
Ruby on Rails 3.2 チュートリアル

実例を使ってRailsを学ぼう

Michael Hartl (マイケル・ハートル)

目次

第1章 ゼロから本番展開まで

- 1.1 はじめに
 - 1.1.1 読者の皆さまへ
 - 1.1.2 Railsとスケールについて
 - 1.1.3 この本における取り決め
- 1.2 さっそく動作させる
 - 1.2.1 開発環境
 - IDE
 - テキストエディタとコマンドライン
 - ブラウザ
 - 使用するツールについて
 - 1.2.2 Ruby、RubyGems、Rails、Git
 - Railsインストーラ (Windows)
 - Gitのインストール
 - Rubyのインストール
 - RubyGemsのインストール
 - Railsのインストール
 - 1.2.3 最初のアプリケーション
 - 1.2.4 Bundler
 - 1.2.5 `rails server`
 - 1.2.6 Model-view-controller (MVC)
- 1.3 Gitによるバージョン管理
 - 1.3.1 インストールとセットアップ
 - 初めてのシステム・セットアップ
 - 初めてのリポジトリ・セットアップ
 - 1.3.2 追加とコミット
 - 1.3.3 Gitのメリット
 - 1.3.4 GitHub
 - 1.3.5 ブランチ (branch)、変更 (edit)、コミット (commit)、マージ (merge)
 - ブランチ (branch)
 - 変更 (edit)
 - コミット (commit)
 - マージ (merge)
 - プッシュ (push)
- 1.4 展開する
 - 1.4.1 Herokuのセットアップ
 - 1.4.2 Herokuに展開する (1)
 - 1.4.3 Herokuに展開する (2)
 - 1.4.4 Heroku コマンド
- 1.5 最後に

第2章 デモアプリケーション

- 2.1 アプリの計画
 - 2.1.1 ユーザーのモデル設計
 - 2.1.2 マイクロポストのモデル設計
- 2.2 Users リソース
 - 2.2.1 ユーザページを探検する

- 2.2.2 MVCの挙動
- 2.2.3 Users リソースの欠点
- 2.3 Mircroposts リソース
 - 2.3.1 マイクロポストのページを探検する
 - 2.3.2 マイクロポストをマイクロにする
 - 2.3.3 ユーザーとマイクロポストを`has_many`で関連づける
 - 2.3.4 繙承の階層
 - 2.3.5 デモアプリケーションの展開
- 2.4 最後に

第3章 ほぼ静的なページの作成

- 3.1 静的ページ
 - 3.1.1 「本当に」静的なページ
 - 3.1.2 Railsによる静的なページ
- 3.2 最初のテスト
 - 3.2.1 テスト駆動開発
 - 3.2.2 ページの追加
 - 赤 (Red)
 - 緑 (Green)
 - リファクタリング
- 3.3 ちょっとだけ動的ページ
 - 3.3.1 タイトル変更をテストする
 - 3.3.2 タイトルのテストをパスさせる
 - 3.3.3 組込みRuby
 - 3.3.4 レイアウトを使って重複を解消する
- 3.4 最後に
- 3.5 演習
- 3.6 高度なセットアップ
 - 3.6.1 `bundle exec`を追放する
 - RVM Bundler の統合
 - binstubオプション
 - 3.6.2 Guardによるテストの自動化
 - 3.6.3 Sporkを使ったテストの高速化
 - GuardにSporkを導入する
 - 3.6.4 Sublime Text上でテストする

第4章 Rails風味のRuby

- 4.1 動機
- 4.2 文字列(string)とメソッド
 - 4.2.1 コメント
 - 4.2.2 文字列
 - 出力
 - シングルクオート内の文字列
 - 4.2.3 オブジェクトとメッセージ受け渡し
 - 4.2.4 メソッドの定義
 - 4.2.5 `title` ヘルパー、再び
- 4.3 他のデータ構造
 - 4.3.1 配列と範囲演算子
 - 4.3.2 ブロック
 - 4.3.3 ハッシュとシンボル
 - 4.3.4 CSS、再び
- 4.4 Rubyにおけるクラス
 - 4.4.1 コンストラクタ
 - 4.4.2 クラス継承
 - 4.4.3 組込みクラスの変更
 - 4.4.4 コントローラクラス
 - 4.4.5 ユーザークラス
- 4.5 最後に
- 4.6 演習

第5章 レイアウトを作成する

- 5.1 構造を追加する
 - 5.1.1 ナビゲーション
 - 5.1.2 BootstrapとカスタムCSS
 - 5.1.3 パーシャル (partial)
- 5.2 Sass と asset pipeline
 - 5.2.1 asset pipeline
 - asset ディレクトリ
 - マニフェストファイル
 - プリプロセッサエンジン
 - プロダクション環境での効率性
 - 5.2.2 素晴らしい構文を備えたスタイルシート
 - ネスト
 - 変数
- 5.3 レイアウトのリンク
 - 5.3.1 ルートのテスト
 - 5.3.2 Railsのルート
 - 5.3.3 名前付きルート
 - 5.3.4 Pretty RSpec
- 5.4 ユーザーのサインアップ：最初のステップ
 - 5.4.1 ユーザーコントローラ
 - 5.4.2 サインアップURI
- 5.5 最後に
- 5.6 演習

第6章 ユーザーのモデルを作成する

- 6.1 Userモデル
 - 6.1.1 データベースの移行
 - 6.1.2 modelファイル
 - モデル注釈
 - アクセス可能な属性
 - 6.1.3 ユーザーオブジェクトを作成する
 - 6.1.4 ユーザーオブジェクトを検索する
 - 6.1.5 ユーザーオブジェクトを更新する
- 6.2 ユーザーを検証する
 - 6.2.1 最初のユーザーテスト
 - 6.2.2 プrezensを検証する
 - 6.2.3 長さを検証する
 - 6.2.4 フォーマットを検証する
 - 6.2.5 一意性を検証する
 - 一意性の警告
- 6.3 セキュアなパスワードを追加する
 - 6.3.1 暗号化されたパスワード
 - 6.3.2 パスワードと確認
 - 6.3.3 ユーザー認証
 - 6.3.4 ユーザーがセキュアなパスワードを持っている
 - 6.3.5 ユーザーを作成する
- 6.4 最後に
- 6.5 演習

第7章 ユーザー登録

- 7.1 ユーザーを表示する
 - 7.1.1 デバッグとRails環境
 - 7.1.2 ユーザーリソース
 - 7.1.3 ファクトリーを使用してユーザー表示ページをテストする
 - 7.1.4 gravatar画像とサイドバー
- 7.2 ユーザー登録フォーム
 - 7.2.1 ユーザー登録のためのテスト
 - 7.2.2 `form_for`を使用する

- 7.2.3 フォームHTML
- 7.3 ユーザー登録失敗
 - 7.3.1 正しいフォーム
 - 7.3.2 ユーザー登録のエラーメッセージ
- 7.4 ユーザー登録成功
 - 7.4.1 登録フォームの完成
 - 7.4.2 flash
 - 7.4.3 実際のユーザー登録
 - 7.4.4 SSLを導入してプロダクション環境を展開する
- 7.5 最後に
- 7.6 演習

第8章 サインイン、サインアウト

- 8.1 セッション、サインインの失敗
 - 8.1.1 Sessionコントローラ
 - 8.1.2 サインインをテストする
 - 8.1.3 サインインのフォーム
 - 8.1.4 確認フォームを送信する
 - 8.1.5 フラッシュメッセージをレンダリングする
- 8.2、 サインイン成功
 - 8.2.1 [このアカウント設定を保存する]
 - 8.2.2 正しいsign_inメソッド
 - 8.2.3 現在のユーザー
 - 8.2.4 レイアウトリンクを変更する
 - 8.2.5 ユーザー登録と同時にサインインする
 - 8.2.6 サインアウトする
- 8.3 Cucumberの紹介(オプション)
 - 8.3.1 インストールと設定
 - 8.3.2 機能と手順
 - 8.3.3 対比: RSpecのカスタムマッチャー
- 8.4 最後に
- 8.5 演習

第9章 ユーザーの更新・表示・削除

- 9.1 ユーザーを更新する
 - 9.1.1 編集フォーム
 - 9.1.2 編集の失敗
 - 9.1.3 編集の成功
- 9.2 認証
 - 9.2.1 ユーザーのサインインを要求する
 - 9.2.2 正しいユーザーを要求する
 - 9.2.3 フレンドリーフォワーディング
- 9.3 すべてのユーザーを表示する
 - 9.3.1 ユーザーインデックス
 - 9.3.2 サンプルのユーザー
 - 9.3.3 ページネーション
 - 9.3.4 部分的リファクタリング
- 9.4 ユーザを削除する
 - 9.4.1 管理ユーザー
 - attr_accessible属性再び
 - 9.4.2 destroyアクション
- 9.5 最後に
- 9.6 演習

第10章 ユーザーのマイクロポスト

- 10.1 マイクロポストのモデル
 - 10.1.1 基本的なモデル
 - 10.1.2 Accessible属性と最初の検証
 - 10.1.3 User/Micropostの関連付け
 - 10.1.4 マイクロポストを改良する

	デフォルトのスコープ
	Dependent: destroy
10.2	10.1.5 コンテンツの検証
	マイクロポストを表示する
	10.2.1 ユーザー表示ページの拡張
	10.2.2 マイクロポストのサンプル
10.3	マイクロポストを操作する
	10.3.1 アクセス制御
	10.3.2 マイクロポストを作成する
	10.3.3 フィードの原型
	10.3.4 マイクロポストを削除する
10.4	最後に
10.5	演習

第11章 ユーザーをフォローする

11.1	Relationshipのモデル
	11.1.1 データモデルの問題(および解決策)
	11.1.2 User/relationshipの関連付け
	11.1.3 検証
	11.1.4 フォローしているユーザー
	11.1.5 フォローされているユーザー
11.2	フォローしているユーザー用のWebインターフェイス
	11.2.1 フォローしているユーザーのサンプルデータ
	11.2.2 統計とフォロー用フォーム
	11.2.3 「フォローしている」と「フォローされている」のページ
	11.2.4 [フォローする] ボタン(標準的な方法)
	11.2.5 [フォローする] ボタン(Ajax)
11.3	ステータスフィード
	11.3.1 動機と計画
	11.3.2 フィードを初めて実装する
	11.3.3 サブセレクト
	11.3.4 新しいステータスフィード
11.4	最後に
	11.4.1 サンプルアプリケーションの機能を拡張する 返信機能
	メッセージ機能
	フォロワーの通知
	パスワードリマインダー
	ユーザー登録の確認
	RSSフィード
	REST API
	検索機能
	読み物ガイド
11.5	演習

前書き

私が前にいた会社(CD Baby)は、かなり早い段階でRuby on Railsに乗り換えたのですが、またPHPに戻ってしまいました(詳細は私の名前をGoogleで検索してみて下さい)。Michael Hartl氏が書いたこの本を強く勧められ、これはやってみなければ、と試した結果、私が再びRailsに乗り換えるに至ったのがこのRuby on Railsチュートリアルです。

私は多数のRails本を参考にしてきましたが、眞の決定版と呼べるものは本書をおいて他にありません。本書ではあらゆる手順が文字通りRails流("Rails way"は railway にかけている)で行われており、最初はなかなか慣れませんでしたが、この本を終えた今、ついにこれこそが自然な方式だと感じられるまでになりました。また、本書はRails本の中で唯一、多くのプロが推奨するテスト駆動開発(TDD: Test Driven Development)方式を全編を通して

実践していく、非常に分かりやすく解説されています。極めつけは、Git、GitHub、そしてHerokuまでも実例に含めた事で、チュートリアルのコード例など実際のプロジェクトの開発プロセスまでも体験できるようになっていることです。チュートリアルのコード例は、どれもチュートリアルと見事に一体化しています。

本書は全体が筋道だったストーリーに貫かれており、非常に素晴らしいものです。私自身、3日間かけて章の終わりにある練習問題もやりながらRailsチュートリアルを一気に読破しました。最初から最後まで、途中を飛ばさずにやるのが一番効果的で有益な読み方です。ぜひやってみて下さい。

是非お試しください。

デレック・シバーズ (Derek Sivers) (sivers.org)

元: CD Baby (創始者)

現在: Thoughts Ltd. (創始者)

謝辞

Ruby on Rails チュートリアルは、私の以前の著書*RailsSpace*と、その時の共著者のAurelius Prochazkaに多くを負っています。Aureには、*RailsSpace*での協力と本書への支援も含め、感謝したいと思います。また、*RailsSpace*と*Rails* チュートリアル両方の編集を担当して頂いたDebra Williams Cauley氏にも謝意を表したく思います。

私にインスピレーションと知識を与えてくれたRubyistの方々にも感謝したいと思います:David Heinemeier Hansson、Yehuda Katz、Carl Lerche、Jeremy Kemper、Xavier Noria、Ryan Bates、Geoffrey Grosenbach、Peter Cooper、Matt Aimonetti、Gregg Pollack、Wayne E. Seguin、Amy Hoy、Dave Chelimsky、Pat Maddox、Tom Preston-Werner、Chris Wanstrath、Chad Fowler、Josh Susser、Obie Fernandez、Ian McFarland、Steven Bristol、Pratik Naik、Sarah Mei、Sarah Allen、Wolfram Arnold、Alex Chaffee、Giles Bowkett、Evan Dorn、Long Nguyen、James Lindenbaum、Adam Wiggins、Tikhon Bernstam、Ron Evans、Wyatt Greene、Miles Forrest、Pivotal Labsの方々、Herokuの方々、thoughtbotの方々、そしてGitHubチーム。最後に、ここに書ききれないほど多くの読者の方々から執筆中にバグ報告や提案を頂き、本書をできる限り良いものにするのに大変役立ちました。

