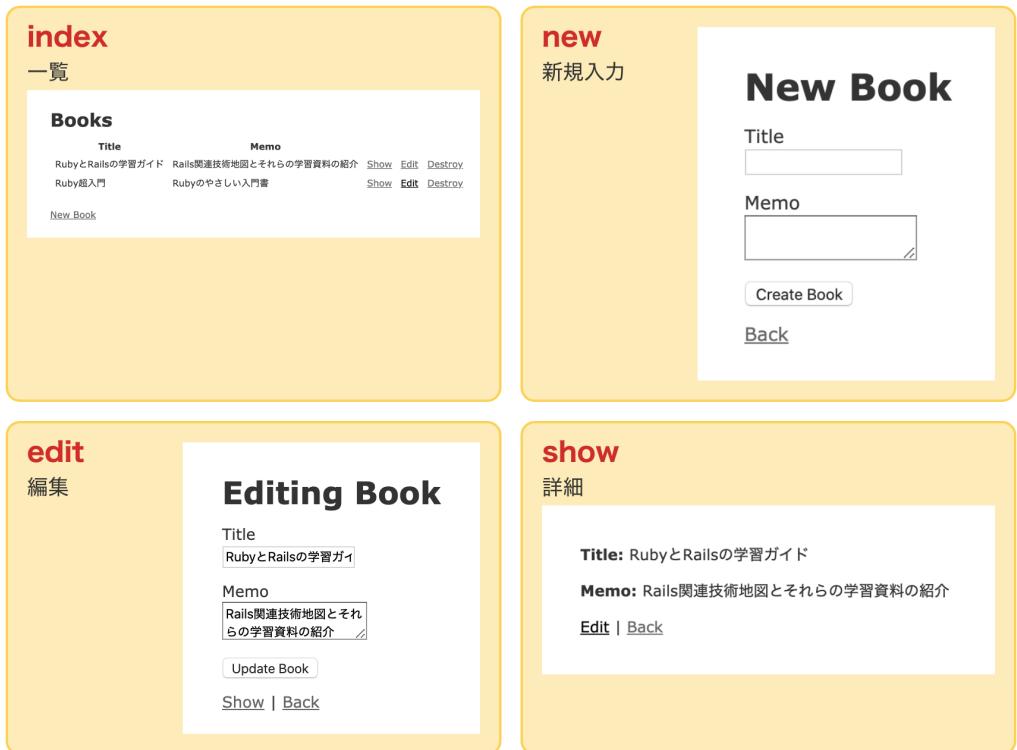
scaffold は英語で「(建築現場などの) 足場」という意味です。工事中の建物のまわりに組まれた足場の上で大工さんが作業している、あの足場です。scaffold コマンドは Web アプリを作り易くするための足場です。CRUD 機能は Web アプリでよく使う機能なので、「CRUD のこの部分にこの機能を足したらできるな」という場合に、scaffold で生成したファイル群を編集して自分のアプリを作っていくと効率良く作れます。

作成されるファイルは以下の通りです。

```
$ rails g scaffold book title:string memo:text
Running via Spring preloader in process 20280
  invoke  active_record
  create    db/migrate/20191012005727_create_books.rb
  create    app/models/book.rb
  invoke  test_unit
  create    test/models/book_test.rb
  create    test/fixtures/books.yml
  invoke  resource_route
    route    resources :books
  invoke scaffold_controller
  create    app/controllers/books_controller.rb
  invoke  erb
  create    app/views/books
  create    app/views/books/index.html.erb
  create    app/views/books/edit.html.erb
  create    app/views/books/show.html.erb
  create    app/views/books/new.html.erb
  create    app/views/books/_form.html.erb
  invoke  test_unit
  create    test/controllers/books_controller_test.rb
  create    test/system/books_test.rb
  invoke  helper
  create    app/helpers/books_helper.rb
  invoke  test_unit
  invoke  jbuilder
  create    app/views/books/index.json.jbuilder
  create    app/views/books/show.json.jbuilder
  create    app/views/books/_book.json.jbuilder
  invoke  assets
  invoke  scss
  create    app/assets/stylesheets/books.scss
  invoke  scss
  create    app/assets/stylesheets/scaffolds.scss
```

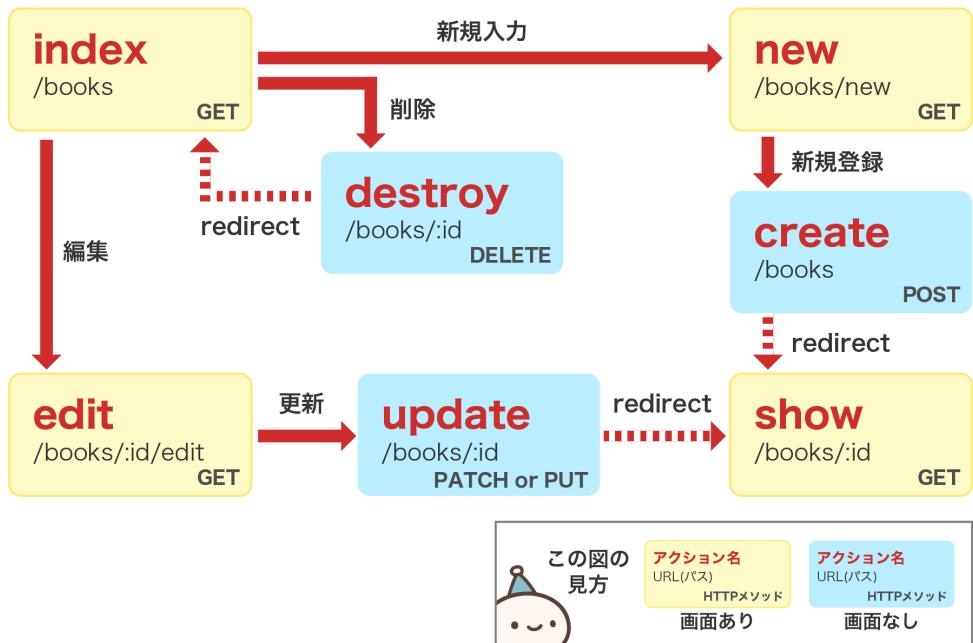
この中の route (invoke resource_route) , controller (invoke scaffold_controller) や view (invoke erb) は前の章で説明した rails の各役割を担うファイルです。

一方、ページの観点から見ると、scaffold は次に示す新規作成、表示、更新、削除のための 4 つの画面と、画面のない 3 つの機能を生成します。



▲図 3.4: CRUD 4 つの画面

それぞれの機能と対応するアクションを示したものが次の図です。前の章でも説明されたアクションは、コントローラに書かれた public なメソッドのことです。1 つの画面を表示するとき、1 つのアクションが実行されているのが基本形です。

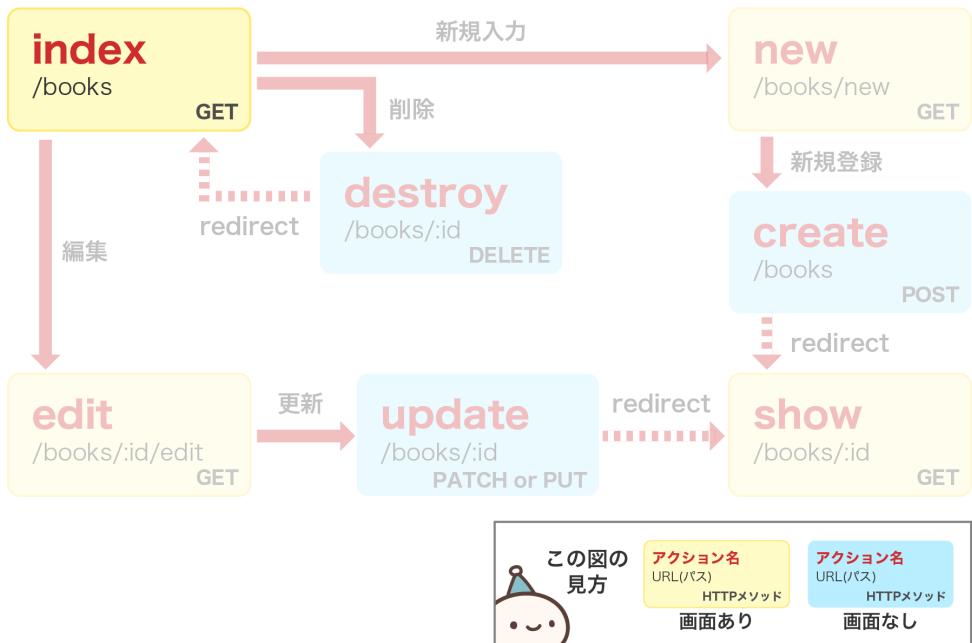


▲図 3.5: CRUD 7 つのアクション

最初は、index アクションのページについて見ていきましょう。

3.2 index アクション

index アクションについて見ていきましょう。さきほどの 7 つのアクションの図で示されたこの部分です。



▲図 3.6: index アクション

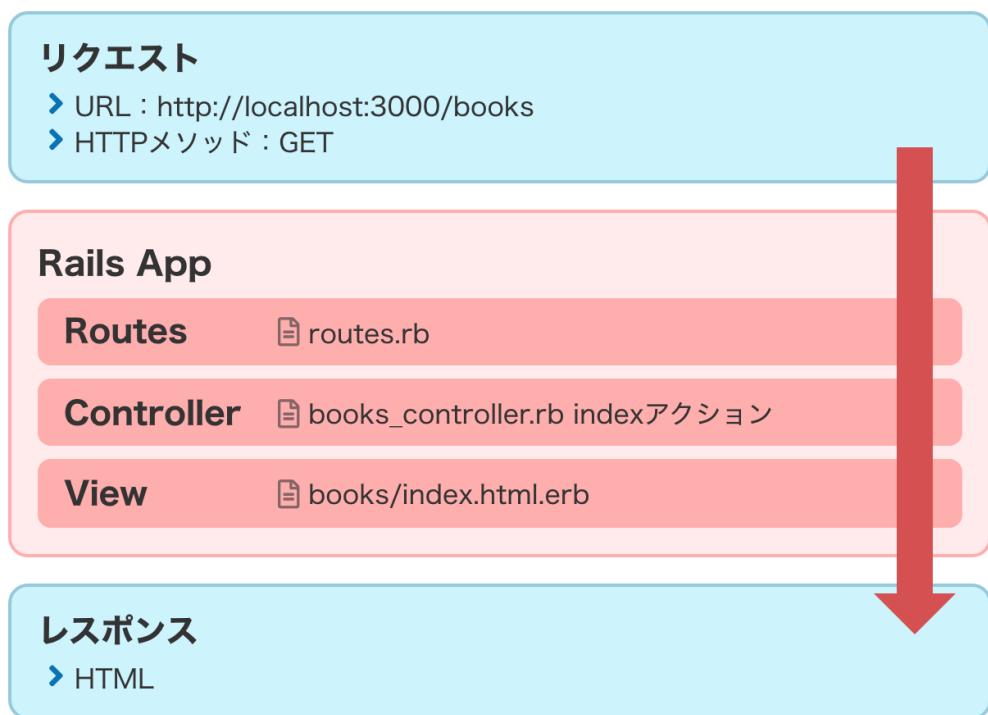
index アクションで表示されるページ（次の図）は登録されているデータの一覧です。ここでは登録されている Book の一覧表示画面になります。この画面が表示されるまでの処理を追いかけてみましょう。

Books

Title	Memo
RubyとRailsの学習ガイド	Rails関連技術地図とそれらの学習資料の紹介
Ruby超入門	Rubyのやさしい入門書
Show Edit Destroy Show Edit Destroy	
New Book	

▲図 3.7: index - 一覧画面

今回も処理の流れを図でみてみましょう。

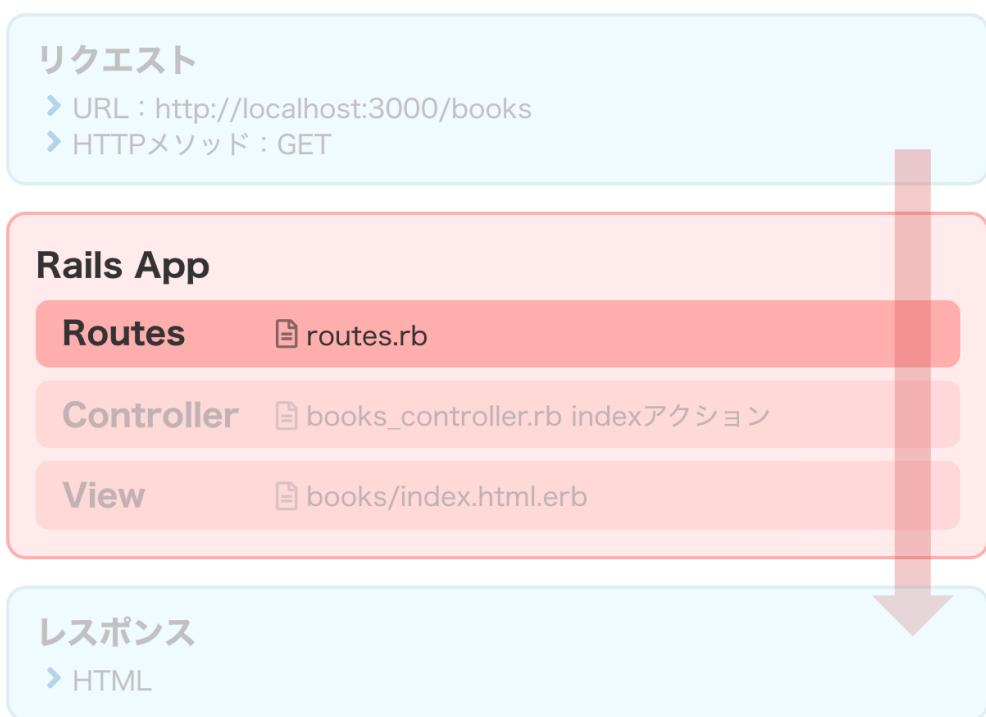


▲図 3.8: Rails アプリがリクエストを受けてレスポンスを返すまで (index アクション)

Routes から順番に処理を追っていきます。

Routes

Routes はリクエストから処理を行うコントローラとアクションを決めるための対応表です。



▲図 3.9: Routes

Routes 表から、リクエストごとに次に処理するコントローラとアクションが決まります。Routes 表を表示するために、rails server を起動して、<http://localhost:3000/rails/info/routes> へアクセスしてみましょう。

<http://localhost:3000/rails/info/routes>

Helper	HTTP Verb	Path	Controller#Action
Path / Url	Path Match		
books_path	GET	/books(.:format)	books#index
	POST	/books(.:format)	books#create
new_book_path	GET	/books/new(.:format)	books#new
edit_book_path	GET	/books/:id/edit(.:format)	books#edit
book_path	GET	/books/:id(.:format)	books#show
	PATCH	/books/:id(.:format)	books#update
	PUT	/books/:id(.:format)	books#update
	DELETE	/books/:id(.:format)	books#destroy

▲図 3.10: Routes 表

今回は URL が"/books"、HTTP メソッドが "GET" なので、下線部が該当行です。「/books に GET でアクセスされたとき、BooksController の index アクションへ処理を渡す」という意味になります。次に進むべきコントローラは books#index (BooksController の index アクション) となることが分かります。

コントローラの処理を見る前に、Routes のコードを見てみましょう。Routes のコードは config/routes.rb にあります。コードは以下のようになっています。

```
Rails.application.routes.draw do
  resources :books
  # For details on the DSL available within this file, see https://guides.rubyonrails.org/routing.html
end
```

Routes 表を作るコードは Rails.application.routes.draw do から end の間に書きます。#記号ではじまる行はわかりやすさのためのコメントで、動作には影響がありません。<http://localhost:3000/rails/info/routes> のページで表示された 8 行の対応表は、resources :books の 1 行によって作られます。CRUD はよく利用されるため、かんたんに書けるようになっているのです。

それでは次の処理、コントローラへ進みましょう。

コントローラ

Routes ではリクエストから処理されるコントローラとアクションが決まると、次はコントローラへ移動します。今回は BooksController の index アクションです。



▲図 3.11: コントローラ

コントローラのコードを見てみましょう。ファイルは app/controllers/books_controller.rb です。

```
class BooksController < ApplicationController
  ...
  def index
    @books = Book.all
  end
  ...
```

class BooksController < ApplicationController から最後の行の end までが BooksController のコードです。この中の index メソッド、つまり def index から次の end までが index アクションになります。

ここでやっていることは 1 行、`@books = Book.all` です。Book.all で Book の全データがついた配列を取得して、`@books` インスタンス変数へ代入します。Book.all で Book の全データがとれる仕組みについては、「モデル」の章で説明します。