著者

マイケル・ハートル (Michael Hartl) は、*Ruby on Rails*を使用したweb開発を手ほどきする教材として有名な *Ruby on Rails* チュートリアルの著者です。(今ではすっかり古くなってしましましたが) Railsのチュートリアル本である*RailsSpace*の著述と開発に携わったことがあり、一時人気を博したRuby on RailsベースのソーシャルネットワーキングプラットフォームInsoshiも開発していました。2011年、マイケルはRailsコミュニティへの貢献が認められてRuby Hero Awardを受賞しました。ハーバード大学卒業後、カリフォルニア工科大学で物理学博士号を取得し、起業プログラムY Combinatorの卒業生でもあります。

著作権とライセンス

Ruby on Rails チュートリアル: 実例を使ってRailsを学ぼうCopyright © 2010 by Michael Hartl.*Ruby on Rails* チュートリアル内のすべてのソースコードはMITライセンスおよびBeerwareライセンスの元で提供されています。

(The MIT License)

Copyright (c) 2012 Michael Hartl

Permission is hereby granted, free of charge, to any person obtaining a copy of this software and associated documentation files (the "Software"), to deal in the Software without restriction, including without limitation the rights to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

```
/ *
* -----
* "THE BEER-WARE LICENSE" (Revision 42):
* Michael Hartl wrote this code. As long as you retain this notice you
* can do whatever you want with this stuff. If we meet some day, and you think
* this stuff is worth it, you can buy me a beer in return.
* -----
*/
```

第1章 ゼロから本番展開まで

Ruby on Rails チュートリアルへようこそ。この本の目的は、「*Ruby on Rails* を使ったウェブ開発を学びたいんだけど、どこから始めればいいの?」という質問に答えることです。本書を読み終わる頃には、あなたは自作のカスタムWebアプリケーションを開発し、展開(deploy)するための知識をすべて得られるはずです。また、Railsの「教育エコシステム」なるものに含まれる、さらに高度な本、ブログ、スクリーンキャストなどの恩恵を受けるのにふさわしい実力を身につけられます。*Ruby on Rails* チュートリアルはRails 3を使用しているため、ここで学んだ内容はRailsの最新バージョンに完全に対応しています。(最新の*Ruby on Rails* チュートリアルについては、この本が掲載されている<http://railstutorial.org/>を参照してください。本書をオンラインで読んでいる方は、<http://railstutorial.org/book>でオンライン版Rails Tutorial bookで最新版を確認してください。

本書の目的は、単にRailsを教えることではありません。そうではなく、Railsを使用したWeb開発を教える、すなわちWorld Wide Web用ソフトウェアを開発できるスキルを獲得(または強化)できるようにすることが目的です。*Ruby on Rails*のみならず、HTML & CSS、データベース、バージョン管理、テスティング、展開のためのスキルセットも本書の対象となります。この目的達成のため、*Ruby on Rails* チュートリアルでは統合的なアプローチを採用しています。つまり、実際に動作するサンプルアプリケーションを一から作成するという例題をこなすことによってRailsを習得します。Siversが前書きで述べたとおり、本書は一貫した直線的なストーリーを持つように構成されており、最初から最後まで飛ばさずに読むことを前提としています。技術書を飛ばし読み/つまみ食いするのが習慣になっている方にとっては、本書のような直線的なアプローチを行なうために多少頭の切り替えが必要になるかもしれませんのが、その価値は十分にあります。ぜひトライしてみてください。*Ruby on*

*Rails*チュートリアルをTVゲームにたとえれば、あなたは主人公であり、1章ごとにRails開発者としてレベルアップして行くとお考えください(演習は中ボスです)。

この第1章では、必要なソフトウェアをインストールし、開発環境(1.2)を整えてRuby on Railsの準備をします。続いて、**first_app**という最初のRailsアプリの作成に着手します。Railsチュートリアルではソフトウェア開発のベストプラクティスを重要視しているので、新しいRailsプロジェクトを作成後、ただちにGit(1.3)を使用してバージョン管理下に置きます。そして何と、この章ではプロダクション環境(1.4)に展開して一般公開するところまで行います。

第2章では、Railsアプリケーションの基本的な仕組みのデモを兼ねて、新たなプロジェクトを作成します。手っ取り早く動かしたいので、このデモアプリ(**demo_app**)はscaffold(囲み1.1)を使用してコードを自動生成します。scaffoldで生成されたコードは複雑で美しくないで、第2章ではURI(Webブラウザで使用するURL¹と呼ばれることがあります)を通してデモアプリを使ってみます。

第3章以降では、サンプルアプリケーション(**sample_app**と呼びます)を作成します。このアプリはTDD(テスト駆動開発で開発し、第3章で静的ページ(と少しの動的コンテンツ)を追加する事から始めます。第4章では少し回り道をし、Railsを動かしているRuby言語について簡単に学びます。第5章から第9章ではレイアウト、ユーザーのデータモデル、そしてユーザー登録・認証システムを作りサンプルアプリの基礎を完成させます。最後に第10章と第11章ではマイクロブログ機能とソーシャル機能を実装し、実際に動作するサイトに仕上げます。

最終的なサンプルアプリは、元々は同じRailsで書かれていたあるソーシャルマイクロブログサイトにとてもよく似た仕上がりになります。もちろん、私たちはこのサンプルアプリを中心に開発を進めて行きますが、Railsチュートリアルでは今後Webアプリケーションを作成する時に、どのようなアプリケーションであっても通用する基礎を築きます。

囲み1.1 早く簡単なScaffoldの甘い誘惑

Railsの作者David Heinemer Hansson氏による有名な「15分で作るブログ」ビデオ(現在はRyan Bates氏による「Rails 2による15分で作るブログ」に更新されています)などを見ていただければわかるように、Railsは当初から刺激的なフレームワークです。これらのビデオはRailsのパワーを目の当たりにするには最適で、是非視聴する事をお勧めします。しかし、これらのビデオでは15分でブログを作るという離れ業を、scaffoldという機能を利用して成し遂げています。このscaffoldという機能は、Railsの**generate**コマンドによる自動生成コードに強く依存します。

Ruby on Railsのチュートリアルを書いていて、あまりにも素早く簡単にコードが生成できるscaffoldの機能を私もついあてにしてしまいそうになります。しかし、生成されたコードの複雑さとその量の多さに、初心者は圧倒されてしまうことでしょう。動くには動くのですが、ソースコードを見ても何がなんだか理解できないことでしょう。scaffoldのコード自動生成に頼ると、スクリプト作成の達人にはなるかもしれません、Rails自体の知識はほとんど身に付きません。

*Ruby on Rails*チュートリアルでは、ほぼ逆のアプローチで開発を行います。scaffoldは第2章での簡単なデモアプリ作成に使用しますが、このチュートリアルの中核は第3章から作成するサンプルアプリケーションです。開発の段階ごとに、手頃な一口サイズのコードを書いてもらいます。この一口サイズコードは、無理なく理解できる程度にシンプルで、かつある

程度の歯ごたえとやりがいを得られるように配慮してあります。最終的には、どんなWebアプリを作成する時にも役立つ、より深く柔軟なRailsの知識を身につけられるようになっています。

1.1 はじめに

2004年にデビューしてからというもの、Ruby on Railsは動的Webアプリケーションを開発するための最も有力な、そして人気のあるフレームワーク（土台）の1つに成長しました。ほんの小さな会社から巨大企業までがRailsを使っています：

37signals、GitHub、Shopify、Scribd、Twitter、LivingSocial、Groupon、Hulu、Yellow Pagesなど、Ruby on Railsを使ってるサイトのリストは増える一方です。ENTP、thoughtbot、Pivotal Labs、Hashrocket、そして無数のコンサルタント、インストラクターなど、Railsを専門に扱う開発者も多く存在します。

Railsのどこがそんなに凄いのでしょうか？第一に、Railsは100%オープンソースで、制限の緩いMITライセンスで公開されているので、ダウンロードも使用も無料であるという点です。Railsがこれほどまでに成功をおさめ、優美でコンパクトなコードを記述できるのは、背後でフレームワークを動かしているRubyという言語の柔軟性を利用してWebアプリケーション作成に特化したDSL（ドメイン固有言語）を実装しているからです。このため、HTMLの生成、データモデルの作成、そしてURLの転送など、Webプログラミングで必要な多くの作業が簡単になり、アプリケーションコードが読みやすく、短く収まります。

第二に、Railsは、Webテクノロジーやフレームワークデザインの進歩に素早く適応します。たとえば、RailsはWebアプリケーション構築の際に使用されるRESTアーキテクチャをいち早く完全に実装したフレームワークの1つです（RESTについてはこのチュートリアルで後々解説していきます）。また、Railsの作成者であるDavid Heinemeier HanssonをはじめとするRailsのコアチームは、ためらうことなく他のフレームワークの有用な機能をどしどし取り入れます。RailsとMerbの合流はその一例です。結果としてRailsは、Merbのモジュール化されたデザインと安定したAPIの恩恵を受ける事ができました。

最後に、Railsコミュニティは非常に熱心で様々な経験の開発者の集まりです。結果、何百人のオープンソースコントリビュータたち、多くの参加者で賑わう各種カンファレンス、多様多種なプラグイン、ページネーションや画像アップロードなど特定の作業を行うためのおびただしいgem、情報豊かな多数のブログ、そしてフォーラムや掲示板、チャットルームなどが存在します。数多くのRailsプログラマのおかげで、ほとんどのアプリケーションエラーはGoogle検索をするだけで、関連するブログポストや掲示板のスレッドを見ることができます。

1.1.1 読者の皆さまへのコメント

RailsチュートリアルにはRailsだけではなく、Ruby言語、HTML、CSS、若干のJavaScript、そしてさらにわずかのSQLのためのチュートリアルを含んでいます。あなたの現時点のWeb開発の知識量にかかわらず、このチュートリアルを読み終える頃には、高度な情報源を難なく読みこなし、各トピックをさらにシステムチックに取り扱えるほどの実力が付く内容になっています。これは、本書で扱われているマテリアルが極めて膨大であるということでもあります。コンピュータのプログラミング経験があまりない人にとっては、その情報量に圧倒されることがあるかもしれません。読者の経験や知識に応じてRailsチュートリアルを習得するためのアドバイスを以下に用意しました。ぜひ参考にしてください。

い。

すべての読者へ: 私が良く聞かれる質問の1つが、「Railsを覚える前にRubyを勉強しておいた方がいいでしょうか」です。答えは「あなた自身の学習スタイルとプログラミング経験によります」です。最初からすべてを体系的に学びたい、またはプログラミングの経験がまったくない場合には、おそらくRubyを最初に学んでおくのがよいでしょう。Rubyを学ぶのであれば、Peter Cooper 「[Beginning Ruby](#)」(邦訳は現時点ではなし)がお勧めです。一方、Rails開発初心者の多くはWebアプリケーションの作成にたちまち夢中になり、たった1つのWebページを作成するために500ページものRuby本を読み通す気にはなれないでしょう。このような方には、対話的にRubyを学習できるWebサービス[Try Ruby](#)²でRubyを大まかに理解し、必要に応じて[Rails for Zombies](#)³でRailsでどんなことができるかを学んでおくことをお勧めします(いずれも日本語版はなし)。

もう1つよく聞かれる質問で「最初からテストしていいのか?」ということがあります。「はじめに」で述べたように、Railsチュートリアルでは最初からテスト駆動開発(TDD)を採用しています。これは私からみてRailsアプリケーションを開発するのに最適な方法ですが、その分かなり複雑になり、習得に時間もかかります。テストが面倒になったり、難しくて前に進めなくなった場合、思い切ってその箇所をスキップするか、テストはコード検証のためのツールと割りきって単に動かすだけにとどめ、理解は後回しにしてしまって構いません。後者でいうテストとは、specと呼ばれるテストファイルを作成し、そこにテストコードを本書で指示されたとおりにコピペすることです。その後テストスイートを実行します。[第5章](#)で実践しますが、最初はテストに失敗し、次にチュートリアルに従ってコードを書くと次のテストにはパスするようにシナリオが作られています。

経験の浅いプログラマ: ご注意いただきたいのですが、Railsチュートリアルは、まったくのプログラミング初心者、Webアプリケーションの初心者を主な読者層としては想定していません。プログラミング言語やWebアプリケーションは、たとえどれほどシンプルなものであっても必ずある程度には複雑なものだからです。Webプログラミングについてまったくの初心者の方が、本書Railsチュートリアルが難しすぎると感じた場合は、まずHTMLとCSSの基礎を勉強してください。それからRailsチュートリアルに再度挑戦してください。(現時点でお勧めの学習教材を思いつかず恐縮ですが、HTMLの学習用として[Head First HTML](#)はなかなかよさそうです。David Sawyer McFarlandの[CSS: The Missing Manual](#)を推薦してくれた読者もいます。)Peter Cooperの[Beginning Ruby](#)の最初の数章をやってみるのもよいでしょう。同書のサンプルアプリケーションは本格的なWebアプリケーションに比べて非常に小さいので、比較的やりやすいと思います。実のところ、驚くほど多くのプログラミング初心者の皆様が本書でWeb開発を学ぼうとしているのです。もちろん、それを止めるようなことはしません。ぜひ挑戦してみてください。そうした方には、[Rails Tutorial screencast series](#)⁴(いずれも日本語版はなし)を特にお勧めします。このビデオチュートリアルは、Railsソフトウェア開発のベテランの作業を文字通り「肩越しに」学ぶことができます。

ソフトウェア開発経験があり、Web開発を始めたい方: プログラミングの経験がある方でクラス、メソッド、データ構造などに馴染みがあるなら、その知識は有利に働くはずです。JavaやC/C++とRubyではかなり記述スタイルが違うので少し違和感を感じるかもしれません、すぐに慣れるはずです。(Rubyではセミコロン';'も使えます) RailsチュートリアルはWeb特有の概念をほぼカバーしているので、PUTとPOSTの違いが分からなくても心配はいりません。

Web開発の経験があり、Railsを始めたい方: PHPやPythonなど動的言語を使用した事があ

るなら、かなり楽に読めるかと思います。特にPythonの経験は有利です。基本的なコンセプトは聞いた事あるものばかりかもしれません、テスト駆動開発(TDD)やRESTスタイルなどは多くの方にとって初耳かもしれません。Ruby特有のクセも、最初はとまどうかもしれません、チュートリアルを進めて行くうちに明らかになっていくと思います。

Railsを使った事がないRubyプログラマの方: 最近では多くのRubyプログラマがRailsを使用しているので、純粋にRubyのみを使っている方は少ないかと思いますが、経験豊富なRubyプログラマの方にはObie Fernandezの[The Rails 3 Way](#)をお勧めします。

Rails開発者で経験の少ない方: もしかしたら既に他のチュートリアルを読んだり小規模なRailsアプリを組んだことがあるかもしれません。Rails開発者からも、本書から多くのことを学べたという声をたくさんいただいている。本書は、あなたがRailsを始めて使い始めた頃のチュートリアルよりも新しい情報がふんだんに盛り込まれており、きっと役に立つはずです。

Rails開発者で経験豊富な方: 本書はあなたには必要ないかもしれません、Rails開発者の方からも本書が「意外と役に立った」との声をいただいているので、Railsを違う視点から見てみるのも楽しいかもしれません。

Ruby on Rails チュートリアルを読み終えたら、(Ruby以外での)開発経験のある方には、Rubyを一から覚えることができ、詳細な解説も載っているDavid A. Blackの[The Well-Grounded Rubyist](#)か、似たような内容で項目ごとに分かれているHal Fultonの[The Ruby Way](#)をお勧めします。その後は[The Rails 3 Way](#)でRailsの知識を深めると良いでしょう。

全部終わった頃には、より高レベルのRailsガイドや情報源を読めるようになっているはずです。私が特にお勧めしたいのは以下です：

- [RailsCasts \(Ryan Bates\)](#) : 質の高い、ほぼ無料のスクリーンキャスト集
- [PeepCode](#): 質の高い商用スクリーンキャスト
- [Code School](#): 対話的にプログラミングを学習できるコース
- [Rails Guides](#): テーマごとに分類された最新のRailsリファレンスです。
- [RailsCasts \(Ryan Bates\)](#): おや、さっきも [RailsCasts](#)のことを書いたような気が。そのぐらい[RailsCasts](#)はお勧めです。

1.1.2 Railsと「スケール」について

この節を続ける前に、Railsフレームワークにとって当初から「ネック」とされてきた事の1つが、多くのデータとユーザによって発生する通信量に対応するための「スケーリングの難しさ」です。ここで1つ言つておきます。ここで必要なのはフレームワークをスケールさせるのではなくて、[Webサイト自体をスケールさせること](#)です。Railsは言うまでもなく素晴らしいのですが、結局はただのフレームワークです。すなわち、問題はRailsがスケールできるかどうかではなく、「スケールするアプリケーションをRailsで組む事ができるか」なのです。どっちにしろ世界で最もトラフィックの多いサイトがRailsを使用している今、答えはYESです。スケーリング自体はRailsだけの問題ではありませんが、もしあなたのアプリケーションがHuluやYellow Pages並のデータ量を扱わなければならなくなってもRailsはあなたの世界征服を止める事はないでしょう。

1.1.3 この本における取り決め

本書における取り決め(コマンドやコードの記述法などの統一)はできるだけ分かりやすく選んだつもりです。ここでは分かりにくいかもしれない部分を解説しておきます。

HTML版、PDF版とともに、文中には内部リンク ([1.2など](#)) と外部リンク (Ruby on Railsダウンロードページなど) が多数含まれています⁵。

本書の出てくる例の多くはコマンドライン(ターミナル)のコマンドを使っています。簡素化のため、すべてのコマンドライン例は次のようにUnixスタイルの\$プロンプトを使っています。

```
$ echo "hello, world"  
"hello world"
```

Windowsユーザの方は、Windowsコマンドラインでは次のように>プロンプトが使用されている事をご理解ください。

```
C:\Sites> echo "hello, world"  
"hello world"
```

Unix上では、コマンドを入力する際sudo(他のユーザーの特権レベルでプログラムを実行するためのコマンド)を使わなければならない場合があります⁶ デフォルトでsudoコマンドで実行されたプログラムは、[1.2.2](#)の例のように通常のユーザはアクセスできないファイルやディレクトリへのアクセス権を持つ「管理者権限」で実行されます。

```
$ sudo ruby setup.rb
```

Section [1.2.2.3](#)で推奨されているRuby Version Manager (rvm)を導入していない場合、ほとんどのUnix/Linux/OS Xシステムではsudoの実行が必要です。rvmを導入している場合は以下のように実行できます。

```
$ ruby setup.rb
```

Railsにはコマンドラインで実行できる多数のコマンドが付属しています。たとえば、[1.2.5](#)にあるように以下のコマンドを使用して開発用WebサーバーをローカルPC上で実行することができます。

```
$ rails server
```

コマンドラインのプロンプトと同じく、Railsチュートリアルではディレクトリの区切りをUnix式にスラッシュ(/)で表記しています。たとえばサンプルアプリケーションの場所は以下になります。

```
/Users/mhartl/rails_projects/sample_app
```

Windowsでの同等のパスは以下になります。

```
C:\Sites\sample_app
```

Railsで作成するアプリケーションのルートディレクトリをRailsルートと呼びます。この用語はやや間違えられやすいようで、多くの人がRailsルートを「Railsそのもののルート」だ

と勘違いします。この点を明確にするため、*Rails*チュートリアルではRailsルートといえば*Rails*で作成したアプリケーションのルートを指すものとし、すべてのディレクトリはここを起点とした相対パスで示します。たとえば、サンプルアプリケーションの**config**ディレクトリは以下です。

```
/Users/mhartl/rails_projects/sample_app/config
```

この場合、このRailsアプリケーションのルートディレクトリは**config**の1つ上の階層であり、以下になります。

```
/Users/mhartl/rails_projects/sample_app
```

以下のようなディレクトリパスを簡潔に記述する場合、

```
/Users/mhartl/rails_projects/sample_app/config/routes.rb
```

本書ではアプリケーションのルートを省略して単に**config/routes.rb**と記載します。

*Rails*チュートリアルには、シェルコマンドやバージョン管理ソフトウェア、Rubyプログラムなどが出力するさまざまな結果が多数掲載されています。出力結果はコンピュータシステムによって微妙に異なるのが普通ですから、本書に掲載されている出力結果が実際の出力と正確に同じであるとは限りませんが、この点は問題ありません。

システムの状態によっては、一部のコマンドを実行した時にエラーが発生することもあります。本書では、あらゆる場合を想定してエラー対策を事細かに記載するような**面倒なこと**はしていません。代わりに、エラーメッセージを素直にGoogleで検索してください。これは実際のソフトウェア開発にも効果的なテクニックです。本書のチュートリアル実行中に何か問題が生じたら、*Rails*チュートリアルのHelpページに記載されているリソースに当たってみてください⁷。

1.2 さっそく動作させる

本書の第1章は、さしづめロースクールで言うところの「草むしり期間」(=選別期間)のようなものです。ここで開発環境の構築に成功できた人ならば、最後までやり通すのは難しくありません。

—Bob Cavezza (*Rails*チュートリアル読者)

いよいよ、Ruby on Railsの開発環境の構築と最初のアプリの作成をする時が来ました。特にプログラミング経験が無い方は少し苦労するかもしれません、どうか諦めずに頑張ってください。苦労するのはあなただけではありません。これは開発者として誰もが何度も通る道であり、苦労は必ず大いに報われることを私が保証いたします。

1.2.1 開発環境

開発環境はRailsプログラマの数だけあります。開発者たちは自分の環境を、最早自分でもわからなくなるぐらいにとことんカスタマイズするものだからです。開発環境は、大きく分けてテキストエディタとコマンドラインを使用する環境とIDE(統合開発環境)を使った環境に分けられます。まずは後者から見てみましょう。

IDE

Rails用のIDEの数は多く、[RadRails](#)、[RubyMine](#)、[3rd Rail](#)などさまざまなものがあります。特にRubyMineの評判がよいようです。さらに、David Loefflerという読者が[RailsチュートリアルでRubyMineを使用する方法](#)⁸ というドキュメントまでまとめてくれました。IDEでの開発が好みという方は、自分のスタイルに合うかどうか上のリンクをチェックしてみるとよいでしょう。

テキストエディタとコマンドライン

私自身は、すべてが整った強力なIDEの代わりに、テキストエディタでテキストを編集し、コマンドラインでコマンドを実行するというシンプルな手段を好んでいます(図 1.1)。どの組み合わせを使うかは自身の好みとプラットフォームによりますが。

- **テキストエディタ:** 私は断然[Sublime Text 2](#)をお勧めします。これは非常によくできたクロスプラットフォームのテキストエディタであり、現段階ではまだベータ版ですが、とてもそうは思えないほどパワフルです。Sublime Textは[TextMate](#)の影響を強く受けています。そして実際、スニペットやカラースキームなどについてはTextMateのカスタマイズ設定と互換性があります。(あなたがMacユーザーであれば、TextMateは今でも良い選択であると言えます。OS X版しかありませんが。)2番目の選択肢にふさわしい[Vim](#)⁹ は多くのプラットフォームで動作します。Vimは無償で手に入りますが、Sublime Textは商用版しかありません。どちらもプロの使用に耐えうるものですが、私の経験ではSublime Textは初心者にとってはるかに扱いやすいものであると言えます。
- **ターミナル:** OS Xの場合、[iTerm](#)かシステム備え付けのTerminal appをお勧めします。Linuxの場合、デフォルトのターミナルで構いません。Windowsの場合、Rails環境をそのままインストールするよりも、Linuxが動作する仮想マシン上にインストールすることを好む人が多いようです。この場合、コマンドラインオプションは従来通りです。開発環境をWindows自身にインストールするのであれば、コマンドラインは[Railsインストーラ付属の物](#)がおすすめです([1.2.2.1](#))。

Sublime Textを選んだ方のために、オプション設定方法を解説した[Railsチュートリアル: Sublime Text](#)¹⁰ を用意しています。(この種の設定はいろいろと面倒でエラーも起きやすいので、あくまで上級者向けです。Sublime Textはデフォルト設定のままでも依然としてRailsアプリケーションを編集するための良い選択肢です。)

```

[sample_app (master)]$ ls -l
total 16
-rw-r--r-- 1 mhartl mhartl 226 Oct 16 14:57 README.mdown
-rw-r--r-- 1 mhartl mhartl 387 Oct 20 13:00 Rakefile
drwxr-xr-x 7 mhartl mhartl 238 Dec 12 21:01 app
drwxr-xr-x 9 mhartl mhartl 306 Mar 15 21:15 config
drwxr-xr-x 8 mhartl mhartl 272 Apr 7 18:34 db
drwxr-xr-x 3 mhartl mhartl 182 Oct 14 16:03 doc
drwxr-xr-x 3 mhartl mhartl 182 Oct 14 16:03 lib
drwxr-xr-x 6 mhartl mhartl 284 Oct 14 16:03 log
drwxr-xr-x 11 mhartl mhartl 374 Nov 30 18:31 public
drwxr-xr-x 13 mhartl mhartl 442 Dec 12 21:01 script
drwxr-xr-x 11 mhartl mhartl 374 Apr 7 18:37 spec
drwxr-xr-x 6 mhartl mhartl 284 Feb 2 16:13 test
drwxr-xr-x 81 mhartl mhartl 2754 Mar 19 12:38 tmp
drwxr-xr-x 3 mhartl mhartl 182 Nov 28 11:58 vendor
[sample_app (master)]$ 