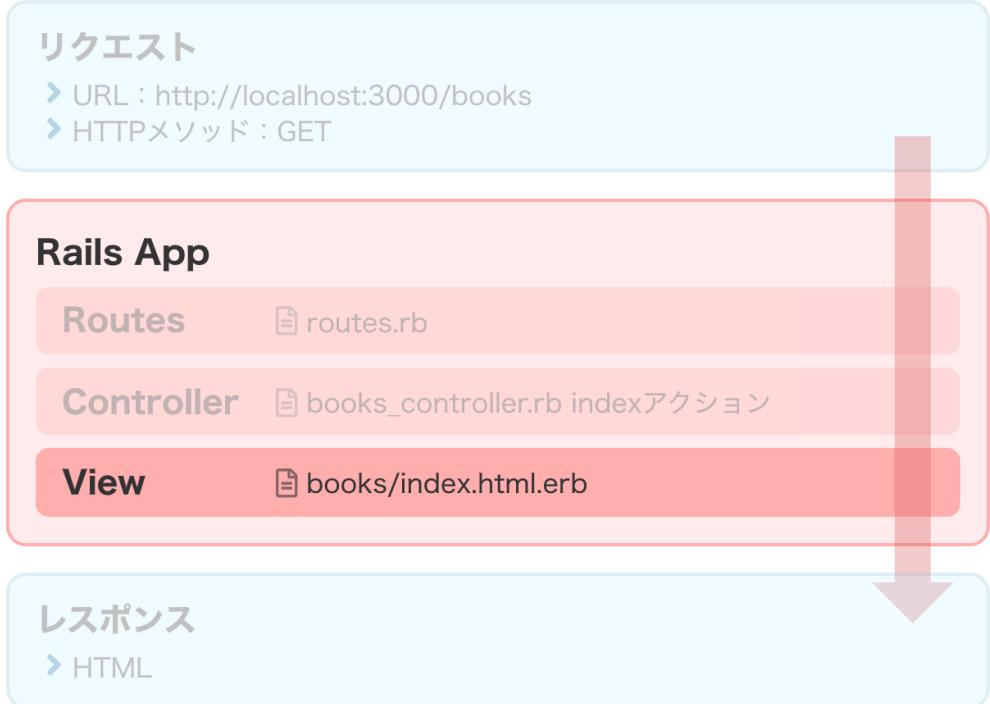
`@books` は`@`から始まる変数、インスタンス変数です。「一番小さな Rails アプリつくり」でも説明したように、インスタンス変数`@books` を使っているのは、このあと view で利用するためです。もしも、ここで変数をローカル変数`books` にしてしまうと、コントローラの処理が終わったところでなくなってしまいます。

もうひとつ大切なことがあります。`@books` と複数形になっています。これは、Book.all で取得するデータが複数の本のデータを格納した配列であることを示すためです。場合によっては、本のデータが 1 個、あるいは 0 個である、という状況もあります。そのときも 1 個、あるいは 0 個のデータが入った配列が取得されます。どのパターンも`@books` へは配列を代入するため、変数名を複数形の`@books` にするのが慣習です。細かいことのようですが、Rails や Ruby でのプログラミングでは変数の名前を複数形と単数形を意識して命名することが大切です。

コントローラのアクションでの処理が終わると、次はビューで処理が移ります。特に指定がないときは、`app/views/コントローラ名/アクション名` の view ファイルへ処理が移ります。ここでは `app/views/books/index.html.erb` です。

ビュー

最後はビューでレスポンスとして返される HTML がつくられます。



▲図 3.12: ビュー

今回は、このような画面の HTML がつくれられます。

Books

Title	Memo	
RubyとRailsの学習ガイド	Rails関連技術地図とそれらの学習資料の紹介	Show Edit Destroy
Ruby超入門	Rubyのやさしい入門書	Show Edit Destroy
New Book		

▲図 3.13: index 画面

ビューのコードを見てみましょう。ファイルは `app/views/books/index.html.erb` です。抜粋して、コントローラで取得した@books を表示させるところを見てみましょう。

```
<% @books.each do |book| %> ← @booksはbookがいくつか入った配列
  <tr>
    <td><%= book.title %></td> ← 初回のbook.title "RubyとRailsの学習ガイド"
    <td><%= book.memo %></td> ← 初回のbook.memo "Rails関連技術地図とそれらの学習資料
の紹介"
    <td><%= link_to 'Show', book %></td>
    <td><%= link_to 'Edit', edit_book_path(book) %></td>
    <td><%= link_to 'Destroy', book, method: :delete, data: { confirm: 'Are you s
ure?' } %></td>
  </tr>
<% end %>
```

表示は`<% @books.each do |book| %>`から`<% end %>`で行っています。`@books`に代入された全データ ("Ruby と Rails の学習ガイド"、"Ruby 超入門") でブロックを繰り返し実行し、title や memo を表示したり、Show (詳細画面)、Edit (編集画面)、削除ボタンのリンクを生成します。

今回のケースでは、`@books`に 2 つのデータが入っています。`@books.each` 文は全データで繰り返し処理を行う文です。1 回目の実行では変数 `book` (2 つの|記号で囲まれているところ) に「Ruby と Rails の学習ガイド」のデータが入った状態となり、その title や memo が表示されます。2 回目の実行では「Ruby 超入門」のデータが格納されて同様の処理が実行されます。

`index.html.erb` は HTML のテンプレートファイルで、HTML の中に Ruby コードを埋め込むことができます。`<%= %>`で囲まれたコードは実行結果が HTML として出力されます。`<% %>` で囲まれたコードは実行されますが、結果を埋め込みません。今回は`@books.each` の繰り返しで`<% %>`が使われています。

Rails アプリはつくられた HTML をレスポンスとしてブラウザに返します。ブラウザは受け取った HTML を私たちが見れる形で表示します。

3.3 まとめ

この章では CRUD の 1 つ、一覧表示 (index 画面) の基礎的な内容について説明しました。リクエストを受けてレスポンスを返すまでの流れをまとめると以下のようになります。Rails アプリはリクエストを受けると、Routes、コントローラ、ビューの処理を経てレスポンス (HTML) を返します。



▲図 3.14: Rails アプリがリクエストを受けてレスポンスを返すまで (index アクション)

今回のリクエストは index (一覧) 画面を表示するものです。上記の処理の後、ブラウザは受け取った HTML を私たちが見れる形で表示します。

Books

Title	Memo	
RubyとRailsの学習ガイド	Rails関連技術地図とそれらの学習資料の紹介	Show Edit Destroy
Ruby超入門	Rubyのやさしい入門書	Show Edit Destroy
New Book		

▲図 3.15: index 画面

ポイントをまとめると以下のようになります。

- CRUD は Web アプリの基本となる 4 つの機能、新規作成 (Create)、表示 (Read)、更新 (Update)、削除 (Destroy)

- rails g scaffold は CRUD をつくる機能
- パスが /books、HTTP メソッドが GET のリクエストは一覧画面を表示する

次の章では CRUD の Create (新規作成) について説明します。

3.4 さらに学びたい場合は

- Rails ガイド: Rails のルーティング
 - routes についての詳しい解説です。
- Rails ガイド: Action Controller の概要
 - コントローラについての詳しい解説です。

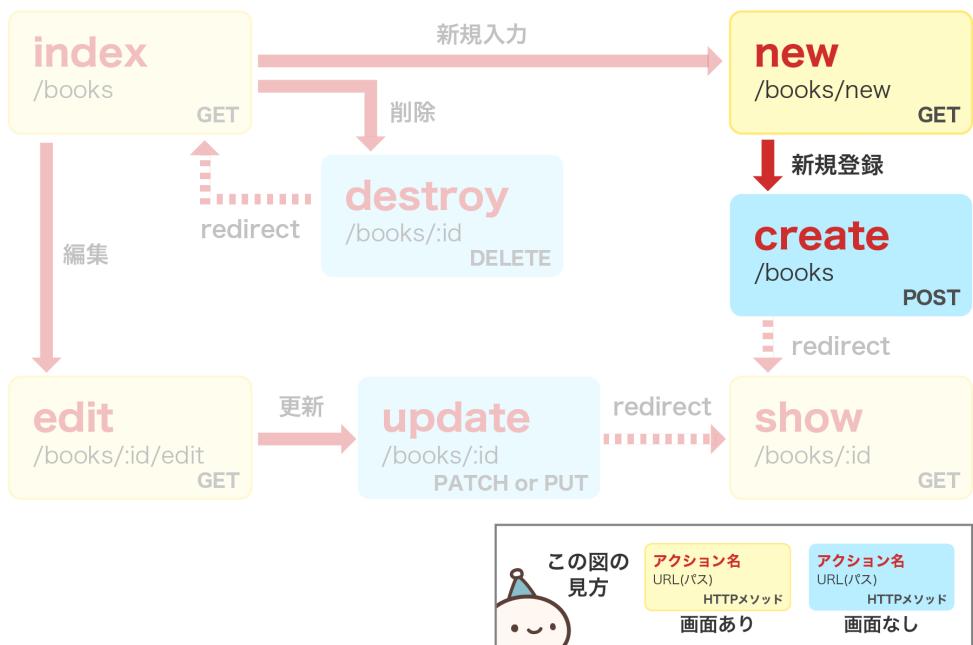
第4章 new, create アクション

この章では CRUD の C (Create) にあたる新規登録を行うための 2 つのアクション、new と create について説明していきます。

説明に使うアプリは前の章でつくったものを引き続き使います。

4.1 概略

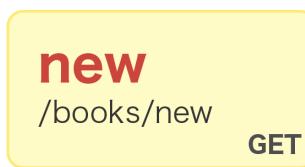
前の章での「CRUD 遷移図」において、new と create はこの部分になります。



▲図 4.1: CRUD 画面遷移図 (new と create)

新規登録は 2 つのアクションで構成されます。処理の流れは以下のようになります。

ステップ 1: new アクション (新規作成画面)



▲図 4.2: new アクション

new アクションが実行され新規入力画面を表示します。ここで本のタイトルとメモを入力します。Create Book ボタンを押すと、次に画面のない create アクションが実行されます。

ステップ 2: create アクション (画面なし)



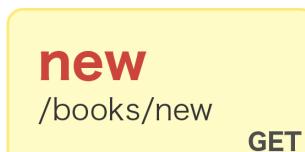
▲図 4.3: create アクション

つづく create アクションで、前のステップで入力されたタイトルとメモで本のデータを新規登録します。

この流れを詳しく説明していきます。最初に、new アクションの新規入力画面の処理を見てみましょう。

4.2 new アクション

新規入力画面が表示されるまでの処理の流れを見ていきましょう。



▲図 4.4: new アクション

New Book

Title

Memo

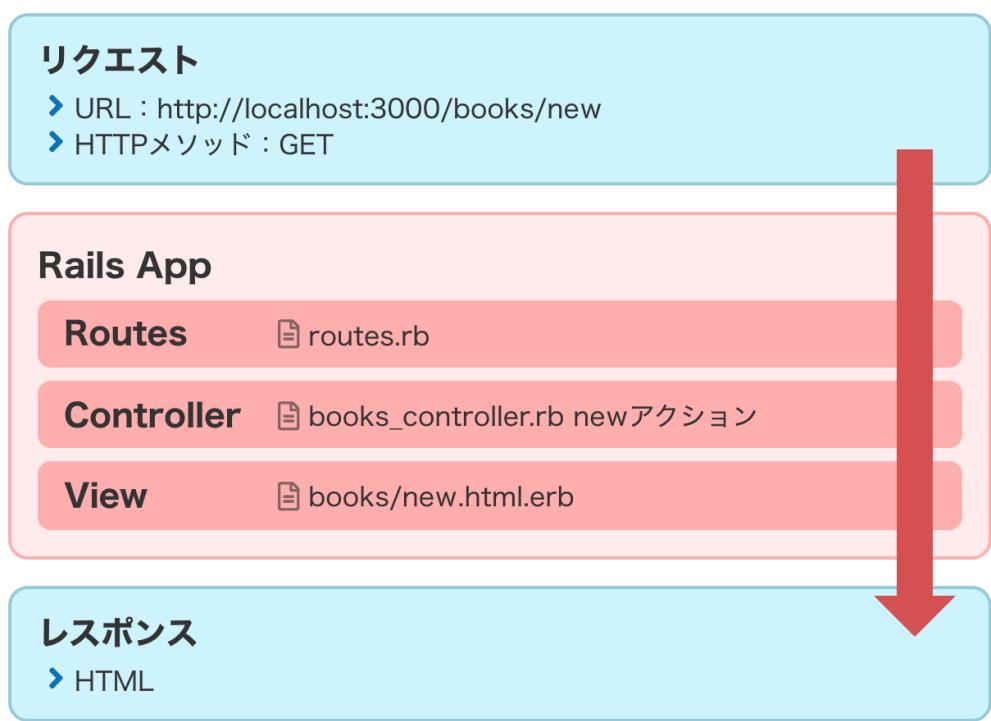
//

Create Book

Back

▲図 4.5: 新規入力画面

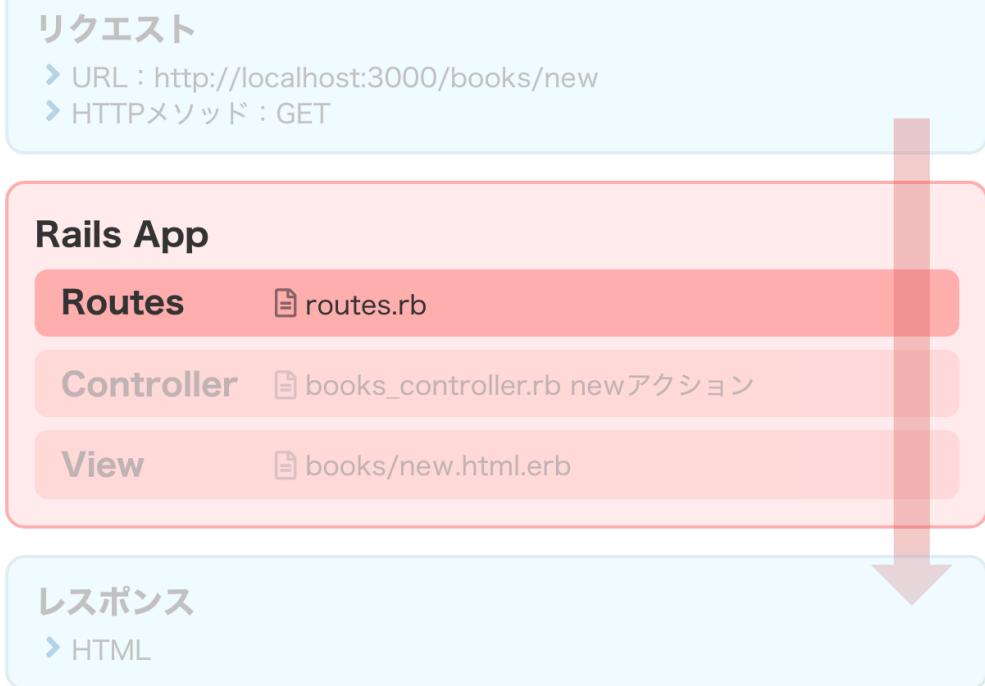
新規入力画面は new アクションで表示されます。Rails アプリはこれまでに説明した通り、Routes、コントローラ、ビューの各処理を経て画面が表示されます。