```

```

class UsersController < ApplicationController
  before_filter :authenticate, :only => [:index, :edit, :update, :destroy]
  before_filter :correct_user, :only => [:edit, :update]
  before_filter :admin_user, :only => :destroy
  ...
  def index
    @title = "All users"
    @users = User.paginate(:page => params[:page])
  end
  ...
  def show
    @user = User.find(params[:id])
    @title = CGI.escapeHTML(@user.name)
  end
  ...
  def new
    @title = "Sign up"
    @user = User.new
  end
  ...
  def create
    @user = User.new(params[:user])
    if @user.save
      sign_in @user
      flash[:success] = "Welcome to the Sample App!!"
      redirect_to @user
    else
      @title = "Sign up"
      render 'new'
    end
  end
  ...
  def edit
    @title = "Edit user"
  end
  ...
  def update
    if @user.update_attributes(params[:user])
      flash[:success] = "Profile updated."
      redirect_to @user
    else
      @title = "Edit user"
      render 'edit'
    end
  end
end

```

図1.1： テキストエディタ/コマンドライン開発環境(TextMate/ ITERMを使用) (拡大)

ブラウザ

ブラウザの種類は豊富ですが、大半のRails開発者はFirefox、Safari、もしくはChromeを選ぶ傾向にあるようです。Railsチュートリアル内のスクリーンショットも原則としてFirefoxの物を使用しています。Firefoxを使う場合、HTMLの構造とCSSのルールをダイナミックに調査/編集できる[Firebug](#)アドオンがおすすめです。Firefoxを使用しない場合、SafariとChromeにはページの調査したい部分を右クリックする事で情報を調べられる機能があります。

使用するツールについて

開発環境を整える過程で、すべてを思い通りに設定するにはかなり長い時間がかかる事を実感するかもしれません。Sublime TextやVimなどのエディタやIDEは特に、何週間かかっても極めるのは難しいものです。初めての方に申し上げておきますが、心配はいりません。開発ツールを覚えるのに時間をかけるのはいたって普通の事なのです。これもまた、開発者なら誰もが通る道です。「良いアイディアが浮かんで今すぐにでもRailsでWebアプリを作りたいだけなのに、Unixエディタの使い方を覚えるのに1週間もかかって前に進めない！」とイライラする事もあるでしょう。しかしながら、道具の使い方は職人にとって当然の心得です。払った努力に見合は見返りは必ずあります。

1.2.2 Ruby、RubyGems、Rails、Git

実質的に世界中のあらゆるソフトウェアは、壊れているか使いにくいかのどちらかだ。人々がソフトウェアに恐怖心を抱くのは、結局これが原因なのだ。人々は何かインストールしようとしたりオンラインフォームに記入したりするたびに、それらがさっぱり動かないという事態にすっかり慣れてしまっている。正

直、私は何かをインストールすることが怖い。これでも私はコンピュータサイエンスの博士号を持っているのだが。

—Paul Graham (*Founders at Work*)

さあ、RubyとRailsをインストールしましょう。本書では可能な限り多くの環境をカバーするようにしていますが、システムが変われば手順がうまくいかないこともあります。問題が生じた場合は、エラーメッセージをGoogleで検索するか、RailsチュートリアルのHelpページを参照してください。

警告: 特に断りのない限り、Railsを含むすべてのソフトウェアはチュートリアルで使用されているものと正確に同じバージョンを使用してください。そうでないと同じ結果を得られないことがあります。バージョンが少々異なっていても同じ結果を得られることもありますが、特にRailsのバージョンに関しては必ず指定を守ってください。これには例外もあり、たとえばRubyそれ自身が該当します。**1.9.2**と**1.9.3**は実質的に同じであり、チュートリアルには影響しません。お好きな方をご使用ください。

Railsインストーラ (Windows)

最近までWindowsでのRailsのインストールは困難だったのですが、Engine Yardの方々(特にDr. Nic WilliamsとWayne E. Seguin)のおかげで非常に簡単にになりました。Windowsユーザーの方はRailsインストーラからインストーラをダウンロードして下さい。exeファイルをダブルクリックし、指示に従ってGit、Ruby、RubyGems、そしてRails自身をインストールします(**1.2.2.2**、**1.2.2.3**、**1.2.2.4**、**1.2.2.5**はスキップしてください)。インストール完了後、**1.2.3**までスキップして最初のアプリケーション作成に取りかかってください。

注意: このRailsインストーラによってインストールされるRailsのバージョンは**1.2.2.5**に記載のものと異なることがあります。その場合、互換性の問題が生じることもあります。この問題を解決するために、Railsのバージョン番号順に並んだRubyインストーラのリストを作成してもらうようNicとWayneに働きかけています。

Gitのインストール

Rails環境の多くの部分は、Gitというバージョン管理システムの機能に依存します(Gitの詳細については**1.3**で解説しています)。Gitは本書で多用されているので、早い内にインストールを済ませておきます。Pro Gitの「Gitのインストール」でプラットフォーム毎の解説を行っているので参考にして下さい。

Rubyのインストール

次にRubyをインストールします。もしかすると既にあなたのシステムに入っているかもしれない、その場合は以下を実行して

```
$Ruby-v ruby "1.9.3"
```

バージョン情報を確認してください。Rails 3はRuby 1.8.7以上で動作しますが、1.9.2が最適です。このチュートリアルはバージョン1.9.2または1.9.3を使用していることを前提で進めていますが、1.8.7でも問題はないはずです(ただし1つだけ文法の違いがあり、第4章で補足しています。また、出力に若干の違いが生じます)。

OS X、またはLinuxを使う場合、Ruby Version Manager (RVM)を使用してRubyをインス

トルすることを強くお勧めします。RVMを使うと、複数のRubyバージョンを共存させられるのでとても便利です。(Windows上で動作する同様のソフトにPikがあります。)同じマシン上で異なるバージョンのRubyやRailsを実行したい場合、これは特に重要です。RVMで問題が生じたときは、作者のWayne E. SeguinとRVMのIRCチャットのチャンネル (#rvm on freenode.net)で連絡を取れることがあります¹¹。Linuxを使用している方には、Sudobitsブログの「UbuntuにRuby on Railsをインストールする方法」が特にお勧めです。

RVMをインストール後、以下を実行してRubyをインストールします¹²。

```
$ rvm get head && rvm reload  
$ rvm install 1.9.3  
<しばらく待つ>
```

最初のコマンドはRVMの更新とリロードを行います。RVMは頻繁に更新されているので、RVM操作の際にはこれを行うようにしてください。次のコマンドはRuby 1.9.3をインストールします。システムによってはダウンロードとコンパイルに多少時間がかかることがありますので、なかなか終わらなくとも心配せずに待ってください。

OS Xユーザーの場合、**autoconf**の実行ファイルがないと問題が生じことがあります。

この場合はHomebrew¹³というOS X用のパッケージ管理システムをインストールしてから以下を実行してください。

```
$ brew install automake  
$ rvm install 1.9.3
```

Linuxユーザーの場合、OpenSSLというライブラリへのパスを追加する必要が生じことがあるとの報告を受けました。

```
$ rvm install 1.9.3 --with-openssl-dir=$HOME/.rvm/
```

一部の古いOS Xシステムでは、readlineライブラリへのパスを追加する必要が生じる場合があります。

```
$ rvm install 1.9.3 --with-readline-dir=/opt/local
```

(何度も書きましたが、環境構築にトラブルはつきものです。地道にネットを検索し、問題を特定することが唯一の解決策です。)

Rubyをインストールしたら、Railsのアプリケーションを実行するために必要な他のソフトウェア向けにシステムを構成する必要があります。通常、これはgemのインストールに関連します。gemとは自己完結型のRubyコードのパッケージです。バージョン番号の異なるgem同士がコンフリクトすることがあるため、一連のgemを自己完結的にまとめたgemsetというものを作成してバージョンを使い分けるのが便利です。本書のチュートリアル用に、以下の要領で**rails3tutorial2ndEd**というgemsetをぜひ作成しておいてください。

```
$ rvm use 1.9.3@rails3tutorial2ndEd --create --default  
Using /Users/mhartl/.rvm/gems/ruby-1.9.3 with gemset rails3tutorial2ndEd
```

このコマンドを実行すると、Ruby 1.9.3に関連付けて**rails3tutorial2ndEd**という

gemsetを作成し(--createオプション)、このgemsetをただちに使用可能にします(use)とデフォルト指定用の(--default)。以後、**1.9.3@rails3tutorial2ndEd**というRubyとgemsetの組み合わせがすべてのターミナルウィンドウで使用可能になります。RVMにはgemset管理のためのコマンドが多数用意されています。詳細については<http://rvm.beginrescueend.com/gemsets/>を参照してください。RVMがうまく動かない場合は以下を実行してヘルプを表示してください。

```
$ rvm --help  
$ rvm gemset --help
```

RubyGemsのインストール

RubyGemsはRubyのプロジェクトのためのパッケージマネージャであり、Rubyのパッケージ(gem)として利用できる多くの有用なライブラリがあります。Railsもgemとしてインストールします。Rubyがインストールされていれば、RubyGemsは簡単にインストールできます。RVMをインストールしてあれば、既にRubyGemsも同時にインストールされているはずです。

```
$ which gem  
/Users/mhartl/.rvm/rubies/ruby-1.9.3-p0/bin/gem
```

RubyGemsがインストールされていない場合は、RubyGemsをダウンロードして解凍し、作成された**rubygems**ディレクトリでセットアッププログラムを実行して下さい。

```
$ ruby setup.rb
```

(アクセス権の問題が生じたら、**1.1.3**を参照してください。おそらく**sudo**コマンドを併用する必要があります。)

既にRubyGemsがインストールされている場合は、システムをチュートリアルで使われているバージョンに更新して下さい。

```
$ gem update --system 1.8.24
```

システムを特定のバージョンに固定しておけば、今後RubyGemsが変更されたときのコンフリクトを防止できます。

gemをインストールすると、RubyGemsによってriとrdocという2種類のドキュメントがデフォルトで作成されます。多くのRubyやRailsの開発者はこれらのドキュメントが自動生成される時間すら惜しいと考えます。(多くのプログラマーはネイティブのriやrdocなど参照せず、さっさとオンラインドキュメントを見に行ってしまいます。) riやrdocの自動生成を抑制するには、**.gemrc**というファイルをリスト**1.1**のようにホームディレクトリに作成し、そこにリスト**1.2**の内容を追加しておきます。(チルダ“~”は「ホームディレクトリ」を表します。**.gemrc**ファイルのようにファイル名冒頭にドット「.」を付けると隠しファイルとして扱われます。Unixの設定ファイルにはこのような名前を付けるのが習わしとなっています。)

リスト**1.1** gem設定ファイルを作成する。

```
$ subl ~/.gemrc
```

冒頭の**subl**はOS Xで Sublime Textを起動するコマンドです。 設定方法については[OS X コマンドライン用Sublime Text 2ドキュメント](#) (英語) を参照してください。使用しているプラットフォームやエディタが異なる場合、sublを他のエディタコマンドに読み替えてください (エディタのアイコンをダブルクリックするか、**mate**、**vim**、**gvim**、**mvim**などのコマンドに差し替えます)。記述を簡素化するため、本書で以後**subl**と書かれていたら各自好みのエディタに読み替えてください。

リスト1.2 **.gemrc**にriとrdoc生成を抑制するコマンドを追加する。

```
install: --no-rdoc --no-ri  
update: --no-rdoc --no-ri
```

Railsのインストール

RubyGemsをインストールしてしまえば、Railsのインストールは簡単です。本書ではRails 3.2で統一します。以下を実行して3.2をインストールしてください。

```
$ gem install rails -v 3.2.13
```

正しくインストールされたかどうかを確認するには、以下のコマンドを実行してバージョンを確認してください。

```
$ rails -v  
Rails 3.2.13
```

メモ: [1.2.2.1](#)でRailsインストーラを使用してRailsをインストールした場合、Railsのバージョンが異なっている可能性があります。この記事の執筆時点では、このバージョンの違いは影響していません。ただし、今後Railsのバージョンが本書指定のものから離れていくにつれ、バージョンの違いによる影響が顕著になる可能性があります。特定のバージョンのRailsインストーラへのリンクを作成してもらえるよう、現在Engine Yardに働きかけています。

Linuxを実行している場合は、この時点で、他にもいくつかのパッケージをインストールする必要が生じる場合があります。

```
$ sudo apt-get install libxslt-dev libxml2-dev libsqlite3-dev # Linuxのみ
```

1.2.3 最初のアプリケーション

どんなRailsアプリケーションも最初の作成手順は同じです。**rails new**コマンドを実行して作成します。このコマンドを実行するだけで、指定のディレクトリにRailsアプリケーションのスケルトンを簡単に作成できます。これを行うには、まず複数のRailsプロジェクトを保存するためのディレクトリを作成し、**rails new**を実行して最初のアプリケーションを作成します ([リスト1.3](#))。

リスト1.3 **rails new**を実行してアプリケーションを新規作成する。

```
$ mkdir rails_projects
$ cd rails_projects
$ rails new first_app
  create
  create README.rdoc
  create Rakefile
  create config.ru
  create .gitignore
  create Gemfile
  create app
  create app/assets/images/rails.png
  create app/assets/javascripts/application.js
  create app/assets/stylesheets/application.css
  create app/controllers/application_controller.rb
  create app/helpers/application_helper.rb
  create app/mailers
  create app/models
  create app/views/layouts/application.html.erb
  create app/mailers/.gitkeep
  create app/models/.gitkeep
  create config
  create config/routes.rb
  create config/application.rb
  create config/environment.rb
.
.
.
  create vendor/plugins
  create vendor/plugins/.gitkeep
  run bundle install
Fetching source index for https://rubygems.org/
.
.
.
Your bundle is complete! Use `bundle show [gemname]` to see where a bundled
gem is installed.
```

Listing 1.3の下の方に注目してください。`rails new`を実行すると、ファイルの作成後に`bundle install`コマンドが自動的に実行されています。もし`bundle install`が実行されていなくても心配はいりません。落ち着いて1.2.4の手順に従えば実行できます。

`rails`コマンドによって非常に多くのファイルとディレクトリが作成されていることにご注目ください。ディレクトリとファイルの構造(図1.2)がこのように標準化されていることは数あるRailsの利点の1つであり、実際に動作するアプリケーションを何も無い状態からただちに作成することができます。さらに、この構造はすべてのRailsアプリに共通しているので、他の開発者の書いたコードの挙動を容易に推察できます。Railsがデフォルトで作成するファイルについては表1.1を参照してください。本書を通してこれらのファイルやディレクトリの目的を学んでいきます。特に、5.2.1以降ではRails 3.1の新機能であるアセットパイプラインの一部となる`app/assets`ディレクトリについて詳しく説明します。アセットパイプラインによって、CSS (Cascading Style Sheet) やJavaScriptファイルなどのアセットを簡単に編成したり展開することができます。

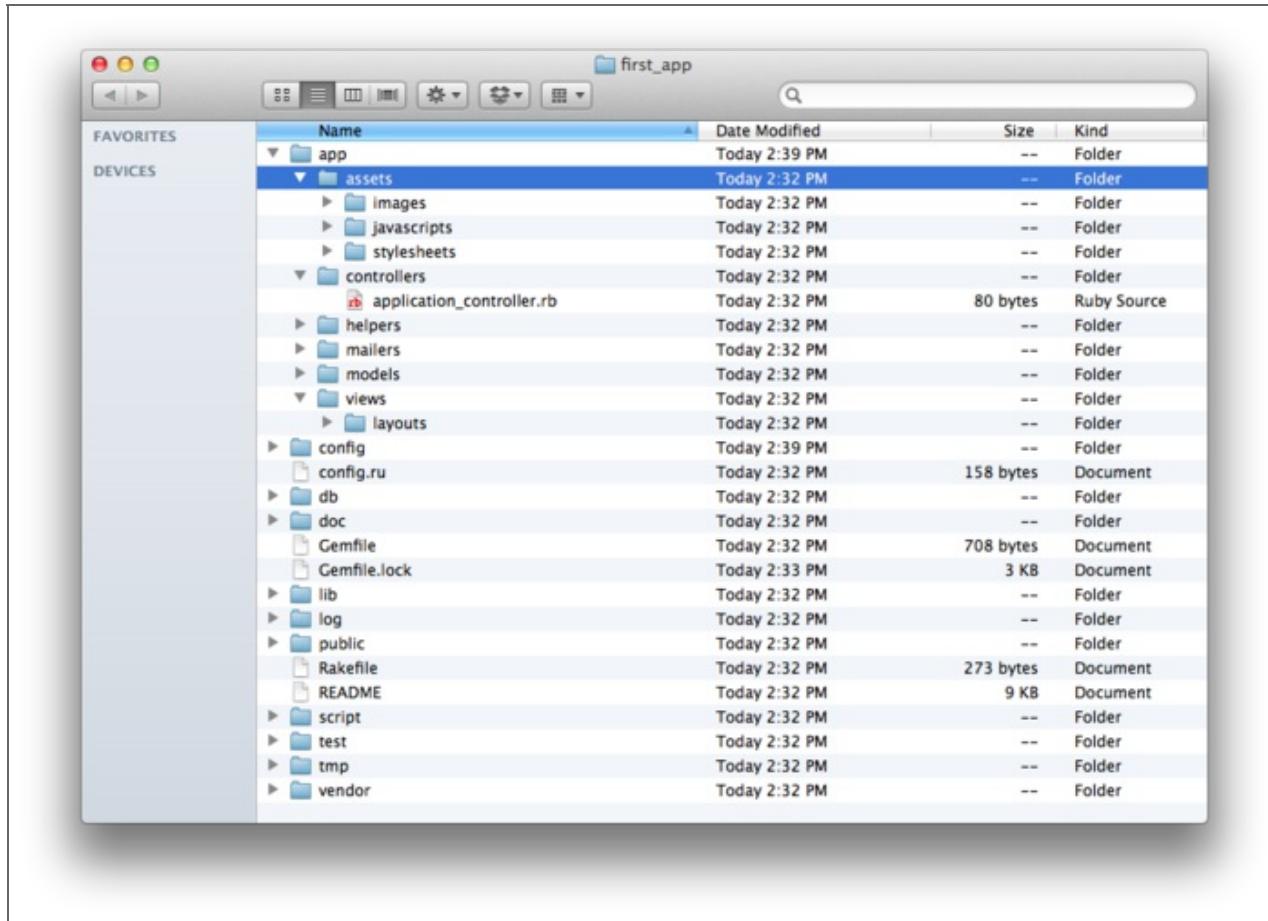


図1.2: 新規作成されたRailsアプリケーションのディレクトリ構造。(拡大)

ファイル/ディレクトリ

目的

app/	モデル、ビュー、コントローラ、ヘルパーなどを含む主要なアプリケーションコード
app/assets	アプリケーションで使用するCSS (Cascading Style Sheet)、JavaScriptファイル、画像などのアセット
config/	アプリケーションの設定
db/	データベース関連のファイル
doc/	マニュアルなど、アプリケーションのドキュメント
lib/	ライブラリモジュール
lib/assets	ライブラリで使用するCSS (Cascading Style Sheet)、JavaScriptファイル、画像などのアセット
log/	アプリケーションのログファイル
public/	エラーページなど、一般(Webブラウザなど)に直接公開するデータ
script/rails	コード生成、コンソールの起動、ローカルのWebサーバの立ち上げなどに使用するRailsスクリプト
test/	アプリケーションのテスト(3.1.2で作成する spec/ ディレクトリがあるため、現在は使用されていません)

tmp/	一時ファイル
vendor/	サードパーティのプラグインやgemなど
vendor/assets	サードパーティのプラグインやgemで使用するCSS (Cascading Style Sheet)、JavaScriptファイル、画像などのアセット
README.rdoc	アプリケーションの簡単な説明
Rakefile	rake コマンドで使用可能なタスク
Gemfile	このアプリケーションに必要なGemの定義ファイル
Gemfile.lock	アプリケーションのすべてのコピーが同じgemのバージョンを使用していることを確認するために使用されるgemのリスト
config.ru	Rackミドルウェア用の設定ファイル
.gitignore	Gitに無視させたいファイルを指定するためのパターン

表1.1: デフォルトのRailsフォルダ構造まとめ。

1.2.4 Bundler

Railsアプリケーションを新規作成したら、次は*Bundler*を実行して、アプリケーションに必要なgemをインストールおよびインクルードします。[1.2.3](#)でも簡単に説明したように、Bundlerは**rails**によって自動的に実行(この場合は**bundle install**)されます。ここではデフォルトのアプリケーションgemを変更してBundlerを再度実行してみます。そのためにはまず、好みのエディタで**Gemfile**を開きます。

```
$ cd first_app/
$ SUBL Gemfile
```

ファイルの内容は[リスト1.4](#)のようになっているはずです。Gemfileの内容はRubyのコードですが、ここでは文法を気にする必要はありません。Rubyについては[第4章](#)でもっとくわしく説明します。

リスト1.4 `first_app`ディレクトリ直下にあるデフォルトの**Gemfile**。

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

gem 'rails', '3.2.13'

# Bundle edge Rails instead:
# gem 'rails', :git => 'git://github.com/rails/rails.git'

gem 'sqlite3'

# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails',    '~> 3.2.3'
  gem 'coffee-rails',  '~> 3.2.2'

  gem 'uglifier', '>= 1.2.3'
end

gem 'jquery-rails'
```

```

# To use ActiveModel has_secure_password
# gem 'bcrypt-ruby', '~> 3.0.0'

# To use Jbuilder templates for JSON
# gem 'jbuilder'

# Use unicorn as the web server
# gem 'unicorn'

# Deploy with Capistrano
# gem 'capistrano'

# To use debugger
# gem 'ruby-debug19', :require => 'ruby-debug'

```

ほとんどの行はハッシュシンボル #でコメントされています。これらの行では、よく使われているgemとBundlerの文法の例をコメント形式で紹介しています。この時点では、デフォルト以外のgemは使用しません。デフォルトは、Rails自体のgem、アセットパイプライン関連のいくつかのgem ([5.2.1](#))、JQuery JavaScriptライブラリ関連のgem、SQLiteデータベースへのRubyインターフェイス用gemです。

gem コマンドで特定のバージョン番号を指定しない限り、Bundlerは自動的に最新バージョンのgemを取得してインストールしてくれます。残念ながらgemを更新すると小さな問題を起こすことがよくあるので、[リスト1.4](#)からコメントアウト行を除いた[リスト1.5](#)に示したように、このチュートリアルではたいていの場合動作確認済みのバージョン番号を指定しています。

リスト1.5 gemのバージョンを指定したGemfile

```

source 'https://rubygems.org'

gem 'rails', '3.2.13'

group :development do
  gem 'sqlite3', '1.3.5'
end

# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails',   '3.2.5'
  gem 'coffee-rails', '3.2.2'

  gem 'uglifier', '1.2.3'
end

gem 'jquery-rails', '2.0.2'

```

[リスト1.5](#)ではJQueryの行を変更しています。JQueryはRailsでデフォルトで使用されるJavaScriptのライブラリです。

```
gem 'jquery-rails'
```

上を以下のように変更します。

```
gem 'jquery-rails', '2.0.2'
```

他にも変更を行います。

```
gem 'sqlite3'
```

上を以下のように変更します。

```
group :development do
  gem 'sqlite3', '1.3.5'
end
```

この変更を行うと、Bundlerは**sqlite3** gemの**1.3.5**を強制的にインストールします。ここではさらに、SQLiteをdevelopment環境 ([7.1.1](#)) でのみ使用するための指定も行なっていますことに注意してください。こうすることで、Heroku ([1.4](#)) で使用するデータベースソフトウェアと衝突する可能性を避けられます。

リスト [1.5](#)では他にもいくつかの変更を行なっています。

```
group :assets do
  gem 'sass-rails',    '~> 3.2.3'
  gem 'coffee-rails',  '~> 3.2.2'
  gem 'uglifier',     '>= 1.2.3'
end
```

上を以下のように変更します。

```
group :assets do
  gem 'sass-rails',    '3.2.5'
  gem 'coffee-rails',  '3.2.2'
  gem 'uglifier',      '1.2.3'
end
```

以下の構文を実行すると

```
gem 'uglifier', '>= 1.2.3'
```

uglifyerのバージョンが**1.2.3**以上であれば最新バージョンのgemがインストールされます。極端に言えばバージョンが**7.2**であってもそれが最新ならインストールされます。なお、uglifyerはアセットパイプラインでファイル圧縮を行うためのものです。一方、以下のコードを実行すると

```
gem 'coffee-rails', '~> 3.2.2'
```

coffee-rails(これもアセットパイプラインで使用するgemです)のバージョンが**3.3**より小さい場合にインストールします。つまり、`>=`と書くと必ずアップグレードが行われますが、`~> 3.2.2`と書くとマイナーなアップグレードしか行われないことがあります。この場合**3.2.1**から**3.2.2**へのアップグレードは行われますが、**3.2**から**3.3**へのアップグレードは行われません。経験上、残念ながらマイナーアップグレードですら問題を引き起こすことがあります。このため、Railsチュートリアルでは基本的にすべてのgemでバージョンをピンポイントで指定しています。(例外として、執筆時点でベータ版やRC(Release Candidate)版のgemについては`~>`を使用し、最終版がリリースされたときに読み込まれるようにしてあります。)主にベテラン開発者の方へ: **Gemfile**で`~>`が指定されているgemも含め、基本的にはなるべく最新のgemを使用してください。ただし、それによって本書に記

載されているのと異なる挙動を示す可能性がある点にご注意ください。

Gemfileを正しく設定した後、**bundle update**¹⁴ と**bundle install**を使用してgemをインストールします。

```
$ bundle update
$ bundle install
Fetching source index for https://rubygems.org/
.
.
.
```

OS Xユーザーの場合、このときに**ruby.h**などのRubyヘッダファイルがないというエラーが表示されることがあります。その場合はXcodeをインストールしてください。この開発ツールはOS Xのインストーラディスクに含まれていますが、フルインストールするのは大きなので、サイズの小さい**Command Line Tools for Xcode**¹⁵だけをインストールすることをお勧めします。Nokogiri gemのインストール中にlibxmlエラーが発生した場合は、以下のようにRubyを再インストールしてください。

```
$ rvm reinstall 1.9.3
$ bundle install
```

bundle installコマンドの実行にはしばらく時間がかかるかもしれません。完了後、アプリケーションが実行可能になります。注: このセットアップ方法は最初のアプリケーションにはよいのですが、理想的な方法ではありません。[第3章](#)ではBundlerを使用してRuby gemをインストールするためのさらに強力で複雑な方法を紹介します。

1.2.5 rails server

[Section 1.2.3](#)の**rails new**コマンドと[1.2.4](#)の**bundle install**コマンドを実行したおかげで、実際に動かすことのできるアプリケーションが作成されました。うれしいことに、Railsには開発マシンからのみブラウズできるローカルWebサーバを起動させるためのコマンドラインプログラム(スクリプト)が付属しているので、以下のコマンドを実行するだけで簡単に起動することができます¹⁶。