▲図 4.6: new(新規入力) 画面の処理の流れ

最初は Routes で処理が行われます。

Routes



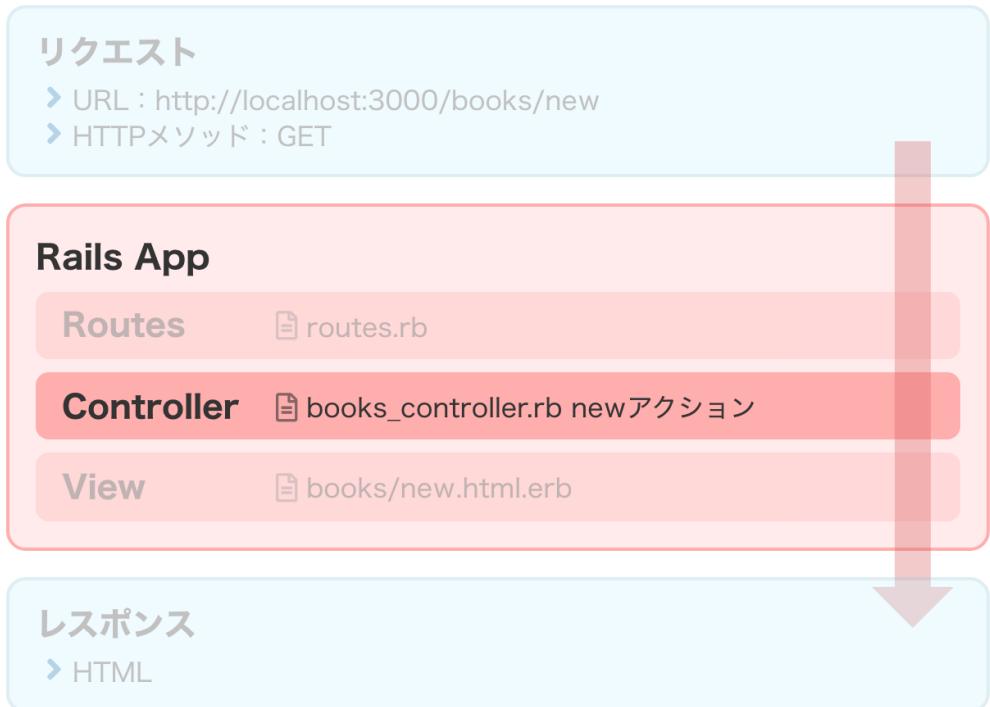
▲図 4.7: 新規入力画面の処理の流れ - Routes

Routes の対応表をつかって、リクエストに対して処理されるコントローラとアクションが決まります。<http://localhost:3000/rails/info/routes> へアクセスして Routes 表を見てみましょう。今回のリクエストはパスが /books/new 、HTTP メソッドが GET なので、BooksController の new アクションへ処理が進みます。Routes 表の下線部に該当します。

Helper	HTTP Verb	Path	Controller#Action
<u>Path / Url</u>		Path Match	
books_path	GET	/books(.:format)	books#index
	POST	/books(.:format)	books#create
new_book_path	<u>GET</u>	<u>/books/new(.:format)</u>	<u>books#new</u>
edit_book_path	GET	/books/:id/edit(.:format)	books#edit
book_path	GET	/books/:id(.:format)	books#show
	PATCH	/books/:id(.:format)	books#update
	PUT	/books/:id(.:format)	books#update
	DELETE	/books/:id(.:format)	books#destroy

▲図 4.8: Routes 表

コントローラ



▲図 4.9: 新規入力画面の処理の流れ - コントローラ

BooksController の new アクションのコードを見てみましょう。ファイルは app/controllers/books_controller.rb です。

```
def new
  @book = Book.new
end
```

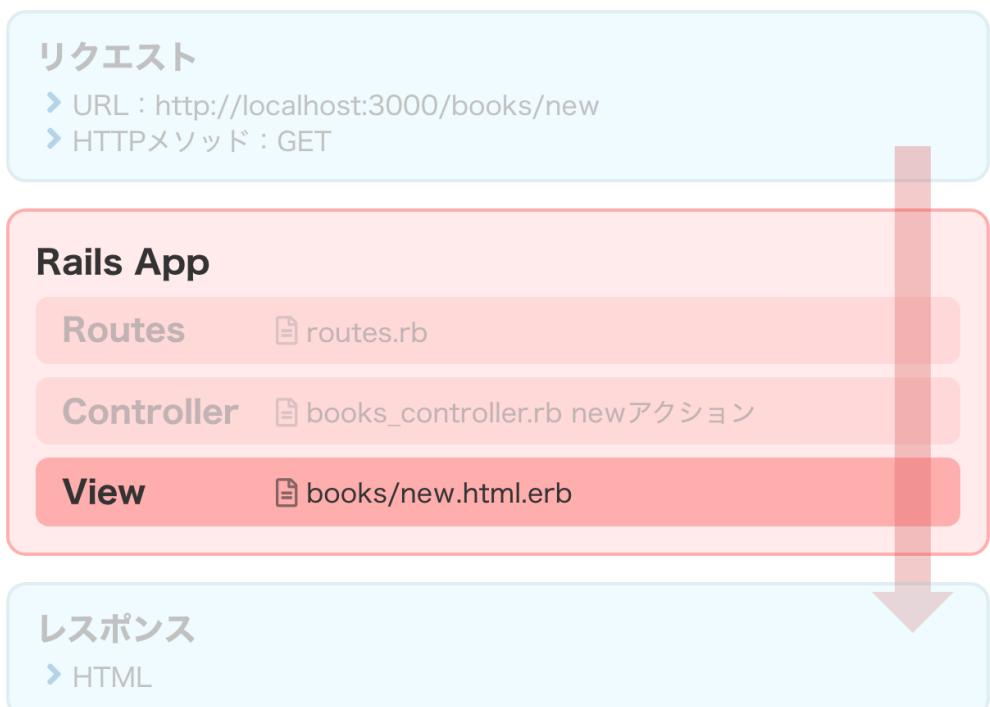
new アクションは `@book = Book.new` の 1 行です。Book.new で Book クラスのインスタンスを作り、@book インスタンス変数へ代入し、ビューへ渡します。Book.new でつくった Book クラスのインスタンスはタイトルとメモを格納できるようになっていますが、タイトルとメモのデータは空っぽです。

インスタンスとはクラスから作られたオブジェクトのことです。オブジェクトとほぼ同じ意味で使われますが、「クラスから作ったオブジェクトである」「そのクラスに属する」ということを強調したいときに使います。クラスはその種族に属するオブジェクト（インスタンス）を作ることができる工場のようなものです。そのクラス自身が仕事をすることもあれば、そのクラスから作ったオブジェクトが仕事をすることもあります。

Book クラスには色々と便利な機能があるのですが、それは後ほど説明します。ここでは、Book に関するビューで使う情報をつくり、インスタンス変数へ代入し、ビューへ送る、と考えると良いでしょう。

コントローラの処理が終わると次はビューです。ここでは進むビューの指示がないため、デフォルトである `views/books/new.html.erb` へ処理が進みます。

ビュー



▲図 4.10: 新規入力画面の流れ - ビュー

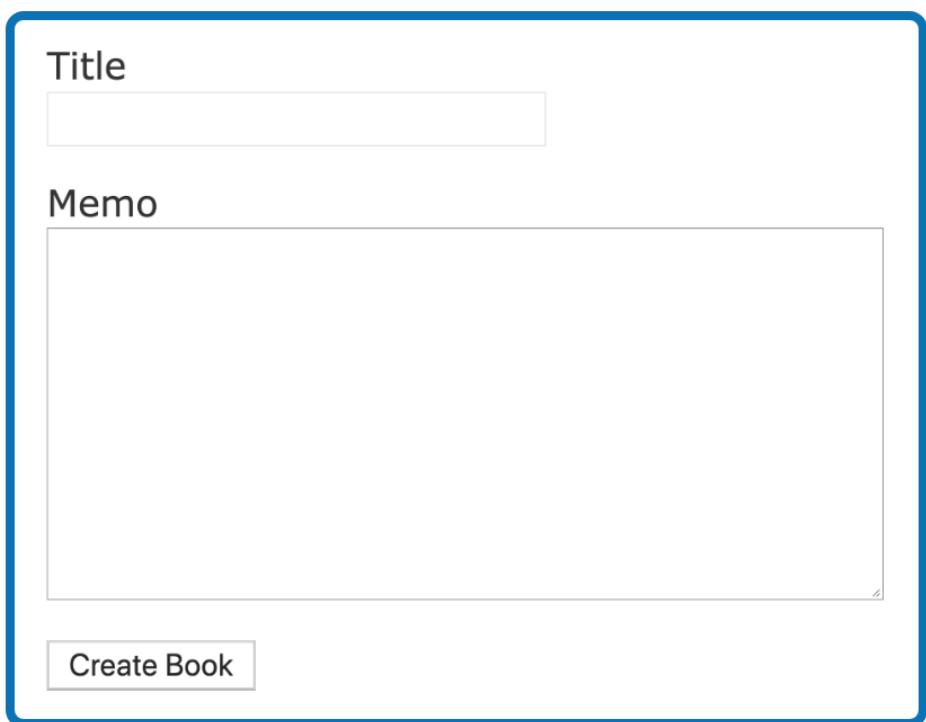
ビューのコード `views/books/new.html.erb` を見てみましょう。

```
<h1>New Book</h1>

<%= render 'form', book: @book %>

<%= link_to 'Back', books_path %>
```

これだけしかありません。随分とあっさりしています。実は、下の図中の枠線部分は別のファイルに書いてあり、`<%= render 'form', book: @book %>` で埋め込まれるようになっています。



▲図 4.11: render の説明

render メソッドは別のビューファイルを埋め込みます。わざわざ別のファイルに書く理由は、他の画面でもそのファイルを利用することで、同じ部品を共用したいからです。埋め込む用のビューファイルをパーシャルと言います。書式は以下の通りです。

```
<%= render 埋め込みたいファイル名, パーシャル内で使う変数名: 渡す変数 %>
```

埋め込むファイル名には 1 つルールがあり、render で書いた文字列の先頭に`_`を付けたファイル名にします。つまり、`<%= render 'form', book: @book %>`で埋め込まれるファイルは`_form.html.erb`になります。

また、`<%= render 'form', book: @book %>`の`book: @book`の部分は、`@book`変数を埋め込み先のパーシャル内で`book`変数として使うための指示です。パーシャル内でも`@`ははじまりのインスタンス変数を利用することも可能です。それでもわざわざ`book`変数として渡しているの

は、パーシャル内で利用する変数を明示すること、他のコントローラでパーシャルを流用する時にインスタンス変数名を揃える必要がないことなどのメリットがあります。

埋め込まれるパーシャルビュー _form.html.erb は以下のようになっています。ファイルは app/views/books/_form.html.erb です。

```
<%= form_with(model: book, local: true) do |form| %>
  <% if book.errors.any? %>
    <div id="error_explanation">
      <h2><%= pluralize(book.errors.count, "error") %> prohibited this book from >
    being saved:</h2>

      <ul>
        <% book.errors.full_messages.each do |message| %>
          <li><%= message %></li>
        <% end %>
      </ul>
    </div>
  <% end %>

  <div class="field">
    <%= form.label :title %>
    <%= form.text_field :title %>
  </div>

  <div class="field">
    <%= form.label :memo %>
    <%= form.text_area :memo %>
  </div>

  <div class="actions">
    <%= form.submit %>
  </div>
<% end %>
```

new.html.erb と _form.html.erb の 2 つのファイルでこの画面はつくられています。

では、_form.html.erb の中を解説していきます。この中で、2 行目からの以下の部分はエラーを表示するコードです。ここでは説明を省略して、それ以外の基本となる部分を説明します。

```
<% if book.errors.any? %>
  <div id="error_explanation">
    <h2><%= pluralize(book.errors.count, "error") %> prohibited this book from be>
  ing saved:</h2>
    <ul>
      <% book.errors.full_messages.each do |message| %>
        <li><%= message %></li>
      <% end %>
    </ul>
  </div>
<% end %>
```

残りの部分について説明していきます。次の図を見てください。

```
app/views/books/_form.html.erb
```

```
<%= form_with(model: book,
local: true) do |form| %>
  <div class="field">
    <%= form.label :title %>
    <%= form.text_field :title %>
  </div>

  <div class="field">
    <%= form.label :memo %>
    <%= form.text_area :memo %>
  </div>

  <div class="actions">
    <%= form.submit %>
  </div>
<% end %>
```

New book

Title

Memo

Create Book

Back

▲図 4.12: コードとページの部品の対応

それぞれ矢印の先の部品を作っています。また、全体としては form という名の部品になってます。form は HTML でブラウザからサーバへ情報を送信する仕組みです。

まずは部品の1つ、タイトルのところを見てみましょう。

```
app/views/books/_form.html.erb
```

```
<%= form_with(model: book,
local: true) do |form| %>
  <div class="field">
    <%= form.label :title %>
    <%= form.text_field :title %>
  </div>
...

```

New book

Title

▲図 4.13: タイトル部品

枠線内がタイトルの部分です。全体は HTML ですが、`<%= %>`で囲まれた部分がその中に埋め込まれた Rails コードです。

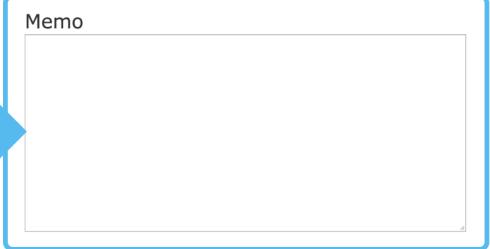
`<div></div>`は中の HTML 要素をグルーピングするための要素です。それだけだと特に見た目を変えませんが、CSS で修飾する要素を指定するためによく使います。ここでは "field" という

HTML での class 名をつけて CSS で修飾できるようにしています。

Rails コードの部分をもう少し詳しく見てみましょう。`form.label :title` で "Title" という文字列を表示しています。その名の通り、ラベルの部分です。`form.text_area :title` はその下にあるテキスト入力欄です。`form` は form ブロック内の変数で、ここでは book に関する form を記述するために使っています。見慣れない書き方かもしれません、ここはそう書くものだと思ってもらえば大丈夫です。

`app/views/books/_form.html.erb`

```
<%= form_with(model: book,
  local: true) do |form| %>
...
<div class="field">
  <%= form.label :memo %>
  <%= form.text_area :memo %>
</div>
...
...
```



▲図 4.14: メモ部品

メモの部分も同様です。`form.label :memo` が "Memo" を表示する部分です。`form.text_area :memo` がその下のテキスト入力欄を作ります。`text_area` は先ほどの `text_field` よりも広くて改行を入力できるテキスト入力欄を作るメソッドです。

`app/views/books/_form.html.erb`

```
<%= form_with(model: book,
  local: true) do |form| %>
...
<div class="field">
  <%= form.submit %>
</div>
...
...
```



▲図 4.15: 投稿ボタン部品

最後は投稿するボタンの部分です。`form.submit` は投稿ボタン (Create Book ボタン) を作ります。このボタンを押すと form 内の情報をまとめてサーバへ送信 (リクエストを送信) します。

ここでは、Bookに関する情報、入力したタイトルとメモをリクエストに含んで送信します。ここで送信されたタイトルとメモが後の行程で登録されます。

では、具体的にどんなリクエストが飛ぶのかを観察してみましょう。

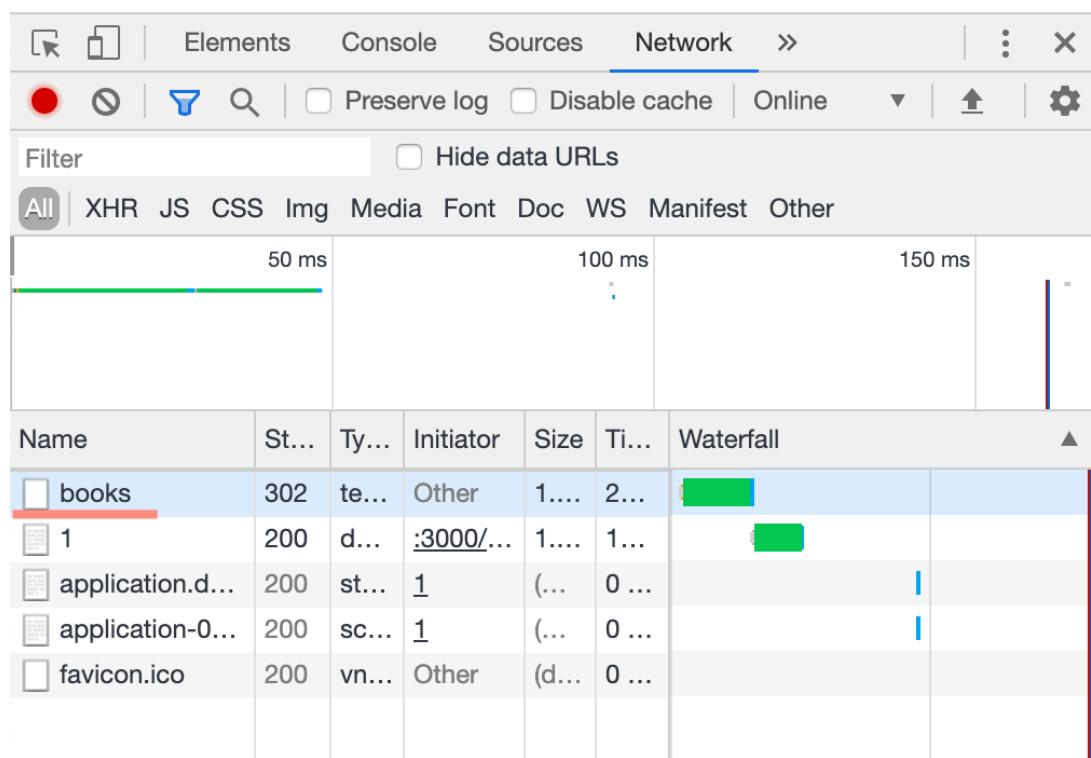
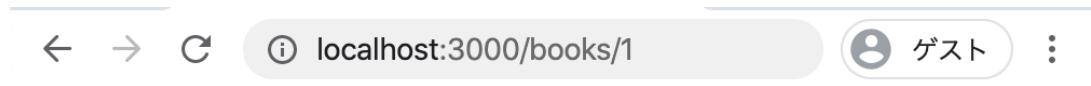
リクエストを観察する

Chromeのデベロッパーツールを使うと、どのようなリクエストがサーバへ送信されたかを見るすることができます。

new画面を表示させ、タイトル欄とメモ欄にBookの情報を入力します。Chromeのメニューからデベロッパーツールを起動します。Networkと書かれたタブを選択します。CreateBookボタンを押し、リクエストを送信してみましょう。

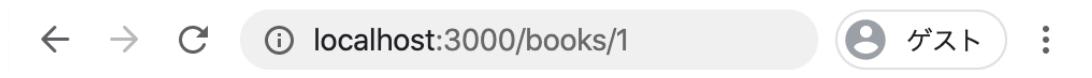
The screenshot shows a browser window with the URL `localhost:3000/books/new`. The page title is "New Book". There are two input fields: "Title" containing "RubyとRailsの学習ガイ" and "Memo" containing "Rails関連技術地図とそれらの学習資料の紹介". Below the inputs are two buttons: "Create Book" and "Back". A red annotation "メニュー - 他のツール - デベロッパーツール" is overlaid on the right side of the page. At the bottom, the Chrome developer tools Network tab is selected, showing a list of requests. The first request in the list has a duration of 20 ms.

▲図 4.16: Chrome でリクエストを観察する (準備)



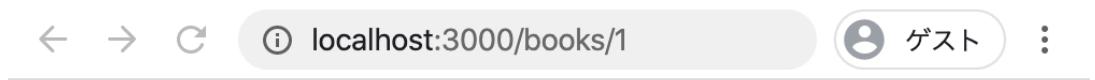
▲図 4.17: Chrome でリクエストを観察する (リクエスト送信)

たくさん表示されました。一番最初の books と書かれた行が先ほどボタンを押して発行されたリクエストです。books の行をクリックして詳細を見てみましょう。

A screenshot of the Chrome DevTools Network tab. The tab is active, showing a timeline of network requests. A POST request to 'http://localhost:3000/books' is selected, showing its details. The 'Headers' tab is selected, displaying the following information:

- Request URL: http://localhost:3000/books
- Request Method: POST
- Status Code: 302 Found
- Remote Address: [::1]:3000
- Referrer Policy: strict-origin-when-cross-origin

The 'Response' tab is also visible. At the bottom of the Network tab, it shows '5 requests | 2.8 KB transferred'.



Book was successfully created.

Title: RubyとRailsの学習ガイド

Memo: Rails関連技術地図とそれらの学習資料の紹介

[Edit](#) | [Back](#)

A screenshot of the Chrome DevTools Network tab. The tab is active and shows a timeline of requests. One request for 'books' is highlighted, showing a response time of 50 ms. The response body is displayed, showing the created book's details: title 'RubyとRailsの学習ガイド', memo 'Rails関連技術地図とそれらの学習資料の紹介', and a note '料の紹介'. The 'Form Data' section shows the posted data: 'authenticity_token' (a long token), 'book[title]' (the title), 'book[memo]' (the memo), and 'commit' (the commit type).

Name	Headers	Preview	Response
books			<p>Upgrade-Insecure-Requests: 1</p> <p>User-Agent: Mozilla/5.0 (Macintosh; Intel Mac OS X 10_15_2) AppleWebKit/537.36 (KHTML, like Gecko) Chrome/79.0.3945.88 Safari/537.36</p>
1			
application.debug-e31ba9c2...			
application-0b67a3cc2fcde6...			
favicon.ico			

▼ Form Data

authenticity_token: RaJ79r9q18pB60/6NzthDwr8cS1M3jkC9F/Q+4rr0wb2UrnW4TiD2JmwnaKarupbtK
KivYt6QZtrI3yrtG7QJg==

book[title]: RubyとRailsの学習ガイド

book[memo]: Rails関連技術地図とそれらの学習資料の紹介

料の紹介

commit: Create Book

5 requests | 2.8 KB transferred

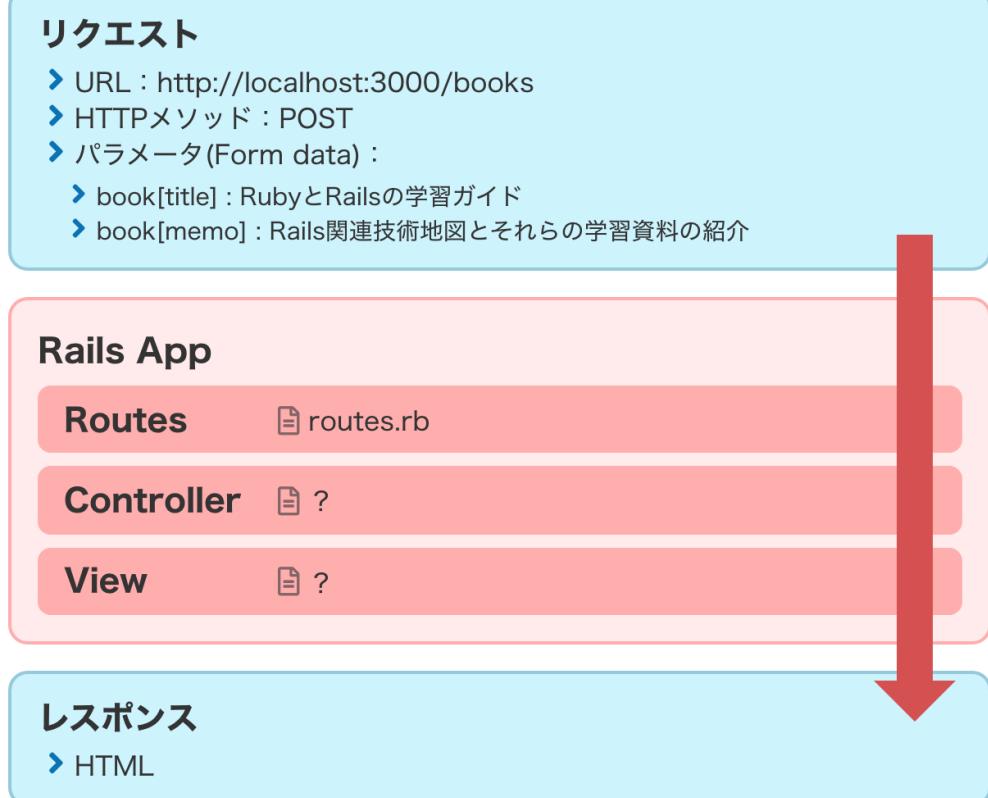
最初に URL と HTTP メソッドが書いてあります。Routes で使う情報がここに載っています。下の方へスクロールすると、Form Data という欄に book[title] と book[memo] の情報を見つけることができます。さきほど new 画面で入力した内容がここに表示されていることを確認してみてください。

次は、飛んだこのリクエストがどのように処理されるかを見ていきましょう。

4.3 Create アクション

新たなリクエスト

new 画面で Create book ボタンを押すと新たなリクエストを飛ばすことが分かりました。ここからは、この 2 つ目のリクエストを追いかけます。リクエストの内容は、さきほど Chrome で確認したように次の図のようになっています。



▲図 4.20: 新たなリクエスト

Routes

いつものように最初の処理は routes です。

リクエスト

- › URL : http://localhost:3000/books
- › HTTPメソッド : POST

Helper	HTTP Verb	Path	Controller#Action
<u>Path / Url</u>		Path Match	
books_path	GET	/books(.:format)	books#index
	<u>POST</u>	<u>/books(.:format)</u>	<u>books#create</u>
new_book_path	GET	/books/new(.:format)	books#new
edit_book_path	GET	/books/:id/edit(.:format)	books#edit
book_path	GET	/books/:id(.:format)	books#show
	PATCH	/books/:id(.:format)	books#update
	PUT	/books/:id(.:format)	books#update
	DELETE	/books/:id(.:format)	books#destroy

▲図 4.21: routes

URL のパスは /books 、 HTTP メソッドは POST なので対応するコントローラとアクションは books#create、つまり BooksController の create アクションが呼び出されます。

HTTP メソッドの POST は今回のようなデータの新規作成時に使います。そのほか、サーバの状態へ何らかの変更を与えるときにはこの POST を利用します。

一方で、index や new の時に利用した HTTP メソッド GET は、サーバの状態を変えない場合に使います。new アクションでは新規入力画面を表示するだけでまだデータを保存しないので、HTTP メソッドは GET を使うのです。

コントローラ

リクエスト

- URL : http://localhost:3000/books
- HTTPメソッド : POST
- パラメータ(Form data) :
 - book[title] : RubyとRailsの学習ガイド
 - book[memo] : Rails関連技術地図とそれらの学習資料の紹介

Rails App

Routes routes.rb

Controller books_controller.rb createアクション

View ?

レスポンス

- HTML

▲図 4.22: 新たなリクエスト

コントローラのソースファイルは app/controllers/books_controller.rb です。ここでやっていることは大きく3つです。

```
def create
  @book = Book.new(book_params) # ← 1. リクエストのパラメータを使って本のデータを作る
  respond_to do |format|
    if @book.save # ← 2. 本のデータを保存する
      # ← 3a. 成功したらshow画面へ
      format.html { redirect_to @book, notice: 'Book was successfully created.' }
      format.json { render :show, status: :created, location: @book }
    else
      # ← 3b. 保存失敗したらnew画面へ(元の画面)
      format.html { render :new }
      format.json { render json: @book.errors, status: :unprocessable_entity }
    end
  end
end
```

```
end
```

3つの処理を順に見ていきます。最初は `@book = Book.new(book_params)` です。

```
def create
  @book = Book.new(book_params)
  ...

```

`Book.new` メソッドの引数に渡している `book_params` はメソッドを呼び出しています。このメソッドはファイルの後半に定義されています。`book_params` の中を見てみましょう。

パラメータ

`app/controllers/books_controller.rb`

```
def book_params
  params.require(:book).permit(:title, :memo)
end
```

`book_params` メソッドはパラメータに関する処理を行っています。パラメータとはブラウザから飛んでくるリクエストの中に含まれる情報で、たとえばユーザーの入力した値が入っています。さきほど Chrome デベロッパーツールを使って見たものが、Rails まで渡ってきています。

パラメータは `params` で取得できます。次は `params` にどんな情報が、どのように入っているかを見てみましょう。次のようにコードを変更してみてください。

`app/controllers/books_controller.rb`

```
def book_params
+ p "*****" # 見つけ易くするための目印。何でも良い。
+ p params # paramsの中身を表示
  params.require(:book).permit(:title, :memo)
end
```

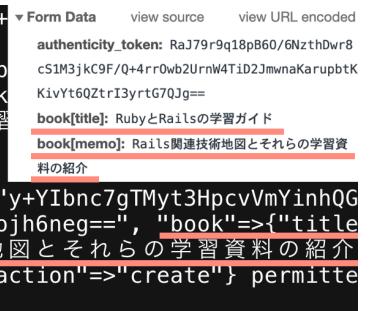
コードを変更して、ブラウザから新規登録画面を表示し、テキストボックス欄に入力し、Create Book ボタンを押します。その後、ターミナルに流れた rails server の文字列から*****を探してみてください。表示されていない場合は、rails server を再起動してみてください。

```
... (略)
Started POST "/books" for ::1 at 2020-01-04 20:07:46 +0900
Processing by BooksController#create as HTML
Parameters: {"authenticity_token"=>"y+YIbnc7gTMyt3HpcvVmYinhQG9NrhM63B0aXzvc3QI>MgSR0QkwH/rsmunFt0qni7SGjlfZ9NIenuevojh6neg==", "book"=>{"title"=>"RubyとRails の学習ガイド", "memo"=>"Rails関連技術地図とそれらの学習資料の紹介"}, "commit"=>"Create Book"}
*****
<ActionController::Parameters {"authenticity_token"=>"y+YIbnc7gTMyt3HpcvVmYinhQG9>NrhM63B0aXzvc3QIMgSR0QkwH/rsmunFt0qni7SGjlfZ9NIenuevojh6neg==", "book"=>{"title"=>
```

```
>"RubyとRailsの学習ガイド", "memo"=>"Rails関連技術地図とそれらの学習資料の紹介"}, "commi>t"=>"Create Book", "controller"=>"books", "action"=>"create"} permitted: false>... (略)
```

実行結果を見ると、確かに `params` の中にブラウザにて入力した値が Hash の形で入っていることが分かりました。

これを、少し前にブラウザのデベロッパーツールで表示させた内容と比較してみましょう。

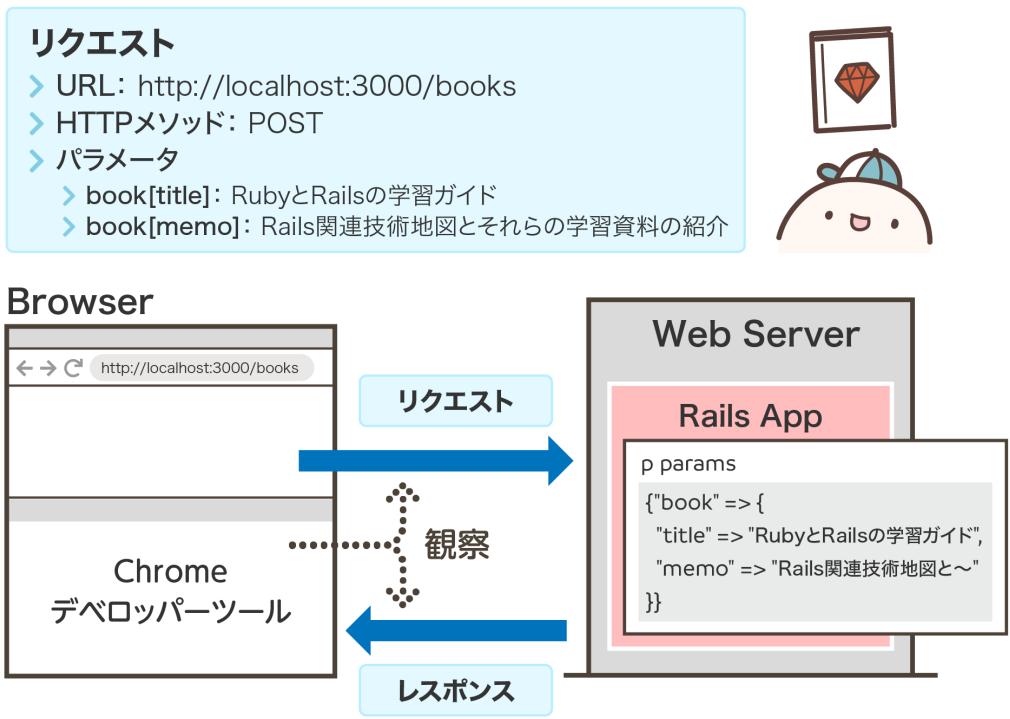


```
Started POST "/books" for ::1 at 2020-01-04 20:07:46 +0900
Processing by BooksController#create as HTML
Parameters: {"authenticity_token"=>"y+YIbnc7gTMyt3HpcvVmYinhQG9NrhM63B0aXzvc3QIMgSR0QkwH/rsmunFt0qni7SGjlfZ9NIenuevojh6neg==", "book"=>{"title"=>"RubyとRailsの学習ガイド", "memo"=>"Rails関連技術地図とそれらの学習資料の紹介"}, "commit"=>"Create Book", "controller"=>"books", "action"=>"create"} permitted: false>
```

▲ 図 4.23: パラメータの送信側と受信側

ここで出力した `params` の値と、さきほどブラウザのデベロッパーツールで表示させたパラメータの値が同じになっていることが分かります。

次の図はさきほど試した、デベロッパーツールで Rails アプリへのリクエストを観察したときの様子を図にしたものです。ブラウザのデベロッパーツールで見たものは、送信するリクエストのパラメータです。一方で Rails アプリで出力したパラメータは、リクエストを受信して Rails アプリで処理している部分です。ブラウザがユーザーの入力データをパラメータとして送信し、私たちが作成しているアプリがそのデータを受け取っていることを確認できました。