```
$ rails server
=> Booting WEBrick
=> Rails application starting on http://0.0.0.0:3000
=> Call with -d to detach
=> Ctrl-C to shutdown server
```

(JavaScriptランタイムがインストールされていないというエラーが表示された場合は、[GitHub](#)のexecjsページにあるインストール可能なランタイムを確認してください。[Node.js](#)が特にお勧めです。) 表示されたメッセージは、Railsアプリケーションが

IPアドレス**0.0.0.0**、ポート番号 3000¹⁷で動作していることを示しています。このIPアドレスは、このマシンに設定されているすべてのIPアドレスで受信待ち(listen)するように指定しています。これにより、**127.0.0.1(localhost)**という特別なアドレスでアプリケーションをブラウズできます。[図1.3](#)は、<http://localhost:3000>をブラウザで表示した時の結果です。

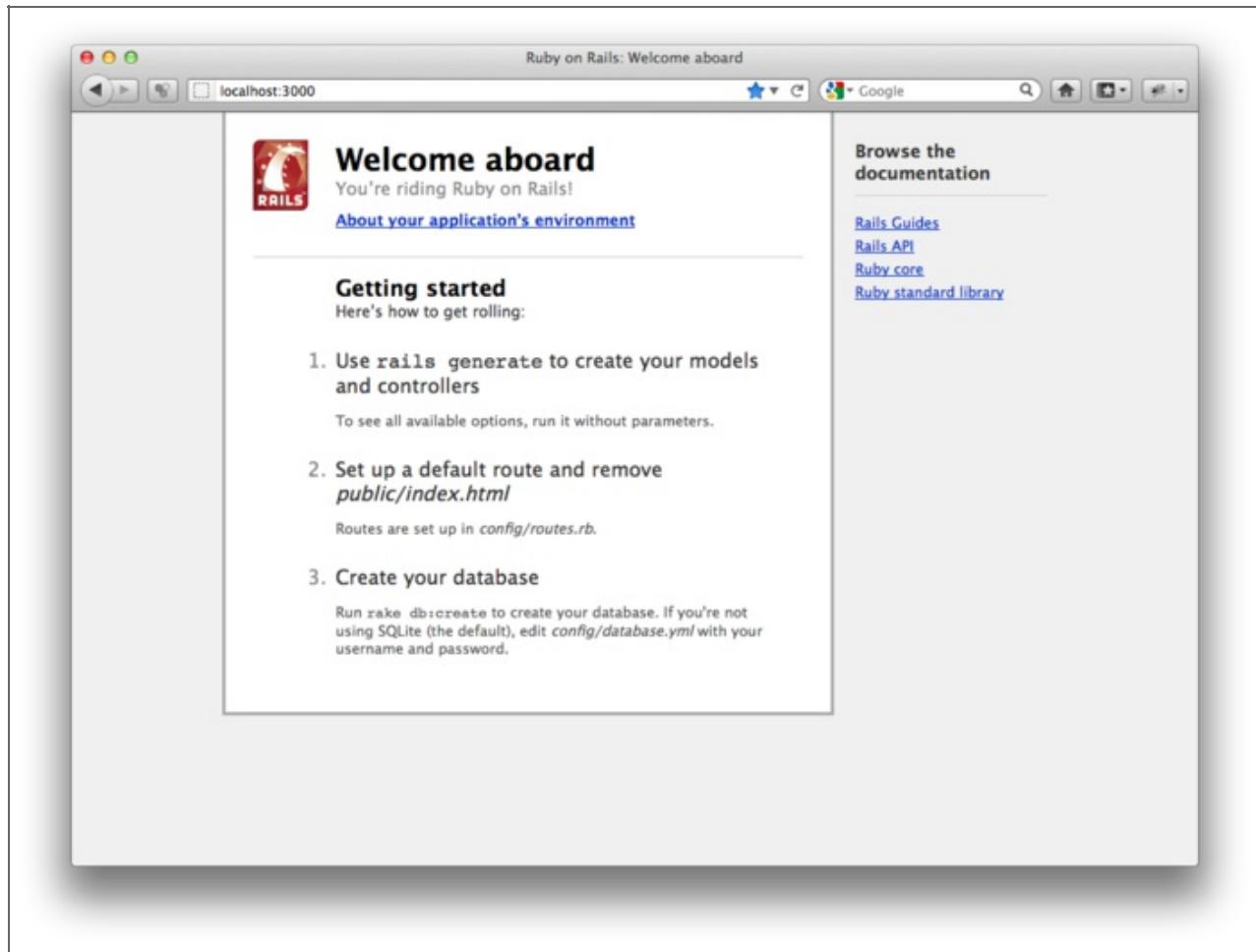


図1.3: デフォルトのRailsページ。 (拡大)

この最初のアプリケーションの情報を見るには、「About your application's environment」のリンクをクリックして下さい。図1.4のような画面が表示されます(図1.4は私の環境でキャプチャしたものなので、これとは若干異なる可能性があります)。

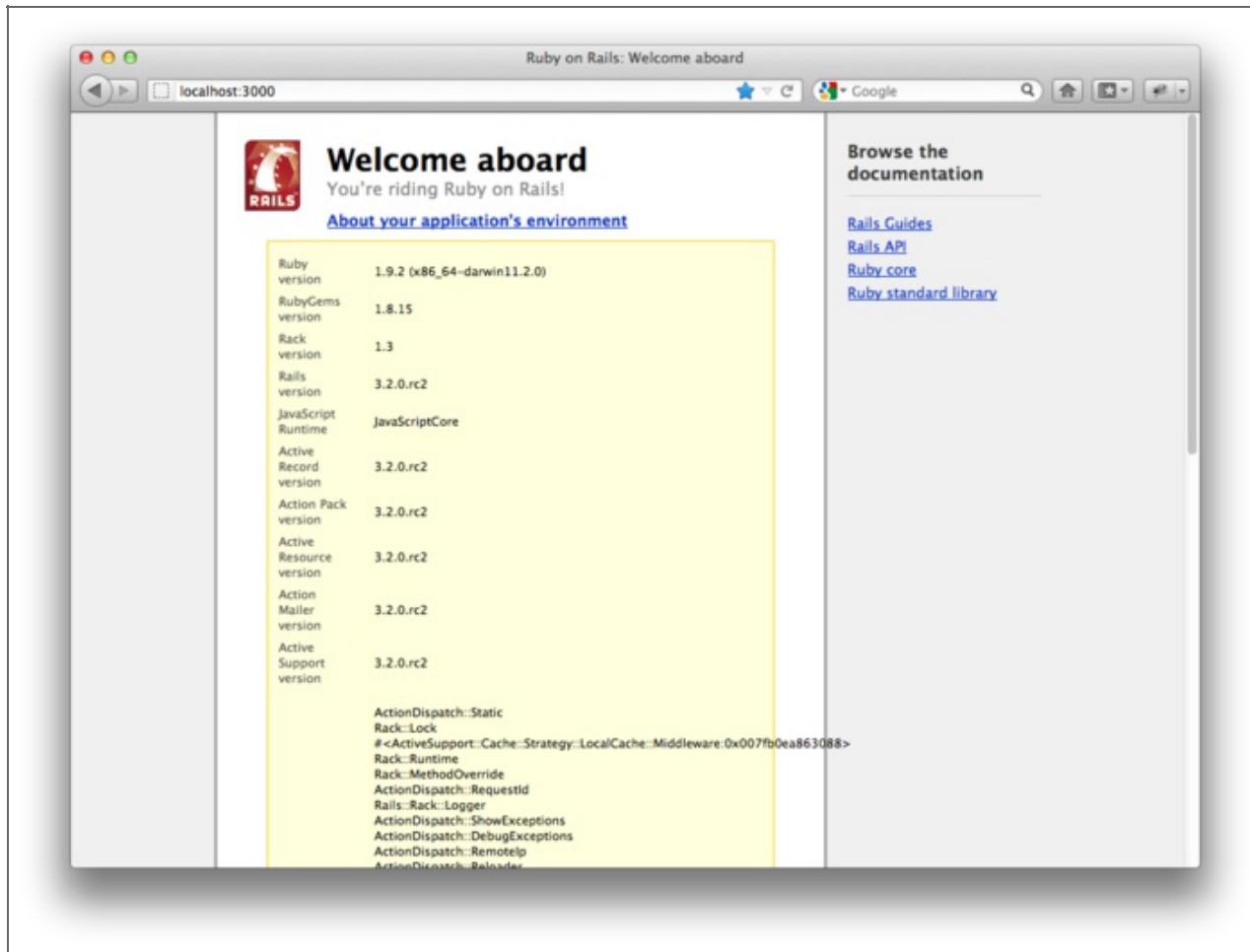


図1.4: アプリケーション環境が表示されているデフォルトページ（拡大）

もちろん、いずれデフォルトのRailsページは不要になりますが、アプリケーションが動いているのを見るのは気分のいいものです。[5.3.2](#)ではこのデフォルトページを削除し、カスタマイズしたホームページに置き換えます。

1.2.6 Model-View-Controller (MVC)

まだ始まったばかりですが、今のうちにRailsアプリケーションの全体的な仕組みを知っておくことは後々役立ちます([図1.5](#))。デフォルトのRailsアプリ構造([図1.2](#))を眺めてみると、**app/**というディレクトリがあり、その中に「**models**」「**views**」「**controllers**」という3つのサブディレクトリがあることに気付いた方もいると思います。ここにはRailsが**MVC (model-view-controller)**というアーキテクチャパターンを採用していることが暗に示されています。MVCでは、ドメインロジック(ビジネスロジックともいいます)と、グラフィカルユーザーインターフェース(GUI)と密に関連する入力/表示ロジックを分離します。Webアプリケーションの場合、「ドメインロジック」はユーザや記事、商品などのデータモデルに相当し、ユーザーインターフェースはWebページを指します。

Railsアプリと通信する際、ブラウザは一般的にWebサーバに*request*(リクエスト)を送信し、これはリクエストを処理する役割を担っているRailsの*controller*(コントローラ)に渡されます。コントローラは、場合によってはすぐに*view*(ビュー)を生成してHTMLをブラウザに送り返します。動的なサイトでは、一般にコントローラは(ユーザーなどの)サイトの要素を表しており、データベースとの通信を担当しているRubyのオブジェクトである*model*(モデル)と対話します。モデルを呼び出した後、コントローラは、ビューをレン

ダリングし、完成したWebページをHTMLとしてブラウザに返します。

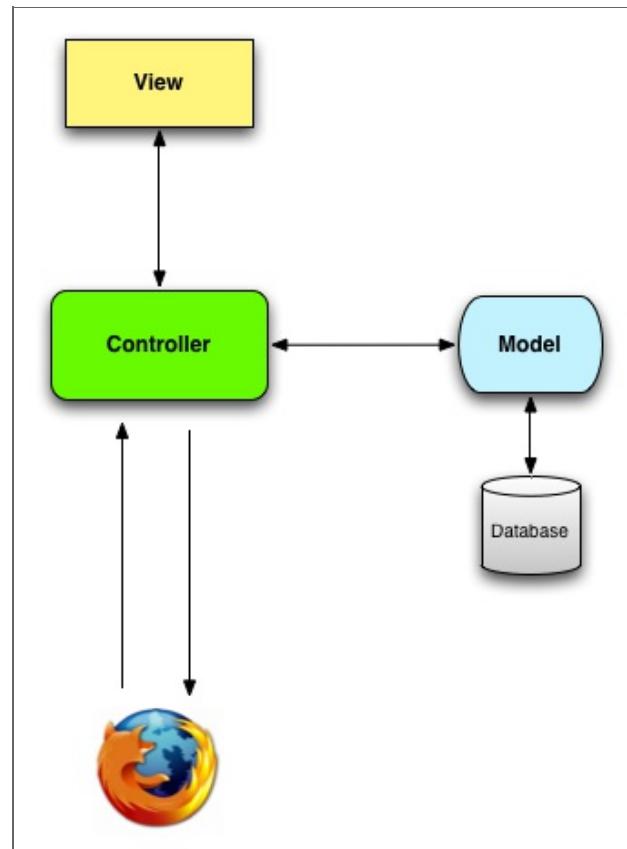


図1.5: model-view-controller (MVC) アーキテクチャの概念図。

この解説が少し抽象的に思えるかもしれません、この章は後に何度も参照する事になるのでご安心ください。さらに、[2.2.2](#)のデモアプリケーションの説明の中で、MVCについてもっと詳しく解説します。サンプリアプリケーションでは、MVCの3つの要素をすべて使用します。[3.1.2](#)ではまずコントローラとビューを使用し、モデルは[6.1](#)から使い始めます。[7.1.2](#)では3つの要素が協調して動作します。

1.3 Gitによるバージョン管理

新しく動作するRailsアプリが完成したところで、さっそくアプリケーションのソースコードをバージョン管理下に置きましょう。これを行わないとアプリケーションが動かないということではありませんが、ほとんどのRails開発者はバージョン管理は開発現場において必要不可欠であると考えています。バージョン管理システムを導入しておけば、プロジェクトのコードの履歴を追ったり、うっかり削除してしまったファイルを復旧(ロールバック)したりという作業が行えるようになります。バージョン管理システムを熟知することは、今やあらゆるソフトウェア開発者にとって必須のスキルであると言つてよいでしょう。

バージョン管理システムにもさまざまなものがありますが、RailsコミュニティではLinuxカーネル用にLinus Torvaldsにより開発された分散バージョン管理システムである[Git](#)が主流になっています。Git(というよりバージョン管理)はそれだけで大きなテーマなので、すべてを説明しようとすると軽く一冊の本を超てしまします。本書では簡単に言及するにとどめますが、幸いネット上には無償で利用できるリソースがあふれています。その中でも特に「*Pro Git*」 Scott Chacon (Apress, 2009) をお勧めいたします(訳注: Pro Gitには素晴らしい日本語版があります: <http://git-scm.com/book/ja/>)。ソースコードのバージョン管理

は何としても導入してください。バージョン管理はRailsを使用するどんな場面でも必要になりますし、バージョン管理システムを応用して、自分の作成したコードを他の開発者と簡単に共有したり [1.3.4\)](#)、最初の章で作成したアプリケーションを本番サーバーへ展開したりすることもできる([1.4](#))からです。

1.3.1 インストールとセットアップ

Gitがインストールされていない場合は、まず[1.2.2.2](#)に従ってGitをインストールしてください(リンク先の節でも述べているように、*Pro Git*の「Gitのインストール」の記載に従うことになります)。

最初のシステムセットアップ

Gitのインストール後、最初に1回だけ行う必要のある設定があります。これは*system*セットアップと呼ばれ、使用するコンピュータ1台につき1回だけ行います。

```
$ git config --global user.name "あなたの名前"  
$ git config --global user.email your.email@example.com
```

私の場合、**checkout**という長つたらしいコマンドの代わりに**co**という短いコマンド(エイリアス)も使えるようにしています。これを行うには以下を実行します。

```
$ git config --global alias.co checkout
```

本書では、**co**エイリアスを設定していないシステムでも動作するようにフルスペルの**checkout**を使用していますが、私自身は実際の開発ではほとんどいつも**git co**を使ってプロジェクトをチェックアウトしています。

手順の最後として、Gitのコミットメッセージを入力するときに使用するエディタを設定できます。Sublime Text、TextMate、gVim、MacVimなどのGUIエディタを使用する場合、シェルから離れずシェル内で起動するようフラグを付けてください¹⁸。

```
$ git config --global core.editor "subl -w"
```

"**subl -w**"の部分は、TextMateの場合は"**mate -w**"、gVimの場合は"**gvim -f**"、MacVimの場合は"**mvim -f**"にそれぞれ置き換えます。

最初のリポジトリセットアップ

今度は、レポジトリを作成するたびに必要な作業を行います。まず、Railsアプリケーションのルートディレクトリに移動し、新しいリポジトリの初期化を行います。

```
$ git init  
Initialized empty Git repository in /Users/mhartl/rails_projects/first_app/.git/
```

次に、プロジェクトのファイルをリポジトリに追加します。ここで1つ問題点があります。Gitはすべてのファイルの変更履歴を管理するようになっていますが、管理対象に含めたくないファイルもあります。たとえば、Railsによって作成されるログファイルは頻繁に内容が変わるので、いちいちバージョン管理に更新させたくないかもしれません。Gitにはこういった

ファイルを管理対象から除外する機能があります。`.gitignore`というファイルをアプリケーションのルートディレクトリに置き、除外したいファイルを指定するためのルールをそこに記載します。¹⁹

表1.1をもう一度見てみると、`rails`コマンドを実行した時にRailsアプリケーションのルートディレクトリに`.gitignore`ファイルが作成されています(リスト1.6)。

リスト1.6 `rails`コマンドで作成されるデフォルトの`.gitignore`ファイルの内容。

```
# See http://help.github.com/ignore-files/ for more about ignoring files.  
#  
# If you find yourself ignoring temporary files generated by your text editor  
# or operating system, you probably want to add a global ignore instead:  
#   git config --global core.excludesfile ~/.gitignore_global  
  
# Ignore bundler config  
.bundle  
  
# Ignore the default SQLite database.  
/db/*.sqlite3  
  
# Ignore all logfiles and tempfiles.  
/log/*.log  
/tmp
```

リスト1.6の設定では、ログファイル、Railsの一時ファイル(`tmp`)、SQLiteデータベースなどが除外されます。(`log/`ディレクトリ以下のログファイルを除外したい場合は、`log/*.*log`と指定することで、ファイル名が`.log`で終わるファイルが除外されます。これらのファイルは頻繁に更新されるため、バージョン管理に含めるのは何かと不便です。さらに、他の開発者と共同作業を行う場合にこのようなファイルをバージョン管理に含めると無用な衝突(conflict)が発生し、関係者一同が無駄にいらいらすることになりかねません。

本書のチュートリアルでは、リスト1.6の`.gitignore`設定で十分です。システム環境によつては、リスト1.7のように設定するとさらに便利になることがあります。この`.gitignore`では、Railsドキュメントファイル、VimやEmacsのスワップファイル、そしてOS Xユーザーにはお馴染みの、あのいまいましい`.DS_Store`ディレクトリ(MacのFinder操作で作成される隠しディレクトリ)も管理対象から除外されます。この拡張版除外設定を使用したい場合は、`.gitignore`を好みのテキストエディタで開き、リスト1.7の内容で置き換えます。

リスト1.7 より多くのパターンを除外する`.gitignore`ファイル。

```
# Ignore bundler config  
.bundle  
  
# Ignore the default SQLite database.  
/db/*.sqlite3  
  
# Ignore all logfiles and tempfiles.  
/log/*.log  
/tmp  
  
# Ignore other unneeded files.  
doc/  
*.swp  
*~  
.project  
.DS_Store  
.idea
```

1.3.2 追加とコミット

最後に、新しく作成したRailsプロジェクトのファイルをGitに追加し、次にそれをコミットします。ファイルを追加する (`.gitignore` で指定されているものを除く) には、以下のコマンドを実行します。

```
$ git add .
```

ここで「`.`」は現在のディレクトリ(カレントディレクトリ)を指します。Gitは再帰的にファイルを追加できるので、自動的にすべてのサブディレクトリも追加されます。このコマンドによりプロジェクトのファイルは、コミット待ちの変更が格納されている「ステージングエリア」という一種の待機場所に追加されます。ステージングエリアにあるファイルのリストを表示するには、`status`コマンドを実行します²⁰。

```
$ git status
# On branch master
#
# Initial commit
#
# Changes to be committed:
#   (use "git rm --cached <file>..." to unstage)
#
#       new file: README.rdoc
#       new file: Rakefile
.
.
```

(出力結果が長いので、省略された部分を示すために縦点を使っています。)

変更を保存するには、`commit`コマンドを使います。

```
$ git commit -m "Initial commit"
[master (root-commit) df0a62f] Initial commit
42 files changed, 8461 insertions(+), 0 deletions(-)
create mode 100644 README.rdoc
create mode 100644 Rakefile
.
.
```

`-m` フラグは、コミットメッセージ(コミット内容の覚書)をその場で追加する場合に使用します。`-m`を省略した場合、[Section 1.3.1](#)で設定されたエディタが起動され、コミットメッセージの入力を求められます。

ここでコミットについて少し解説しておきます。Gitにおけるコミットは、あくまでローカルマシン上の操作であることに注意してください。この点について、もう一つの有名なオープンソースバージョン管理システムであるSubversionとははつきり異なります。Gitの場合、コミットを実行してもリモート上にあるリポジトリを直接変更することはありません。Gitでは、ローカルでの変更保存(`git commit`)と、リモート上のリポジトリへの変更反映(`git push`)の2段階に分かれています。[1.3.5](#)にはこのpushの例が記載されています。

ちなみに、`log`コマンドでコミットメッセージの履歴を参照できます。

```
$ git log
commit df0a62f3f091e53ffa799309b3e32c27b0b38eb4
```

```
Author: Michael Hartl <michael@michaelhartl.com>
Date: Thu Oct 15 11:36:21 2009 -0700
```

```
Initial commit
```

git logを終了するには**q**キーを押してください。

1.3.3 Gitのメリット

今の時点では、ソースコードをバージョン管理下に置かなければならぬ理由が今ひとつよくわからないという方がいるかもしれませんので、例を1つ紹介します(この後の章でも多くの実例を紹介します)。仮に、あなたが重要な**app/controllers**ディレクトリを削除してしまったとしましょう。

```
$ ls app/controllers/
application_controller.rb
$ rm -rf app/controllers/
$ ls app/controllers/
ls: app/controllers/: No such file or directory
```

ここでは、Unixコマンドの**ls**で**app/controllers**ディレクトリの中身を表示した後、**rm**コマンドをうっかり実行してこのディレクトリを削除してしまいました。**-rf**フラグは、「recursive」(サブディレクトリやその中のファイルもすべて削除する)、「force」(削除して良いかどうかをユーザーに確認しない)を指定するオプションです。

現在の状態を確認してみましょう。

```
$ git status
# On branch master
# Changed but not updated:
#   (use "git add/rm <file>..." to update what will be committed)
#   (use "git checkout -- <file>..." to discard changes in working directory)
#
#       deleted:    app/controllers/application_controller.rb
#
no changes added to commit (use "git add" and/or "git commit -a")
```

ファイルがいくつか削除されたが、この変更が行われたのは現在の「作業ツリー」のみなので、まだコミット(保存)されていません。つまり、以前のコミットを**checkout**コマンド(と、現在までの変更を強制的に上書きして元に戻すための**-f**フラグ)でチェックアウトすれば、簡単に削除前の状態に戻すことができます。

```
$ git checkout -f
$ git status
# On branch master
nothing to commit (working directory clean)
$ ls app/controllers/
application_controller.rb
```

削除されたディレクトリとファイルが無事復旧しました。これでひと安心です。

1.3.4 GitHub

Gitを使用してプロジェクトをバージョン管理下に置くことができたので、今度はGitHubにソースコードをアップロードしてみましょう。GitHubは、Gitリポジトリの置き場所を提供したり(ホスティング)、リポジトリを開発者同士で共有するサービスを提供するWebサー

ビスとして有名です。GitHubにわざわざリポジトリをプッシュするのには2つの理由があります。1つ目は、ソースコード(とそのすべての変更履歴)の完全なバックアップを作成することです。2つ目は、他の開発者との共同作業をより簡単に行うことです。GitHubへのプッシュは必須ではありませんが、GitHubのメンバーになっておくと、多くのオープンソースプロジェクトに参加できるようになります。

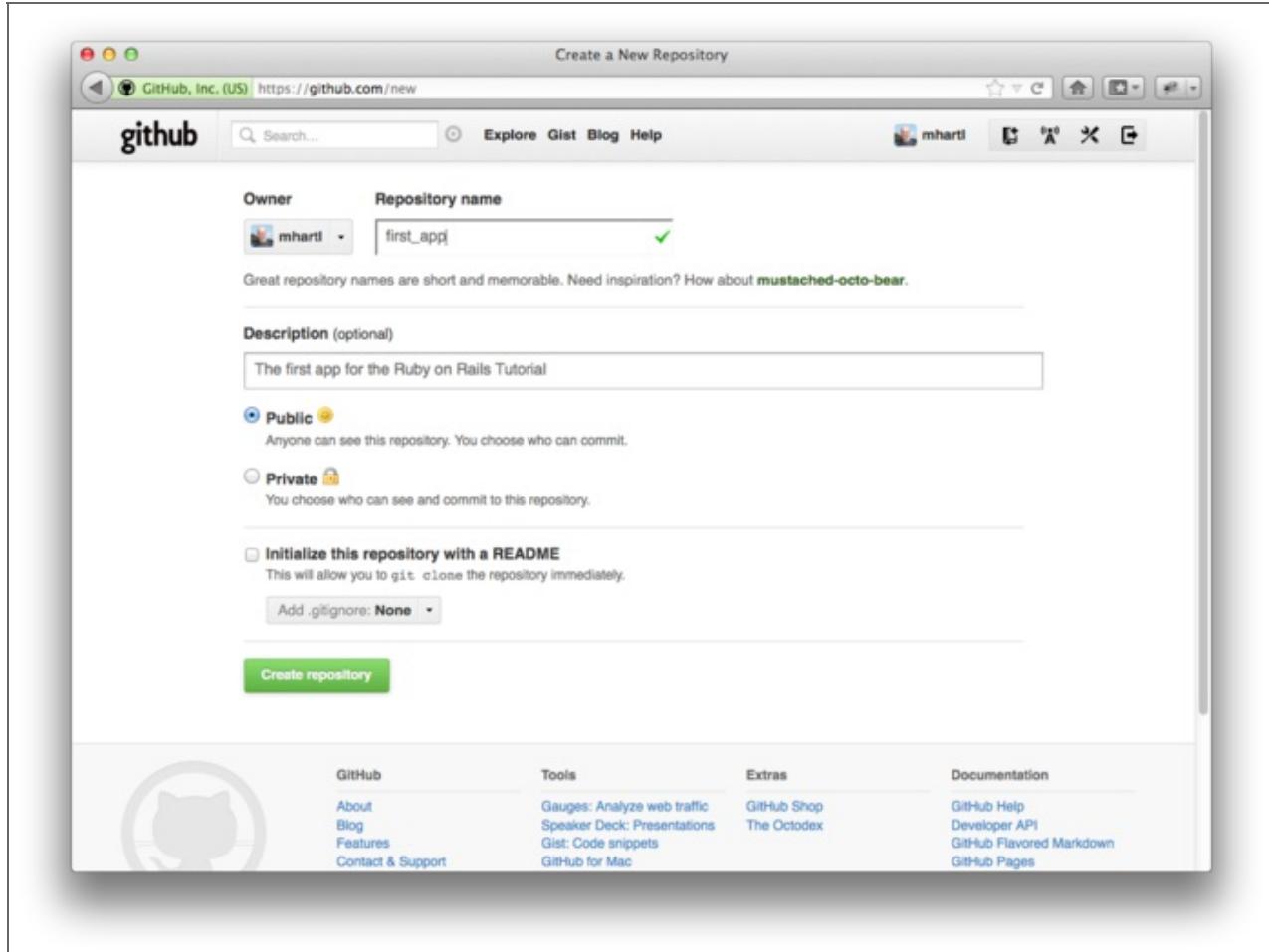


図1.6: アカウント作成直後のGitHubページ。(拡大)

GitHubにはさまざまな有料プランがありますが、オープンソースのコードなら無料で利用できるので、初めて利用するのであれば無料のGitHubアカウントを作成しましょう(念のため、GitHubのSSHキー作成方法のチュートリアルを先に読んでおいてください)。アカウント作成後、[Create repository]をクリックし、図1.6に従ってフォームに記入します。(注意:このときにREADMEファイルを使用してリポジトリを初期化しないでください。**rails new**コマンドを実行するときにこれらのファイルが作成されるからです。)リポジトリを作成したら、以下を実行してアプリケーションをプッシュします。