▲図 4.24: ブラウザとデベロッパーツール

Strong Parameters

book_params の説明に戻ります。params の後ろについている、require と permit とはなんでしょうか？

app/controllers/books_controller.rb

```
def book_params
  params.require(:book).permit(:title, :memo)
end
```

params 以降の require, permit メソッドは、パラメータの内容を制限します。意図していないデータが入ってくるのを防ぐための仕組みです。ここでは、book の title, memo だけを受け取るようにしています。require には対象となるモデル名（モデルについては次章で説明します）を、permit には更新を許可するカラム名を指定します。

このパラメータを制限する仕組みは Strong Parameters と呼ばれます。これが必要な理由は、攻撃に対する防御、つまりセキュリティ対策です。ブラウザから飛ばすパラメータは、ユーザーの手によって改ざんすることも可能です。つまり、任意のパラメータを飛ばして攻撃をすることもで

きます。そのため、1つ前の new 画面で用意した form に存在しないパラメータが飛んでくる可能性もあるので、ここで変更を許可するパラメータを絞っています。

たとえば、titleだけを更新したいケースがあり、titleだけを更新する form をつくったとします。Book.new(book_params) で new メソッドは、引数で受け取った値を自分のカラムへ代入します。このとき、book_params に title の情報だけがやってくればいいのですが、攻撃者は memo の情報もパラメータとして飛ばすこともあります。StrongParameters で受け取り可能なパラメータを絞っていないと、プログラマの意図しないカラムが更新されてしまうことになります。

4.4 まとめ

create アクションでの処理について説明してきました。create アクション全体の中で、どこまで進んだかを確認してみましょう。

`app/controllers/books_controller.rb`

```
def create
  @book = Book.new(book_params) # ← 1. リクエストのパラメータを使って本のデータを作る
  respond_to do |format|
    if @book.save # ← 2. 本のデータを保存する
      # ← 3a. 成功したらshow画面へ
      format.html { redirect_to @book, notice: 'Book was successfully created.' }
      format.json { render :show, status: :created, location: @book }
    else
      # ← 3b. 保存失敗したらnew画面へ（元の画面）
      format.html { render :new }
      format.json { render json: @book.errors, status: :unprocessable_entity }
    end
  end
end
def book_params
  params.require(:book).permit(:title, :memo)
end
```

この章では1.まで、create アクションにパラメータ (params) が届いたのを確認したところで説明しました。params でパラメータの情報を取り、StrongParameters を使って必要なものだけに制限します。

Book.new(book_params) で本のデータを作ります。new はクラスのインスタンスを作るメソッドです。実は Book は「モデル」という種族に属する便利な機能を持ったクラスです。モデルについての説明は次の章で行います。

このあと、本の情報を保存し(2.の部分)、その結果により表示する画面を切り替えます(3.の部分)。続きは次の章で説明します。

この章のまとめです。

- 新規入力画面は new アクション、新規登録は create アクション
- new アクションではまだデータを保存せず、サーバのデータ変更を伴わないため HTTP メ

ソッド GET を使う

- create アクションではデータを保存し、サーバのデータ変更を伴うため HTTP メソッド POST を使う
- ユーザーがブラウザで form へ入力した内容はリクエスト内のパラメータとして Rails アプリへ届き、params で渡ってきたパラメータを取得できる
- セキュリティ問題対策のため StrongParameters (require メソッド、permit メソッド) を利用して params に制限をかける

次の章ではモデルについて説明します。

4.5 さらに学びたい場合は

- Rails ガイド: Action Controller の概要
 - 前章でも紹介した、コントローラについての詳しい解説のページです。StrongParameters に関する説明もここに書いてあります。

第5章 モデル

この章ではデータの長期保存と、その際に使うモデルについて説明します。

説明に使うアプリは前の章でつくったものを引き続き使います。

5.1 データの永続化

コード中で変数に代入したデータは、変数の有効範囲（スコープと言います）が終わると消えてしまします。次のコードを見てみましょう。

```
def print_hello_world
  x = "Hello world!"
  puts x
end
```

ここで変数 `x` のように、`@`などの記号が変数名の先頭についていない変数をローカル変数と呼びます。変数 `x` の有効範囲（スコープ）は、そのメソッドの中だけです。この場合、メソッド `print_hello_world` の実行が終わると変数 `x` と、それが指すオブジェクトは消えてしまいます。

別の種類の変数として、前の章でも出てきた`@book`のようなインスタンス変数（名前が`@`はじめの変数）があります。インスタンス変数は、インスタンス（オブジェクト）がなくなると一緒に消えます。Rails の場合は、1つのリクエスト内が有効範囲だと考えることができます。ここではおおまかに、あるページをブラウザで表示することが1回のリクエストだと考えて差し支えありません。別のページを表示したり、リロードを行ったりすると、新しい別のリクエストになります。インスタンス変数に代入すると、コントローラからビューまで使うことができますが、その後なくなります。

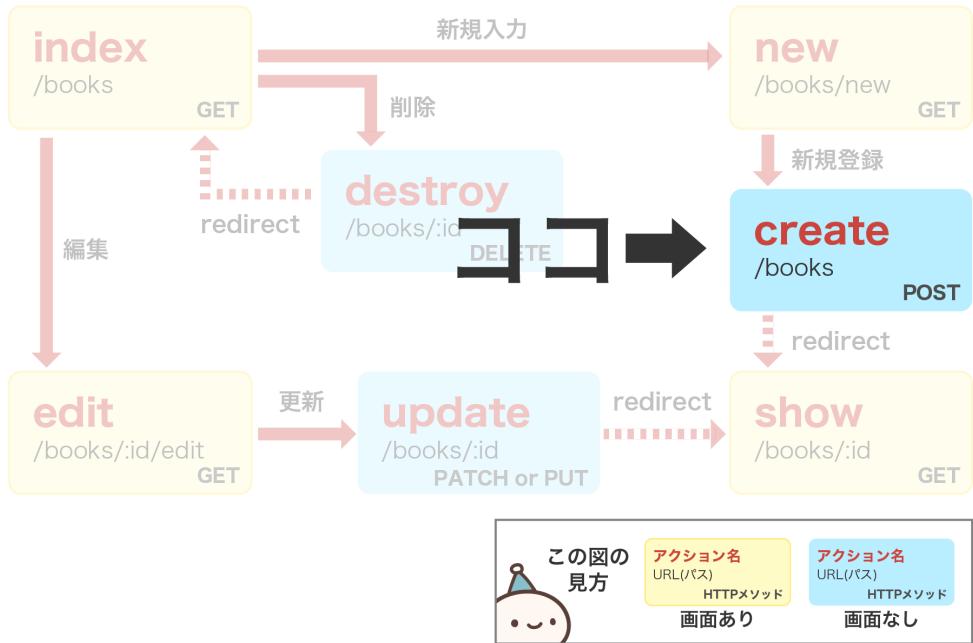
それでも、つくった Rails アプリでは別のリクエストでも情報が見れますよね？ 新規入力画面で情報を入力して一度登録してしまえば、その後ブラウザから何回アクセスしても、たとえば一覧画面で何回リロードしても表示されます。

つまり、複数のリクエストに渡ってデータが保存されていることになります。これはインスタンス変数では実現できません。データがずっと残っているのは「データを保存する」仕事をしている「何かの仕組み」があるからなのです。

その仕組みがこの章の主役、モデル（Model）です。

この章はモデルについて説明します。また、この章では前の章につづいて CRUD の create の部分を題材に説明します。

この章はここを説明します。



▲図 5.1: この章の題材

処理の流れはこの部分です。前の章でコントローラの処理の途中まで説明したので、そのつづきになります。



▲図 5.2: 処理の流れ

コードではこの部分です。

app/controllers/books_controller.rb

```
def create
  @book = Book.new(book_params) # ← 1. リクエストのパラメータを使って本のデータを作る
  respond_to do |format|
    if @book.save # ← 2. 本のデータを保存する
      # ← 3a. 成功したらshow画面へ
      format.html { redirect_to @book, notice: 'Book was successfully created.' }
      format.json { render :show, status: :created, location: @book }
    else
      # ← 3b. 保存失敗したらnew画面へ（元の画面）
      format.html { render :new }
      format.json { render json: @book.errors, status: :unprocessable_entity }
    end
  end
end
```

このコードの中で、モデルを利用しているところはこの部分です。

```
def create
  @book = Book.new(book_params) # ← BookクラスはModelという種族に属する
  respond_to do |format|
    if @book.save # ← ここで保存
      format.html { redirect_to @book, notice: 'Book was successfully created.' }
      format.json { render :show, status: :created, location: @book }
    else
      format.html { render :new }
      format.json { render json: @book.errors, status: :unprocessable_entity }
    end
  end
end
```

では、モデルの仕事について見てきましょう。

5.2 モデルの基本的な使い方 その1 保存

モデルを使うとデータを保存することができます。以下の手順で保存できます。

```
book = Book.new(title: "RubyとRailsの学習ガイド", memo: "Rails関連技術地図とそれらの学習資料の紹介")
book.save!
```

Book.new で Book モデルオブジェクトを作ります。このとき、タイトルとメモの情報を渡すことができます。

なお、モデル名（モデルのクラス名）は英語の单数形、大文字始まりにするルールがあります。ここでは Book がモデル名で、单数形で大文字始まりになっています。また、すべて小文字の book は変数名です。1つの Book モデルオブジェクトが代入されるため单数形を使います。

Book モデルオブジェクトの save! メソッドを呼び出すと保存できます。保存するメソッドはいくつかあり、他にも save,create!, create メソッドなどがあります。save と save! の違いは、なんらかの理由で保存できなかったときの動作です。保存に失敗したとき、save メソッドは false を返し、save! メソッドでは例外が投げられます。create と create! の違いも同様です。

5.3 モデルの基本的な使い方 その2 読み込み

さきほど保存したデータを読み込んでみましょう。

```
books = Book.all.to_a
```

Book.all で保存されている Book モデルの全データを取得できます。以前に説明した一覧画面(index アクション) でこのメソッドが使われています。

Book.all.to_a で Array (配列) に Book オブジェクトが詰まって返ってきます。to_a は

Array オブジェクトへ変換するメソッドです。代入される変数名は小文字の books で、複数の Book オブジェクトが代入されるので複数形を使います。

5.4 モデルの基本的な使い方 その3 検索

```
book = Book.where(title: "RubyとRailsの学習ガイド").first  
book.title #=> "RubyとRailsの学習ガイド"  
book.memo #=> "Rails関連技術地図とそれらの学習資料の紹介"
```

where メソッドを使うと検索ができます。タイトルが"ハチミツとクローバー"である Book オブジェクトが返ります。検索結果が複数になることもあるので、first メソッドで最初の 1 つを取得しています。取得するオブジェクトは 1 つなので、それを代入する変数名 book は単数形になっています。

取得した Book オブジェクトは title メソッドでタイトルを、memo メソッドでメモをそれぞれ返します。

5.5 実習 : rails console でモデルを使う

Rails には "rails console" という、1 行ずつ入力したコードを実行する機能があります。Ruby が持っている irb をつかって、1 行ずつ入力した Rails のコードを実行することができます。つくったアプリがあるフォルダへ移動して、rails console を使ってみましょう。ターミナルで rails c と実行してみてください。c は console の頭文字です。メッセージ中"in process 53813"の数字は実行するたびに異なります。

```
cd books_app  
rails c
```

TODO: 最新 Rails バージョンで置き換え

```
$ rails c  
Running via Spring preloader in process 16386  
Loading development environment (Rails 6.0.0)  
irb(main):001:0> exit
```

rails console で以下のコードを実行してみてください。

```
book = Book.new(title: "some title", memo: "some memo") # モデルオブジェクト作成  
book.save! # 保存  
Book.last # 上で保存したデータの表示
```

前に出てきた where での検索も試してみてください。rails console を終了するときは exit と打ってください。

また、rails console で保存したデータは、ブラウザで保存したデータと同じ場所に格納され、同じように取り扱われます。ここで保存したデータはブラウザからも見ることができます。

```
rails s
```

ブラウザで `http://localhost:3000/books` へアクセスして、rails console から保存したデータが表示されることを確認してみましょう。

5.6 モデルの仕組み

次は、モデルのコードを見てみましょう。モデルのコードは `app/models/` フォルダ以下にあります。Book モデルのコード `app/models/book.rb` を見てみましょう。

```
class Book < ApplicationRecord
end
```

Book モデルにはコードがほとんどありません。

では、`save` や `all` といったメソッドが使えるのはなぜでしょうか？ また、`title` や `memo` の情報があることをどこで知るのでしょうか？

問: `save` や `all` といったメソッドが使えるのはなぜでしょうか？

答えは `ApplicationRecord` クラスを継承しているからです。`ApplicationRecord` クラス（およびさらに親のクラス）がモデルの仕事に必要な機能を持っています。それを継承している `Book` クラスも同じ機能を持ちます。

問: `title` や `memo` といった要素があることをどこで知るのでしょうか？

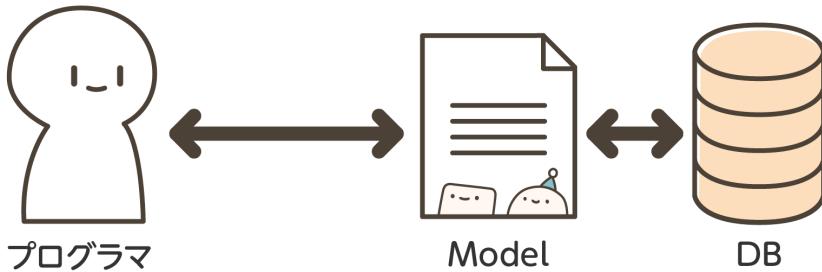
答えは「データベースから情報を得る」です。

`ApplicationRecord` はデータベースから情報を得て、`Book` モデルに `title`、`memo` という要素があることを知っています。その情報を使って、`book.title` や `book.memo` といったメソッドを提供します。

では、データベースとは何でしょうか？

データベース（DB）とは？

DB とは、データを保存したり、読み出したり、検索したりするために特化したプログラムです。モデルは DB を便利につかうための仕組みでもあります。DB は高機能で堅牢でかつ高速です。しかし、DB へアクセスするには一般に専用の言語（SQL と言います）を用いることが多く、Ruby のコードからは扱いづらいという問題があります。モデルは Ruby で DB を容易に扱うことができる機能も提供します。



Modelが簡単にDBにアクセスできる機能を提供するので、
プログラマはModelを通じてDBとデータをやりとりできる。

▲図 5.3: DB とは？

では、DBはいつのまに作られたのでしょうか？

5.7 DBはいつ作られたのか？

books app を作る一連のコマンドを入力した際に、以下のコマンドで DB を作成していました。

```
rails g scaffold book title:string memo:text  
rails db:migrate
```

scaffold コマンドはいろいろなを作りますが、その 1 つとして DB の設計書「マイグレーション（migration）ファイル」をつくります。

```
rails g scaffold book title:string memo:text  
...  
invoke  active_record  
create   db/migrate/20191008234834_create_books.rb  
create   app/models/book.rb  
...
```

そして rails db:migrate コマンドを実行すると、さきほど作られた設計書を使って DB を作ります。

設計書である migration ファイルから順に見ていきましょう。

5.8 マイグレーション（migration）ファイル

DB の設計書をマイグレーションファイルと呼びます。マイグレーションファイルも Rails

(Ruby) のコードで書かれています。

では、マイグレーションファイルを見てみましょう。ファイル名には"20191008234834"と実行した日付から作られた数字が入っているので、rails g コマンドを実行するごとに異なるファイル名になります。

```
db/migrate/20191008234834_create_books..rb
```

```
class CreateBooks < ActiveRecord::Migration[6.0]
  def change
    create_table :books do |t|
      t.string :title
      t.text :memo

      t.timestamps
    end
  end
end
```

1行目 `ActiveRecord::Migration[6.0]` の末尾にある数字は Rails のバージョンを表します。Rails6.0.x ではこのように 6.0 となります。

3行目 `create_table :books` で books という名前のテーブルを作ります。DB はテーブルという単位でデータを管理します。ここでは、本に関するデータを保存するために books という名前のテーブルを作っています。テーブル名はモデル名の複数形にするというルールがあります。

4行目 `t.string :title` と 5行目 `t.text :memo` はテーブルの要素を作成しています。これで books テーブルには title という要素と、memo という要素を持つことになります。この要素のことを DB の用語でカラムといいます。string はデータの型の 1つです。文字列を格納します。text もデータの型の 1つで、string よりも長い文字列を格納できる型です。

それに続く `t.timestamps` は、`created_at` (作成日時)、`updated_at` (更新日時) を記録するカラムを作ります。

DB のテーブルは Excel をイメージすると分かり易いです。DB へデータを格納していくことは、title、memo といった列がある Excel のシートに1行ずつデータを格納していくイメージです。

	A	B	C	D
1	title	memo	created_at	updated_at
2	RubyとRailsの学習ガイド	Rails関連技術地図とそれらの学習資料の紹介	2019/12/12 7:52	2019/12/12 7:52
3	Ruby超入門	Rubyのやさしい入門書	2019/12/15 7:13	2019/12/15 9:32

▲ 図 5.4: books テーブルのイメージ

ところで、`t.string :title` や `t.text :memo` に見覚えがありませんか？

これらは scaffold のコマンドで書かれていました。実は、scaffold で指定していたのはテーブル名とカラム、データの型でした。scaffold では、books テーブルに string 型である title カラムと text 型である memo カラムを作成する、という指示を与えていたことになります。

ここまでで、scaffold は DB 設計書 (migration) を作ることを説明しました。

```
$ rails g scaffold book title:string memo:text
```



ファイル生成

db/migrate/20191008234834_create_books.rb

```
class CreateBooks < ActiveRecord::Migration[6.0]
  def change
    create_table :books do |t|
      t.string :title
      t.text :memo

      t.timestamps
    end
  end
end
```

▲図 5.5: scaffold は migration を作る

では、migration から実際に DB を作るにはどうすれば良いでしょうか？

```
$ rails g scaffold book title:string memo:text
```



ファイル生成

db/migrate/20191008234834_create_books.rb

```
class CreateBooks < ActiveRecord::Migration[6.0]
  def change
    create_table :books do |t|
      t.string :title
      t.text :memo

      t.timestamps
    end
  end
end
```



???

DB

では、migrationから実際にDBを作るには
どうすればいいでしょう？

▲図 5.6: migration から DB を作るには？

DB 設計書 (migration) から DB テーブルを作るのが `rails db:migrate` コマンドです。
`rails db:migrate` コマンドを実行すると、`/db/migrate` フォルダにあるマイグレーションファイルを実行して DB テーブルを作ります。

```
$ rails g scaffold book title:string memo:text
```



ファイル生成

```
db/migrate/20191008234834_create_books.rb
```

```
class CreateBooks < ActiveRecord::Migration[6.0]
  def change
    create_table :books do |t|
      t.string :title
      t.text :memo

      t.timestamps
    end
  end
end
```



```
$ rails db:migrate
```

migrationファイル(DB設計図)からDBを作る

DB

booksテーブル

title, memo, created_at, updated_at

▲図 5.7: rails db migrate コマンドが migration から DB を作る

5.9 保存したあとの処理

それでは、コントローラでの処理の話に戻りましょう。`@book.save` に関する動作を見て行きます。次のプログラム「2. 本のデータを保存する」の部分です。

```
app/controllers/books_controller.rb
```

```
@book = Book.new(book_params) # 1. リクエストのパラメータを使って本のデータを作る
respond_to do |format|
```

```
if @book.save # 2. 本のデータを保存する
  # 3a. 成功したらshow画面へ
  format.html { redirect_to @book, notice: 'Book was successfully created.' }
  format.json { render :show, status: :created, location: @book }
else
  # 3b. 失敗したらnew画面へ（前の画面）
  format.html { render :new }
  format.json { render json: @book.errors, status: :unprocessable_entity }
end
end
```

長いので、着目するところだけ残して短くしましょう。まず、`respond_to do |format|`はリクエストされたレスポンスの形式によって分岐させる文です。ブラウザで new 画面に内容を入力してリクエストを飛ばした場合は `format.html` が選ばれます。もう一方の `json` はデータ形式の1つで、たとえばスマートフォンアプリなど、ブラウザ以外で表示させるときによく利用されます。`format` に関する処理を除くと次のようになります。

```
@book = Book.new(book_params)
if @book.save
  redirect_to @book, notice: 'Book was successfully created.'
else
  render :new
end
```

`@book.save` は成功すると `true`、失敗すると `false` を返します。 `@book.save` の前に書いてある `if` は分岐させる命令です。 `if` のあとに書かれた処理（ここでは `@book.save`）が `true` のときはその後ろから `else` の前までを、`false` のときは `else` 以降 `end` までを実行します。

`@book.save` に成功すると `true` を返すので、`redirect_to @book, notice: 'Book was successfully created.'` が実行されます。 `redirect_to` はリダイレクト（新たにリクエストを発行して画面遷移させる）指示で、ここでは `show` アクションへのリクエストが発生します。保存した本の詳細ページ（`BooksController` の `show` アクション）を表示します。後ろの `notice` 部は画面に表示させる文を設定しています。

Book was successfully created.

Title: RubyとRailsの学習ガイド

Memo: Rails関連技術地図とそれらの学習資料の紹介

Edit | Back

▲図 5.8: リダイレクト後の show アクション画面と notice の表示

`@book.save` に失敗すると `false` を返すので、`if` の `else` 節、`render :new` が実行されます。`render` はコントローラの次の処理であるビューを指定します。ここでは `app/views/books/new.html.erb` がビューとして使われ、新規入力画面、つまり直前で入力していたページが表示されます。

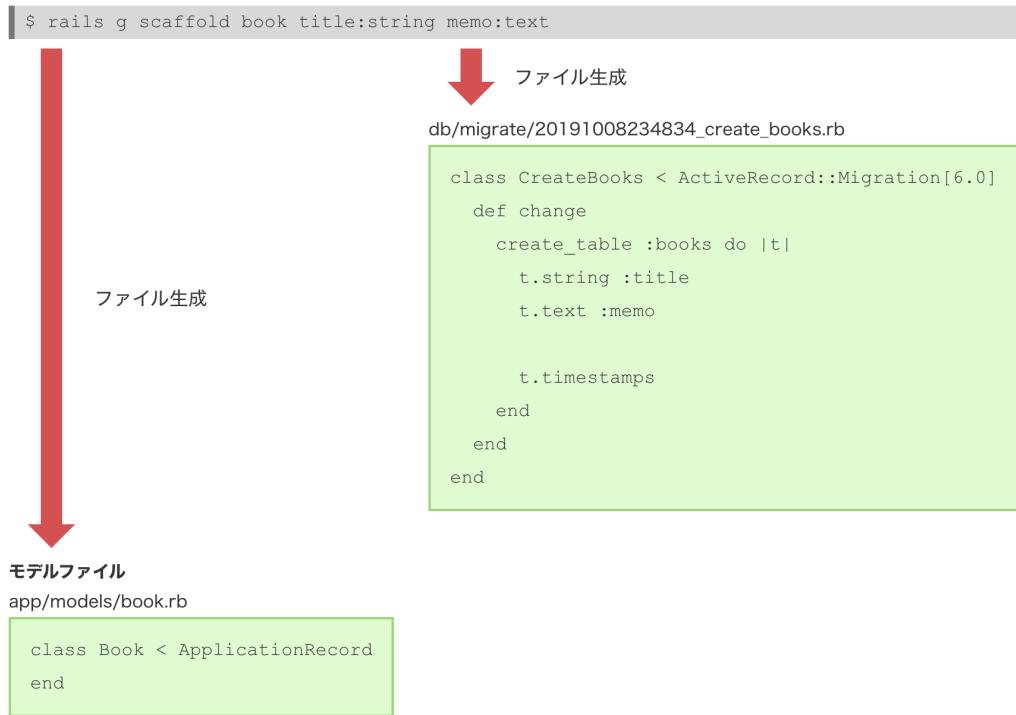
ところで、`save` メソッドが失敗するのはどんなときでしょうか？DBへの接続エラーなどのアクシデントで失敗することもありますが、モデルにある検証機能「バリデーション」を使って、想定外の入力に対して保存を失敗させることもあります。たとえば、数字を期待している郵便番号入力欄に、数字以外の文字が入力されるようなケースは、バリデーションを実装することで入力画面へ戻しユーザーに再入力を促すことができます。バリデーションについてはここでは説明しませんが、便利な機能なので Rails ガイド: Active Record バリデーションなどの説明を参考にしてみてください。

5.10 まとめ

scaffold で作られる model、migration

では、`scaffold` で作られる `model`、`migration` をまとめます。

`scaffold` コマンドを実行すると、ファイルがつくられます。



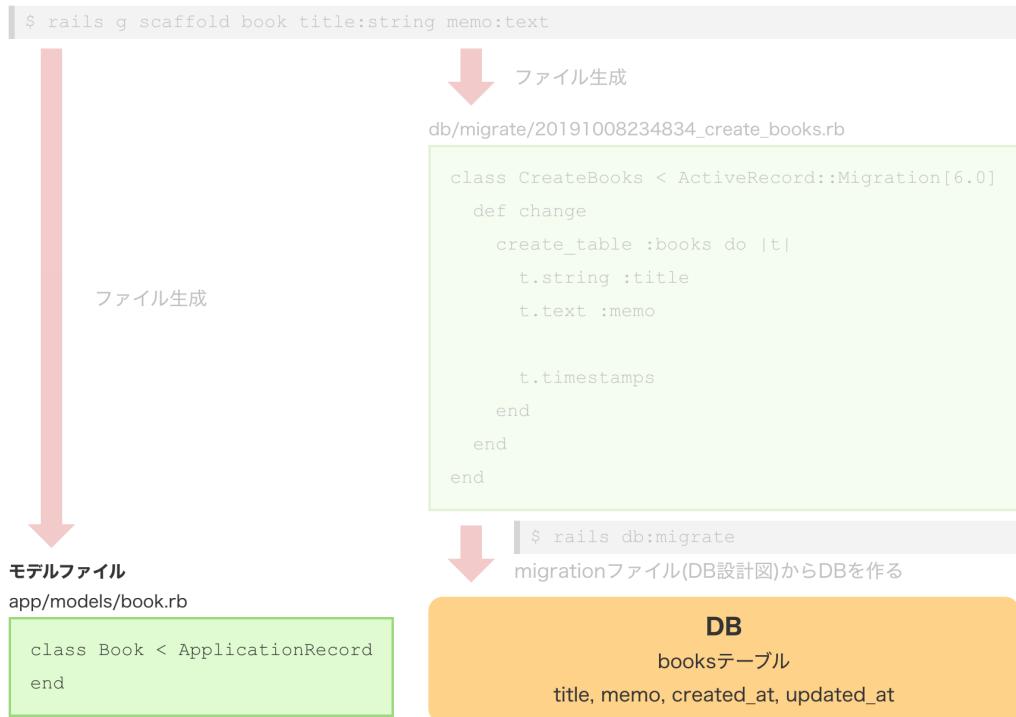
▲図 5.9: `scaffold` コマンドがファイルを作る

つづいて `rails db:migrate` コマンドを実行すると、DB がつくられます。



▲図 5.10: `rails db migrate` コマンドが DB を作る

できたモデルと DB をつかってアプリが動きます。



▲図 5.11: モデルと DB でアプリは動く

モデルの使い方

```
book = Book.new(title: "RubyとRailsの学習ガイド",
               memo: "Rails関連技術地図とそれらの学習資料の紹介")
```

Book.new で Book モデルオブジェクトを作ります。モデル名は大文字はじまりで、単数形にするルールがあります。引数で title、memo といった各カラムのデータを渡せます。

```
book.save
```

save メソッドを呼ぶと保存できます。成功すると true を、失敗すると false を返します。

```
books = Book.all.to_a
```

Book.all で DB に保存されている Book Model の全データを取得できます。Book.all.to_a を実行すると、Array に Book オブジェクトが詰まって返ってきます。

```
book = Book.where(title: "RubyとRailsの学習ガイド").first
book.title #=> "RubyとRailsの学習ガイド"
book.memo #=> "Rails関連技術地図とそれらの学習資料の紹介"
```

where メソッドを使うと検索ができます。タイトルが"Ruby と Rails の学習ガイド"である Book オブジェクトを取得します。取得した Book オブジェクトは、title メソッドでタイトルを、memo メソッドでメモを返します。

ポイントをまとめると次のようになります。

- DB はデータの保存、読み込み、検索に特化したプログラム
- モデルは DB を便利に使うための道具。DB とモデルはセットで使われる
- マイグレーション (migration) ファイルは DB を作るための設計書
- rails db:migrate はマイグレーションファイルから DB を作るコマンド

5.11 さらに学びたい場合は

モデルは多くの機能を持っています。RailsGuides にもたくさんの説明ページがあります。上手に使うことで便利な機能をかんたんに作ることができます。

- Rails ガイド: Active Record の基礎
 - モデルに関する詳しい説明です。
- Rails ガイド: Active Record マイグレーション
 - DB のマイグレーションに関する詳しい説明です。
- Rails ガイド: Active Record クエリインターフェイス
 - モデルの中で、検索に関する詳しい説明です。
- Rails ガイド: Active Record バリデーション
 - モデルには「バリデーション」と呼ばれる、値のチェック機能がついています。実際のアプリを作るときによく使う便利な機能なので、ぜひチャレンジしてみてください。
- Rails ガイド: Active Record の関連付け
 - 複数のモデルを結び付ける「関連付け」の機能は強力でアプリを作るときによく利用します。ぜひチャレンジしてみてください。

また、発展的な内容を以降に書きますので、あわせて参考にしてみてください。

5.12 既存の DB テーブルにカラムを増やすには？

既存の DB テーブルにカラムを増やすにはどうすれば良いでしょうか？前につくった migration ファイルを変更してもうまくいきません。各 migration ファイルは 1 回だけ実行される仕組みなので、過去に実行された migration ファイルを変更しても、そのファイルは実行されないからです。そこで、新しいカラムを追加するには、新しい migration ファイルをつくります。

rails g コマンドに migration を指定すると migration ファイルだけを生成できます。たとえば、books テーブルに string 型の author を加えるには以下のようにします（ファイル名中の"20191212002328"は実行するごとに異なります）。

```
rails g migration AddAuthorToBooks author:string
```

```
$ rails g migration AddAuthorToBooks author:string
  invoke  active_record
  create    db/migrate/20191212002328_add_author_to_books.rb
```

rails g migration コマンドの基本形は以下になります。

```
rails g migration Addカラム名Toテーブル名 カラム名:型名
```

作成された migration ファイルは以下のようになっています。

```
20191212002328_add_author_to_books.rb
```

```
class AddAuthorToBooks < ActiveRecord::Migration[6.0]
  def change
    add_column :books, :author, :string
  end
end
```

生成された migration ファイルには、books テーブルへ author カラムを string 型で追加する指示が書かれています。

migration ファイルを作成したら、`rails db:migrate` コマンドで DB へ内容を反映します。

```
$ rails db:migrate
```

```
== 20191212002328 AddAuthorToBooks: migrating =====
-- add_column(:books, :author, :string)
 -> 0.0031s
== 20191212002328 AddAuthorToBooks: migrated (0.0033s) =====
```

これで、DB の books テーブルへ author カラムが追加されました。

ここまで作業をまとめると、以下のようになります。

```
$ rails g migration AddAuthorToBooks author:string
```

↓ migrationファイル生成

```
db/migrate/20191212002328_add_author_to_books.rb
```

```
class AddAuthorToBooks < ActiveRecord::Migration[6.0]
  def change
    add_column :books, :author, :string
  end
end
```

生成されたmigrationファイルには booksテーブルへauthorをstring型で追加する指示が書かれている

↓ \$ rails db:migrate
migrationからDBを作る

DB
booksテーブル
title, memo, author, created_at, updated_at

▲図 5.12: books テーブルに string 型の author を加える migration

ちなみに、Rails の generate 機能は開発をアシストする機能なので、使わないで 0 から手でコードを書いても同様に動きます。つまり、rails g を使ってファイルを自動生成してから変更しても、0 から手でコードを書いても、どちらも同じ結果になります。ほかの rails g コマンド、scaffold や controller なども全て同様です。

5.13 新しいモデルと migration を一緒に作るには？

rails g コマンドに model を指定すると model と migration を生成できます。

```
rails g model book title:string memo:text
```

```
db/migrate/20191217005616_create_books.rb
app/models/book.rb
```

5.14 rails g コマンドまとめ

rails g コマンドの一覧をまとめます。

rails g migration: migration

```
$ rails g migration AddAuthorToBooks author:string
```

rails g model: model + migration

```
$ rails g model book title:string memo:text
```

rails g controller: routes + controller + view

```
$ rails g controller books index
```

rails g scaffold: model + migration + routes + controller + view

```
$ rails g scaffold book title:string memo:text
```

5.15 scaffoldでつくった既存テーブルへカラムを追加するには？

すでにある books テーブルに string 型の author を加えて、ブラウザから入力できるようにしてみましょう。

books テーブルに string 型の author を加える

```
$ rails g migration AddAuthorToBooks author:string
```

```
$ rails g migration AddAuthorToBooks author:string
      invoke  active_record
      create    db/migrate/20191217005831_add_author_to_books.rb
```

作成されたファイルは以下のようになっています。

```
db/migrate/20191217005831_add_author_to_books.rb
```

```
class AddAuthorToBooks < ActiveRecord::Migration[6.0]
  def change
    add_column :books, :author, :string
  end
end
```

migration から DB を作る

```
$ rails db:migrate
```

```
== 20191217005831 AddAuthorToBooks: migrating =====
-- add_column(:books, :author, :string)
 -> 0.003s
== 20191217005831 AddAuthorToBooks: migrated (0.0033s) =====
```

view を修正

フォームパーシャル

app/views/books/_form.html.erb

```
<%= form_with(model: book, local: true) do |form| %>
...
<div class="field">
  <%= form.label :title %>
  <%= form.text_field :title %>
</div>

<div class="field">
  <%= form.label :memo %>
  <%= form.text_area :memo %>
</div>

+ <div class="field">
+   <%= form.label :author %>
+   <%= form.text_field :author %>
+ </div>

<div class="actions">
  <%= f.submit %>
</div>
<% end %>
```

詳細表示画面

app/views/books/show.html.erb

```
<p id="notice"><%= notice %></p>

<p>
  <strong>Title:</strong>
  <%= @book.title %>
</p>

<p>
  <strong>Memo:</strong>
  <%= @book.memo %>
</p>

+<p>
+  <strong>Author:</strong>
+  <%= @book.author %>
+</p>

<%= link_to 'Edit', edit_book_path(@book) %> |
<%= link_to 'Back', books_path %>
```

一覧表示画面

app/views/books/index.html.erb

```
<p id="notice"><%= notice %></p>

<h1>Books</h1>

<table>
  <thead>
    <tr>
      <th>Title</th>
      <th>Memo</th>
+     <th>Author</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @books.each do |book| %>
    <tr>
      <td><%= book.title %></td>
      <td><%= book.memo %></td>
+     <td><%= book.author %></td>
      <td><%= link_to 'Show', book %></td>
      <td><%= link_to 'Edit', edit_book_path(book) %></td>
      <td><%= link_to 'Destroy', book, method: :delete, data: { confirm: 'Are you sure?' } %></td>
    </tr>
    <% end %>
  </tbody>
</table>

<br>

<%= link_to 'New Book', new_book_path %>
```

controller を修正

StrongParameters に author を追加します。

app/controllers/books_controller.rb

```
class BooksController < ApplicationController
...
  # Never trust parameters from the scary internet, only allow the white list through.
  def book_params
-   params.require(:book).permit(:title, :memo)
+   params.require(:book).permit(:title, :memo, :author)
  end
end
```

動作確認

rails server を起動して動作を確認してみましょう。

New Book

Title

RubyとRailsの学習ガイ

Memo

Rails関連技術地図とそれ
らの学習資料の紹介 //

Author

五十嵐邦明

Create Book

Back

▲図 5.13: 新規入力画面

new、show、index の各画面に Author 欄が追加されて、登録できるようになりました。

第6章 Gem ライブライ

この章では Gem について説明します。Gemfile や bundle コマンドについても説明をしていきます。

6.1 Gem ライブライ

プログラミングの世界では便利なプログラムがたくさん公開されています。いろいろなプログラムで共通して使える便利な公開されたプログラムをライブラリと呼びます。

Ruby の世界にもライブラリがあり、Gem と呼ばれます。Gem は rubygems.org で公開されていて、10 万を越える Gem が登録されています。Rails は Web アプリをかんたんに作る Gem の集合体として設計されています。

6.2 Gem をインストールして利用する

Gem をインストールするには gem i コマンドを利用します。gem i の i は install の省略形です。省略せずに gem install とすることもできます。例として、p メソッドをより見やすい形で拡張させた ap コマンドを提供する awesome_print という Gem をインストールしてみましょう。

```
$ gem i awesome_print
```

```
Fetching awesome_print-1.8.0.gem
Successfully installed awesome_print-1.8.0
Building YARD (yri) index for awesome_print-1.8.0...
Done installing documentation for awesome_print after 0 seconds
1 gem installed
```

コマンドを実行すると、上記のような表示が出て Gem を使えるようになります。Gem 名の後ろにハイフンで続く数字はバージョンです。実行したときの最新バージョンがインストールされるため、数字は異なることもあります。

インストールした awesome_print は、きれいに表示する ap というメソッドを提供します。irb を起動して試しに使ってみましょう。irb は対話形式で Ruby プログラムを実行できる仕組みです。rails console も irb を利用しています。

```
$ irb
```

```
require "awesome_print"
ap [1,2,3]
```

```
$ irb
> require "awesome_print"
=> true
> ap [1,2,3]
[
  [0] 1,
  [1] 2,
  [2] 3
]
=> nil
```

このような形式で配列 [1,2,3] が表示されれば成功です。Ruby が標準で持っている p メソッドも同等のことができますが、ap メソッドの方がきれいに読み易い形で表示してくれます。

プログラムで最初に実行した `require "awesome_print"` は `ap` を使えるようにするためのコードです。Gem ライブリは、`gem install` したあと、`require` を実行することで利用できます。使い方は Gem ごとに異なるため、Gem 名で検索してドキュメントを読んでみてください。GitHub のページが用意されていることが多いです。

6.3 Bundler と Gemfile

Gem は前述のように、`gem install` コマンドで簡単にインストールすることが可能ですが、この方法でたくさんの Gem をインストールしようとすると、Gem の数だけコマンドを打たなくてはいけません。それは大変なので、Gem の管理をかんたんにする Bundler という仕組みが用意されています。

Bundler で Gem 群をインストールするには 2 つの手順を実行します。

1 つ目は `Gemfile` という名前のファイルに使用する Gem を書くこと。2 つ目は `bundle install` コマンドを実行することです。仕組みの名前は Bundler ですが、コマンド名は `bundle` と最後に `r` が付かない点に注意です。

Rails アプリは最初から Bundler の仕組みを利用するようになっています。`Gemfile` は Rails アプリの場合、Rails のルートフォルダに置いてあります。エディタで開いてみましょう。

TODO: 最新 Rails バージョンで更新

```
source 'https://rubygems.org'
git_source(:github) { |repo| "https://github.com/#{repo}.git" }

#ruby '2.7.0'

# Bundle edge Rails instead: gem 'rails', github: 'rails/rails'
gem 'rails', '~> 6.0.2', '>= 6.0.2.1'
# Use sqlite3 as the database for Active Record
gem 'sqlite3', '~> 1.4'
...
```

たくさんの gem が表示されました。Rails アプリは最初につくった状態ですでに 10 個以上の

gem が登録されています。ここに、さきほどの awesome_print gem を追加してみましょう。書く場所はどこでも良いのですが、一番最後の行へ追加することにします。Gemfile へ次の一行を追加して保存します。

```
gem 'awesome_print'
```

追加して保存したら、以下のように bundle install コマンドを実行します。または、install を省略して bundle とすることもできます。このコマンドの実行時にはネットワークへ接続が必要なため、少し時間がかかります。また、各 Gem のバージョンや、Bundle complete!につづけて表示される数字は異なることがあります。

TODO: 最新 Rails バージョンで更新

```
$ bundle
Using rake 13.0.1
...
Using awesome_print 1.8.0
...
Bundle complete! 18 Gemfile dependencies, 76 gems now installed.
Use `bundle info [gemname]` to see where a bundled gem is installed.
```

これで awesome_print gem を使う準備は完了です。試しに rails console を起動して、ap メソッドを使ってみましょう。

```
$ rails c
> ap [1,2,3]
[
  [0] 1,
  [1] 2,
  [2] 3
]
=> nil
> ap Book.first
(0.3ms)  SELECT sqlite_version(*)
Book Load (0.1ms)  SELECT "books".* FROM "books" ORDER BY "books"."id" ASC LIMIT ?  [["LIMIT", 1]]
#<Book:0x00007fb66370fe10> {
  :id => 1,
  :title => "RubyとRailsの学習ガイド",
  :memo => "Rails関連技術地図とそれらの学習資料の紹介",
  :created_at => Sun, 20 Oct 2019 00:32:02 UTC +00:00,
  :updated_at => Sun, 20 Oct 2019 00:32:02 UTC +00:00
}
```

ターミナルに改行されて色がついて読み易く表示されているかと思います。また Rails アプリでは require を書く必要がありません。ちなみに、書いても問題ありません。require は複数回実行しても問題ないためです。

公開されている Ruby で書かれたプログラムに Gemfile が添えられていたときには bundle

install コマンドを実行してからプログラムを実行してみましょう。多くの場合、ドキュメントにその旨が書かれていますが、暗黙の了解で省略されていることもあります。

また、Gemfile を変更して bundle コマンドを実行すると、Gemfile.lock というファイルが作成されます。Gemfile.lock には、使われている Gem 名とそのバージョン情報が記録されています。Gemfile.lock は自動で作られるものなので、編集する必要はありません。ソースコードを保管するときは、Gemfile と Gemfile.lock の両方のファイルを保管してください。

2つのファイルの違いを例え話で説明すると、Gemfile は Gem をインストールするための発注書です。Gemfile に使いたい Gem 名を書いて、bundle install コマンドを実行すると、発注書に従って Gem がインストールされます。Gemfile.lock は納品書です。発注書にもとづいて実際にインストールされた Gem とそのバージョン情報などが書かれています。

6.4 Gemfile に書かれた Gem のバージョンアップ

各 Gem は隨時、新しいバージョンがリリースされます。Gemfile に書かれた Gem の新しいバージョンをインストールしたい場合は bundle update コマンドを使います。実行すると、新しいバージョンの Gem があればインストールして、Gemfile.lock ファイルを更新します。bundle update コマンドを実行すると、Gemfile 中のすべての Gem がバージョンアップ対象となります。

特定の Gem だけをバージョンアップしたい場合は、bundle update Gem 名 と Gem 名をつけて実行すれば OK です。指定した Gem が利用している他の Gem がある場合は、セットでバージョンアップされます。

6.5 Gemfile を使って実行する

bundle update の結果、新しいバージョンの Gem がインストールされた場合、古いバージョンの Gem はアンインストールされないので、同じ Gem の複数のバージョンがインストールされた状態になります。通常は新しいバージョンの Gem が利用され、それで問題がないケースが多いのですが、Gemfile に書かれた（古い）バージョンの Gem を使って実行したいケースもあります。

そのような場合は、bundle exec コマンドを使うことで Gemfile, Gemfile.lock に書かれた Gem バージョンで Ruby のプログラムを実行することができます。

```
bundle exec rake -T
```

このように、実行したいコマンドの前に bundle exec と書くことで、Gemfile, Gemfile.lock に書かれたバージョンの Gem を使って実行します。bundle exec に続けて、rails コマンドのほか、ruby コマンド、rake コマンド、irb コマンドなど、Ruby に関するあらゆるコマンドを書くことができます。

ところで、ここまで rails コマンドを使ってきましたが、ここに bundle exec を書かなくてもい

いのでしょうか？ 答えは、「書く必要はない」です。rails コマンドを実行すると、bin/rails ファイルが実行され、その中で bundle exec 相当の処理を行っています。

6.6 Gemfile でのバージョン指定

利用するバージョンを指定したい場合は、Gemfile にバージョンを追記します。Rails アプリの Gemfile を見ると、いくつかバージョンがすでに指定してあるものがあります。

```
gem 'bootsnap', '>= 1.4.2'
```

これは「bootsnap Gem のバージョンは、1.4.2 以上」という意味の指定になります。これは数式の意味と同じですね。一方、こんな見慣れないバージョン指定表記があります。

```
gem 'sqlite3', '~> 1.4'
```

`~>` という記号、このケースでは、`>= 1.4.0` かつ `< 1.5` という意味になります。小さいバージョンアップは受け入れて、大きなバージョンアップは受け入れない、という記号です。

バージョンの表記の仕様は Bundler のページ に解説されています。

6.7 まとめ

ポイントをまとめます。

- いろんなプログラムから使える便利な公開されたプログラムをライブラリと呼ぶ
- Gem は Ruby の世界のライブラリ
- Gem をインストールするには gem i コマンド（gem install の省略形）を使う
- Bundler は複数の Gem をかんたんに管理する仕組み
- Bundler では Gemfile という名前のファイルに使用する Gem を書く
- Gemfile を作成し bundle コマンド（bundle install の省略形）を実行すると、Gem 群がインストールされる
- bundle コマンドを実行すると Gemfile に加えて Gemfile.lock ファイルが生成されるので、両方をソース管理対象にする
- Gemfile は発注書、Gemfile.lock は納品書に相当する

第7章 画像アップロード機能の追加

アプリに画像アップロード機能を追加します。画像情報を格納するための DB カラムを追加し、carrierwave gem を利用して画像アップロード機能を実装します。

この章では「CRUD の基礎と index アクション」でつくった books_app を引き続き題材に使っていきます。モデルの章の後半で行った books テーブルに author カラムを追加した後の状態を想定していますが、author カラムの追加作業は行わなくてもこの章の内容は実行可能です。

7.1 画像情報を格納するための DB カラムを追加

最初に、既存の books テーブルに string 型の picture カラムを増やします。モデルの章でやったように、前に作った migration ファイルを編集するのではなく、新しい migration ファイルをつくります。rails g migration コマンドを実行し、migration ファイルを作成します。rails g migration コマンドの基本形を再掲します。

```
rails g migration Addカラム名Toテーブル名 カラム名:型名
```

今回は books テーブルに picture カラムを string 型で追加するので、実行するコマンドは以下になります。ターミナルで実行してみましょう。

```
rails g migration AddPictureToBooks picture:string
```

```
$ rails g migration AddPictureToBooks picture:string
Running via Spring preloader in process 3249
  invoke  active_record
  create    db/migrate/20191020225655_add_picture_to_books.rb
```

作成された migration ファイルは以下のようになっています。

```
db/migrate/20191020225655_add_picture_to_books.rb
```

```
class AddPictureToBooks < ActiveRecord::Migration[6.0]
  def change
    add_column :books, :picture, :string
  end
end
```

生成された migration ファイルには、books テーブルへ author カラムを string 型で追加する指示が書かれています。

migration ファイルを作成したら、`rails db:migrate` コマンドで DB へ内容を反映します。

```
$ rails db:migrate
== 20191020225655 AddPictureToBooks: migrating =====
-- add_column(:books, :picture, :string)
 -> 0.0027s
== 20191020225655 AddPictureToBooks: migrated (0.0028s) =====
```

7.2 carrierwave gem を追加

次は画像 upload 機能を持つライブラリ carrierwave gem を追加します。gem を追加する場合は Gemfile へ追記します。記述する場所はどこでも良いのですが、今回は一番最後の行へ追記することにします。Gemfile へ次の一行を追加して保存します。

```
gem 'carrierwave'
```

Gemfile の内容で gem を利用できるように bundle コマンドをターミナルで実行します（メッセージ中"Installing carrierwave 1.0.0"の数字は異なる場合があります）。

```
bundle
```

```
$ bundle
Using rake 13.0.0
...
Installing carrierwave 1.3.1
Bundle complete! 19 Gemfile dependencies, 79 gems now installed.
Use `bundle info [gemname]` to see where a bundled gem is installed.
```

bundle コマンドを実行すると、Gemfile に書かれた gem がまだなければインストールし利用可能になります。また、Gemfile.lock に利用するバージョンが書き込まれます。

続いて、carrierwave を利用可能にするために、carrierwave が提供する `rails g uploader Picture` コマンドを実行して必要なファイルを作成します。その前に、`bin/spring stop` コマンドを実行し、spring というキャッシュの仕組みを再起動しておきます。このコマンドは環境によって実行不要な場合もありますが、確実に成功するように実行しています。

```
$ bin/spring stop
$ rails g uploader Picture
```

```
$ bin/spring stop
Spring stopped.

$ rails g uploader Picture
Running via Spring preloader in process 4097
      create  app/uploaders/picture_uploader.rb
```

7.3 モデルの変更

次にモデルから carrierwave を利用して画像を扱えるように編集します。
app/models/book.rbを開いて、次の行、

```
class Book < ApplicationRecord
```

の直後に、以下を追加します。

```
mount_uploader :picture, PictureUploader
```

7.4 コントローラの変更

コントローラの Strong Parameter の部分に picture も追加します。
app/controllers/books_controller.rbを開いて、以下を変更します。

```
class BooksController < ApplicationController
...
  def book_params
-    params.require(:book).permit(:title, :memo, :author)
+    params.require(:book).permit(:title, :memo, :author, :picture)
  end
end
```

7.5 ビューの修正

続いて、ビューファイルを変更していきます。フォームのパーシャルファイルを変更し、picture の欄を追加します。text_fieldではなく、file_fieldを使っていることに注意してください。
app/views/books/_form.html.erb

```
<%= form_with(model: book, local: true) do |form| %>
...
<div class="field">
  <%= form.label :title %>
  <%= form.text_field :title %>
</div>

<div class="field">
  <%= form.label :memo %>
  <%= form.text_area :memo %>
</div>
```

```
<div class="field">
  <%= form.label :author %>
  <%= form.text_field :author %>
</div>

+ <div class="field">
+   <%= form.label :picture %>
+   <%= form.file_field :picture %>
+ </div>

<div class="actions">
  <%= form.submit %>
</div>
<% end %>
```

次に詳細表示画面を修正します。

app/views/books/show.html.erb

```
<p id="notice"><%= notice %></p>

<p>
  <strong>Title:</strong>
  <%= @book.title %>
</p>

<p>
  <strong>Memo:</strong>
  <%= @book.memo %>
</p>

<p>
  <strong>Author:</strong>
  <%= @book.author %>
</p>

+<p>
+  <strong>Picture:</strong>
+  <%= image_tag(@book.picture_url) if @book.picture.present? %>
+</p>

<%= link_to 'Edit', edit_book_path(@book) %> |
<%= link_to 'Back', books_path %>
```

一覧表示画面も変更します。一覧表示画面では画像は表示させず、ファイル名だけを表示することにします。

app/views/books/index.html.erb

```
<p id="notice"><%= notice %></p>

<h1>Books</h1>
```

```
<table>
  <thead>
    <tr>
      <th>Title</th>
      <th>Memo</th>
      <th>Author</th>
+     <th>Picture</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @books.each do |book| %>
    <tr>
      <td><%= book.title %></td>
      <td><%= book.memo %></td>
      <td><%= book.author %></td>
+     <td><%= book.picture %></td>
      <td><%= link_to 'Show', book %></td>
      <td><%= link_to 'Edit', edit_book_path(book) %></td>
      <td><%= link_to 'Destroy', book, method: :delete, data: { confirm: 'Are you sure?' } %></td>
    </tr>
    <% end %>
  </tbody>
</table>

<br>

<%= link_to 'New Book', new_book_path %>
```

これで画像アップロード機能が追加されました。

7.6 動作確認

rails server を起動して、ブラウザから `http://localhost:3000/books` ヘアクセスしてみましょう^{*1}。

```
rails s
```

```
$ rails s
=> Booting Puma
```

^{*1} PictureUploader が見つからない旨のエラー ("Unable to autoload constant PictureUploader"など) が発生した場合は、rails server を一度止め、`bin/spring stop` コマンドを実行してから rails server をもう一度起動して、再アクセスしてみてください。また、`app/uploaders/picture_uploader.rb` ファイルが存在するかも確認してみてください。存在しない場合は `rails g uploader Picture` コマンドが実行されていないケースが考えられます。

```
... (略)
* Listening on tcp://localhost:3000
Use Ctrl-C to stop
```

New Book リンクをクリックすると、「ファイルを選択」ボタンが増えているかと思います。ボタンを押して画像ファイルを選び、アップロードしてみましょう。

New Book

Title

RubyとRailsの学習ガイ

Memo

Rails関連技術地図とそれ
らの学習資料の紹介 //

Author

五十嵐邦明

Picture

ファイルを選択 rrsg_small.png

Create Book

Back

▲図 7.1: new

Book was successfully created.

Title: RubyとRailsの学習ガイド

Memo: Rails関連技術地図とそれらの学習資料の紹介

Author: 五十嵐邦明



Picture:

[Edit](#) | [Back](#)

▲図 7.2: show

upload した画像ファイルがブラウザに表示されているのを確認してみてください。

7.7 まとめ

ポイントをまとめます。

- carrierwave gem を使うと画像アップロード機能を追加できる
- Gemfile に新しい gem を追加した後、bundle コマンドでインストールする
- マイグレーションファイルの生成は rails g migration Add カラム名 To テーブル名 カラム名:型名

第8章 あとがき

8.1 さらに学びたい方への資料

Git と GitHub でソースコードを管理する

Git はソースコードを管理するシステムであり、遠隔地にあるリポジトリ（ソースコード管理庫）と通信するコマンド群も提供しています。Git を使うと、プログラムを昔のある時点まで戻ったり、他の人が書いたプログラムを自分の手元に取得することができます。GitHub は Git を使ったソースコード管理を提供するほか、コードを公開したり、コードで対話する仕組みである Pull Request といった機能を提供する開発プラットフォームです。資料としては、『RailsGirls ガイド GitHub に自分のアプリを Push する』などが参考になります。

- RailsGirls ガイド GitHub に自分のアプリを Push する
 - <https://railsgirls.jp/github>

つくったアプリを公開する

ここまで作ったアプリを Web サービスとして公開するには、PaaS (Platform as a Service) と呼ばれるサービスを使うとかんたんに公開できます。アプリケーションのソースコードを転送するだけで Web サービスを公開できるので、AWS などのサーバを借りるタイプのものと比べて手軽に使うことができます。Rails アプリでは Heroku と呼ばれるサービスがよく使われています。資料としては、『RailsGirls ガイド Heroku に Rails アプリをアップ』のページが最新の情報へとメンテナンスされていることが多くお勧めです。

- RailsGirls ガイド Heroku に Rails アプリをアップ
 - <https://railsgirls.jp/heroku>

学び方の資料

- Ruby と Rails の学習ガイド 2019 技術書典 6 拡大版
 - 五十嵐邦明 著 <https://igaigarb.booth.pm/>

Rails を学びはじめると、Web サービスにはたくさんの技術が使われていることに気づきます。いま学んでいるこの技術は全体の中でどんな位置づけなんだろう？ そして次は何を学べばいいのだろう？ この本は Ruby そして Rails を学ぶ旅のガイドブックです。技術を学び、戻ってくるたびにレベルアップする、Web サービスを作る技術者の冒険の旅をサポートする地図です。各技術を学ぶための資料や書籍の情報を書いています。

Rails を学ぶ資料

本書では Rails の基礎の仕組みを説明し、簡単なアプリを作って、アプリを作る上で基本的な

デバッグ作業を説明してきました。

さらに Rails の理解を深めたい方には、以下の資料や書籍をお薦めします。

- Ruby on Rails ガイド
 - <https://railsguides.jp>
 - 電子書籍版

Rails の各機能を網羅的に解説しているサイトです。Rails の機能の使い方が分からぬときには最初にあたるサイトでもあります。調べ物で使うのも良いですが、一度通読しておくとどのような機能があるかを把握できてその後の進みが速いです。常に最新の Rails についての記述が書いてあるサイトでもあります。本書でも各章の最後に、その章の内容に該当する Rails ガイドへのリンクを書いているのでぜひ読んでみてください。

- Ruby on Rails チュートリアル
 - <https://railstutorial.jp>
 - 電子書籍版

Rails アプリを作りながら学ぶ Web 上の資料です。Rails を学ぶ初期の資料として定番のものです。実際にアプリを作りながら知識を深めたいという方へお勧めです。

- 現場で使える Ruby on Rails 5 速習実践ガイド
 - 株式会社万葉 監修 マイナビ出版 ISBN: 978-4839962227

Rails の基礎知識を広く学べる、通称『現場 Rails』と呼ばれる書籍です。Rails の基礎を網羅的に説明しています。Rails の受託開発や開発支援を長年つづけている株式会社万葉のみなさんが執筆されているので、タイトル通り現場で使える技術に即した内容になっています。Rails5.2 対応。

- Ruby on Rails 6 実践ガイド
 - 黒田努 著 インプレス ISBN: 978-4295008057

Rails の入門書籍です。サンプルアプリつくりを通して Rails の各機能を説明していきます。必要な知識を過不足なく順序よく並べて説明しています。最初の版から良い内容でしたが、積み重ねた改訂によりさらに磨きがかかっています。Rails6.0 対応。

- Ruby on Rails 6 エンジニア養成読本
 - すがわらまさのり、前島真一 著 技術評論社 ISBN: 978-4297108694

Rails6.0 時代の Rails 新機能をはじめ、Rails の各機能の使い方とその解説を幅広く説明している書籍です。実践的な内容がたくさん書かれていますが、解説がどれもとても分かりやすく、早く理解できるなど感じました。Rails はバージョンアップに伴って知識も更新していく必要がありますが、この本は新しい知識を得られると共に、Rails 変遷の歴史も学べます。本書からのステップアップにとても良い 1 冊です。Rails6.0 対応。

- パーフェクト Ruby on Rails
 - すがわらまさのり、前島真一、近藤宇智朗、橋立友宏 著 技術評論社 ISBN: 978-4774165165

Rails の基礎から応用まで網羅した書籍です。実際に Rails でプロダクトを作っている著者陣がそのノウハウを踏まえて体系的に Rails を解説しています。サンプルアプリを手を動かして作ることができるのも良いところです。初級者から中級者へのステップアップに。名著です。Rails4.1 対応。

Ruby を学ぶ資料

Rails は Ruby というプログラミング言語で作られています。Ruby の知識を学ぶことで、内容をより深く理解し、自分が書く内容を豊かにできます。

- ゼロからわかる Ruby 超入門
 - 五十嵐邦明、松岡浩平 著 技術評論社 ISBN: 978-4297101237

私と松岡さんで書き、べこさんがイラストを担当した書籍です。プログラミングが初めての方へ向けた、Ruby プログラミングの入門書です。プログラミングの基礎である条件分岐、繰り返しの概念を図解を入れて丁寧に説明します。そして Ruby でよく使われる整数、文字列、配列、ハッシュと言ったオブジェクトを説明しています。また、エラー時の対処方法、リファレンスマニュアルの調べ方、デバッグの基礎も学べるので、自分で調べる力をつけることができます。Rails で必要となる知識を学ぶことをゴールにしているので、メソッドやクラス、モジュールの作り方、使い方など、最低限必要となる知識を効率良く学べます。また、Gem とそれを束ねる Bundler の使い方を学べます。Ruby2.5 対応。

- たのしい Ruby 第6版
 - 高橋征義、後藤裕蔵 著 SBクリエイティブ ISBN: 978-4797399844

Ruby プログラミングに必要な知識を幅広く説明していて、この本でカバーしていない Ruby の知識を探すのが難しいほどです。15 年以上に渡って改訂されて読み継がれている名著です。Ruby2.6 対応。

- まんがでわかる Ruby
 - youchan、湊川あい 著 <https://booth.pm/ja/items/1306534>

お絵かきプログラムを題材に、Ruby の基礎をマンガと分かりやすい文章で解説しています。短いプログラムを書くだけで、なぜだかさまざまな図形が描かれる。たのしくプログラミングしていたら、基礎知識がいつのまにか身についていた、そんな本です。Ruby の特徴でもある「たのしさ」を最大限に引き出してくれる 1 冊です。

HTML と CSS を学ぶ資料

HTML と CSS をつかったデザインを学ぶ資料としては、サルワカの Web デザイン入門 や、少し古めのものですが ごく簡単な HTML の説明 などがあります。

- サルワカの Web デザイン入門
 - <https://saruwakakun.com/html-css/basic>
- ごく簡単な HTML の説明

- <https://www.kanzaki.com/docs/htminfo.html>

Git を学ぶ資料

Git と呼ばれるソース管理システムを使うと、プログラムの履歴を管理することができます。プログラムを昔のある時点まで遡ったり、他の人が書いたプログラムを自分の手元に取得することができます。GitHub は Git を使ったソースコード管理を提供するほか、コードで対話する仕組みである Pull Request といった機能を提供する開発プラットフォームです。

- わかばちゃんと学ぶ Git 使い方入門
 - 湊川あい 著 シーアンドアール研究所 ISBN: 978-4863542174

Git と GitHub を GUI アプリから使う方法をマンガ形式で分かりやすく説明しています。

- やりたいことが今すぐわかる 逆引き Git 入門
 - 高見龍 著 秀和システム ISBN: 978-4798059594

ターミナルからコマンドを打つ方法 (CLI とも呼びます) で Git を分かりやすく説明しています。

8.2 Ruby コミュニティ

なにより、自分で作りたいアプリを作ってみるのが一番勉強になります。作っている上で、もしもつまづき、インターネットで検索しても分からぬことがあったときには、近くの Ruby コミュニティへぜひ足を運んでみてください。新しい Rubyist (あなたのことです!) を温かく迎えてくれるはずです。Ruby コミュニティは「都市名.rb」という名前で活動していることが多いです。全国各地の Ruby コミュニティについて、地域 Ruby の会のページに情報がまとまっています。また、RubyKaigi や地域 Ruby 会議と呼ばれるカンファレンスも定期的に開催されています。

ネット上の集まりとして、ruby-jp という、Rubyist が集まる Slack ワークスペースもあります。2000 人以上が参加し、Ruby や Rails に限らず、あらゆる技術について話をしています。質問をするチャンネルもありますので、つまづいたことがあるときに聞いてみるときっと回答が返ってくると思います。技術以外にも、各地のカンファレンスやミートアップの情報について話したりするのはもちろん、つくった料理の紹介や語学学習情報の交換など、さまざまな話題を扱うチャンネルがあります。

- 地域 Ruby の会
 - <https://github.com/ruby-no-kai/official/wiki/RegionalRubyistMeetUp>
- RubyKaigi
 - <https://rubykaigi.org/>
- 地域 Ruby 会議
 - <https://regional.rubykaigi.org/>
- ruby-jp

- <https://ruby-jp.github.io/>

8.3 謝辞とメッセージ

この講義資料を最初につくったときにレビューをしてくれた濱崎健吾さんへ感謝します。講義にも TA として参加し、毎回の講義の内容についても一緒に考えててくれて、大変心強かったです、なによりましたのいい時間でした。

一橋大学の Jonathan Lewis 先生、鈴木隆一先生、ニフティ株式会社の宮内秀哲さん、講義の機会をいただき、一緒に講義を進めさせていただきありがとうございました。私にとって素晴らしい経験でした。ニフティ株式会社の寄附講座として行われたこの講座を開講してくださったニフティ株式会社様へ感謝致します。松岡浩平さん、講義を引き継いでいただき、また教科書についてブラックアップを進め、そして Ruby 超入門と一緒に作ってくれてありがとうございました。たくさんの提言で私の理解も深まり、たいへん勉強になりました。

レビューをしてくれた丸山有慧さん、境友香さん、松永久美子さん、私の気付かない視点をいろいろ教えていただき、また、急なお願いにも快く応じていただきありがとうございました。

達人出版会の高橋征義さん、山根ゆりえさん、大変お世話になりました。教科書を書き始めた2012年からの4年間を電子書籍という形でまとめられたことを嬉しく思っています。レビューの内容ももちろん、レビュー方法も勉強になりました。

また、改訂にあたり図を描いてくださったべこさんへ深く感謝します。HTML/CSS を活用して、どんどんバージョンアップしていく Rails を楽に追いかけていくように工夫してくださいました。また、わかりやすい図を目指して一緒に考えてくださいました。

執筆スポンサーとして協賛いただき、私の執筆時間をつくってくださった合同会社フィヨルド、ギフティ株式会社、YassLab 株式会社の各社様に感謝いたします。Ruby を学ぶ人たちへより多くの学習資料を届けるという趣旨に賛同いただき、深く感謝します。

素晴らしい言語とコミュニティをつくってくれたまつもとゆきひろさん、そして Rubyist のみなさんへ感謝します。matz さんのおかげで私の人生はたのしいことをたくさん経験できています。

最後に、ここまで読んでくださった皆様、本当にありがとうございました。みなさんと Ruby コミュニティのどこかでお会いできたらこれほど幸せなことはありません。どこかで見かけましたら、ぜひ気軽に声かけください。いつかお会いできる日を楽しみにしています。

2020年1月 五十嵐邦明

8.4 著者略歴

五十嵐邦明 ガーネットテック 373 株式会社

フリーランスの Ruby, Rails エンジニア。2003年、Ruby で書かれた Web 日記アプリ tDiary を使い始めて Rubyist になる。一橋大学の非常勤講師として 2012 年から 2 年間、Ruby と Rails を

教える。受託開発会社、Web サービス開発会社 CTO などを経て 2017 年 4 月よりフリーランス、2019 年ガーネットテック 373 株式会社を設立、同社代表取締役社長。RubyWorld Conference や RubyConf 台湾などで講演。島根県 Ruby 合宿、Rails 寺子屋、Rails Girls などで講師を行う。著書に『ゼロからわかる Ruby 超入門』、『Rails の教科書』、『Ruby と Rails の学習ガイド』ほか。

Rails の教科書

2020 年 2 月 29 日 ver 1.0

著 者 五十嵐邦明

印刷所 日光企画

© 2020 五十嵐邦明