```
$ git remote add origin git@github.com:<username>/first_app.git  
$ git push -u origin master
```

最初のコマンドは、現在のメイン(*master*)ブランチ用の"origin"としてGitHubに追加します。次のコマンドで実際にGitHubにプッシュします(-uフラグについては気にする必要はありません。気になるのであれば"git set upstream"で検索してみてください)。もちろん、<username>はあなたの名前に置き換えてください。たとえば、私が**railstutorial**用に実行したコマンドは以下のとおりです。

```
$ git remote add origin git@github.com:railstutorial/first_app.git
```

このコマンドを実行すると、GitHubにファイル閲覧、コミット履歴の表示など多数の機能を備えたアプリケーションのリポジトリ用のページが作成されました(図1.7)。

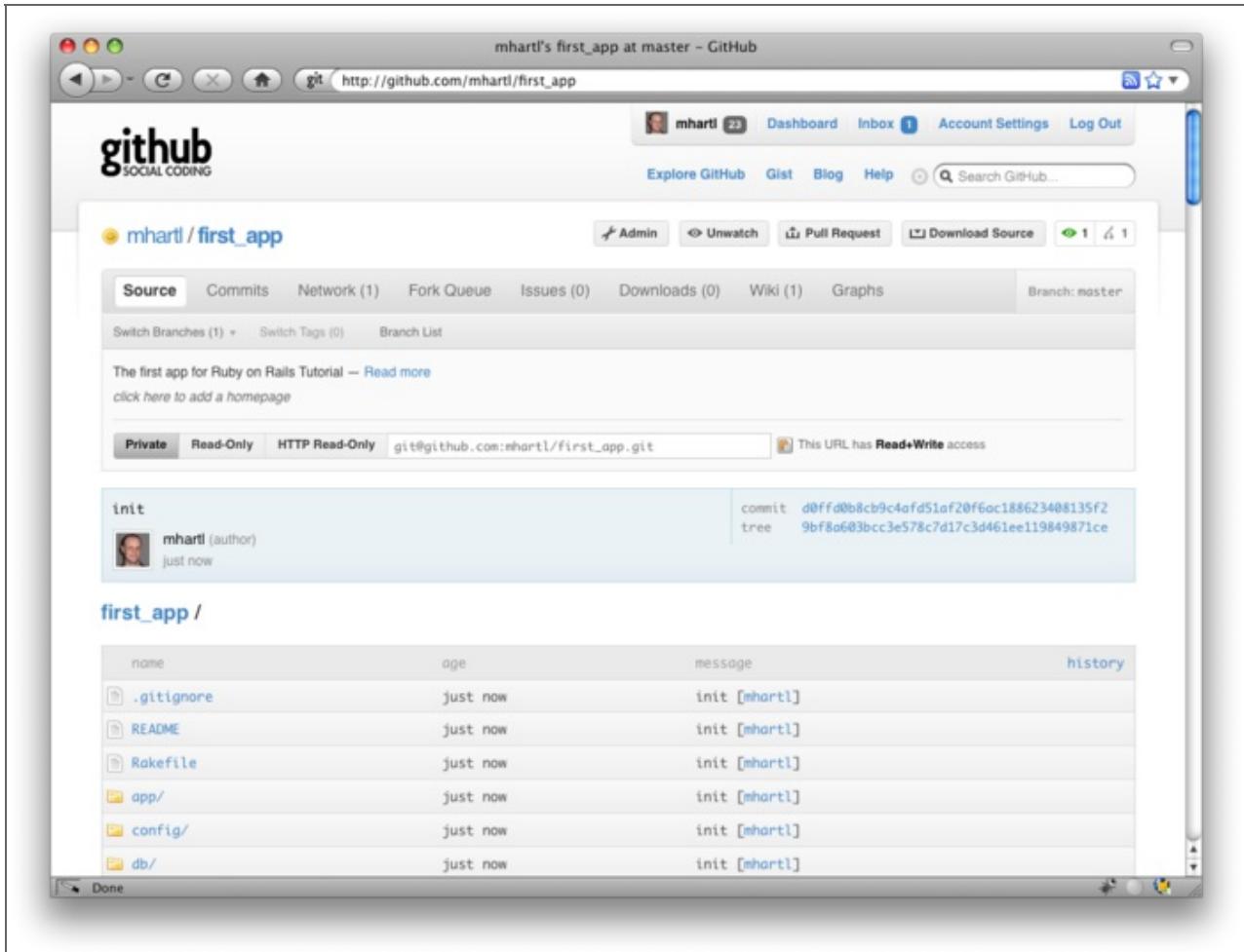


図1.7: GitHubのリポジトリページ。(拡大)

なお、GitHubにはコマンドラインインターフェイスを拡張したGUIアプリケーションもあります。GUIアプリケーションの方が好みであれば、GitHub for WindowsやGitHub for Macをチェックしてみてください(Linux用のGitHubは今のところGitしかないようです)。

1.3.5 ブランチ (branch)、変更 (edit)、コミット (commit)、マージ (merge)

1.3.4で紹介した手順を踏んでいれば、GitHubは自動的にリポジトリのメインページにREADMEファイルの内容を表示している事に気付くかもしれません。このチュートリアルでは、プロジェクトはrailsコマンドによって作成されたRailsアプリケーションなので、READMEファイルはRailsに既に含まれています(図1.8)。READMEファイルの拡張子は.rdocになっているので、GitHubでは適切なフォーマットで表示されます。しかしその内容はRailsフレームワークそのものに関するもので、そのままでは役に立ちません。この節ではREADMEの内容を編集し、プロジェクトに関する記述に置き換えます。それと同時に、Gitでbranch、edit、commit、mergeを行う際に私がお勧めしたいワークフローの実例をお見せします。

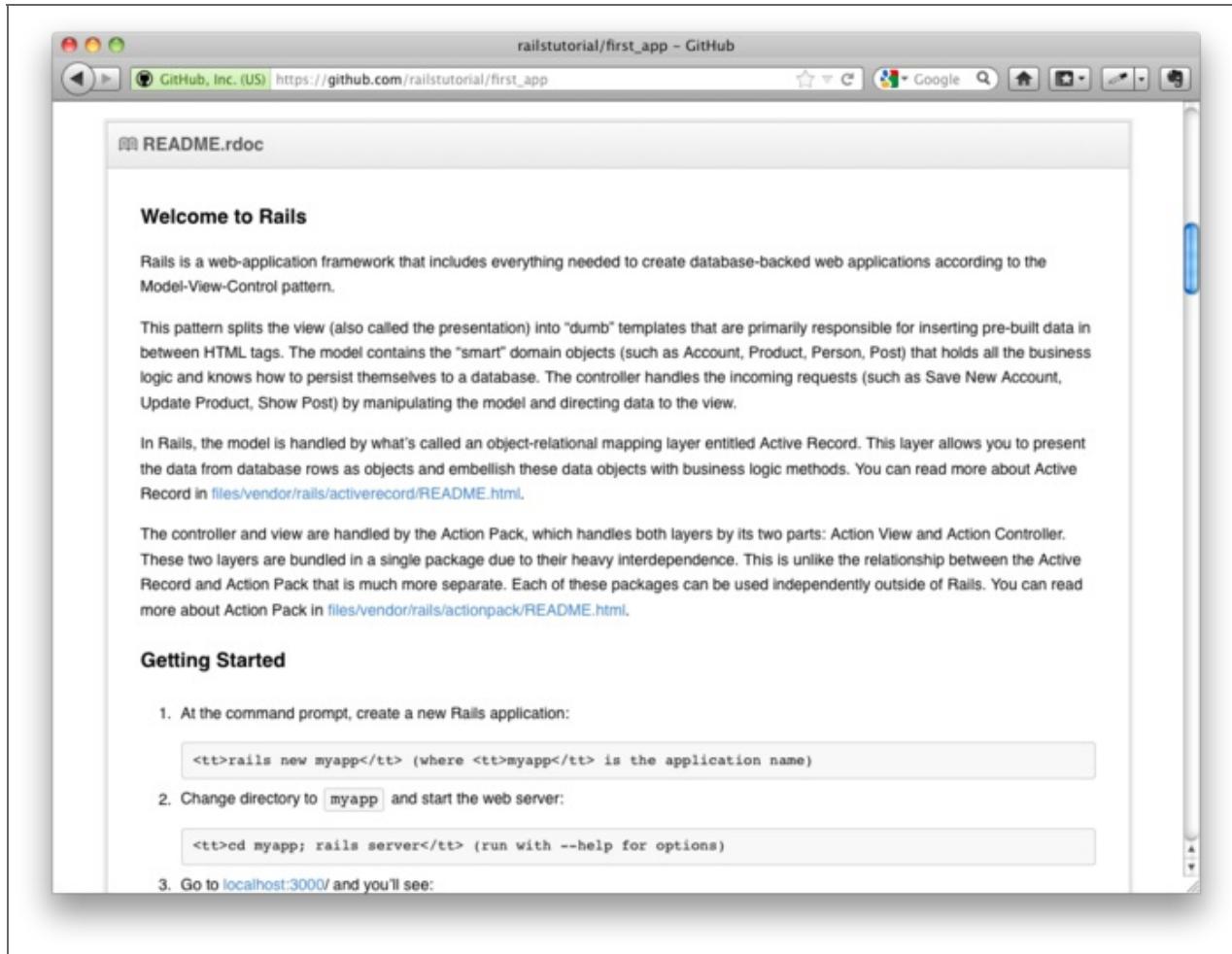


図1.8: GitHub上の、役に立たない初期のREADMEファイル(拡大)

ブランチ (branch)

Gitは、ブランチ (*branch*) をきわめて簡単かつ高速に作成することができます。ブランチは基本的にはリポジトリのコピーで、ブランチ上では元のファイルを触らずに新しいコードを書くなど、自由に変更や実験を試すことができます。通常、親リポジトリはマスター ブランチと呼ばれ、トピック ブランチ (短期間だけ使う一時的なブランチ) は **checkout** と **-b** フラグを使って作成できます。

```
$ git checkout -b modify-README
Switched to a new branch 'modify-README'
$ git branch
master
* modify-README
```

2つ目のコマンド (**git branch**) は、単にすべてのローカルブランチを一覧表示しているだけです。「*」はそのブランチが現在使用中であることを表します。1番目の**git checkout -b modify-README** コマンドで、ブランチの新規作成とそのブランチへの切り替えが同時に行われていることに注目してください。**modify-README** ブランチに*が付いていることで、このブランチが現在使用中であることが示されています (1.3で**co** エイリアスを設定した場合は、**git co -b modify-README** と入力することもできます)。

ブランチの真価は他の開発者と共同で作業する時に發揮されますが²¹、このチュートリアルのように一人で作業する時にも役立ちます。マスター ブランチはトピック ブランチで行つ

た変更に影響されないので、たとえブランチ上のコードがめちゃくちゃになってしまっても、マスター ブランチをチェックアウトしてトピック ブランチを削除すれば、いつでも変更を破棄することができます。具体的な方法についてはこの章の最後で説明します。

ちなみに、通常このような小さな変更のためにわざわざブランチを作成する必要はありませんが、ブランチがよい習慣であることに変わりはないので、少しでも練習しておきましょう。

変更 (Edit)

トピック ブランチを作成後、README の内容をわかりやすく書き換えてみましょう。私の場合、デフォルトの RDoc を編集するときには主に **Markdown** というマークアップ言語を使用しています。拡張子を **.md**にしておけば、GitHub にアップロードしたときに自動的にドキュメントがきれいに整形されます。今回は Git に付属する **mv** コマンド（注： Unix の mv コマンドではありません！）を使って README の拡張子を変更し、それから README の内容を **リスト 1.8** の内容に書き換えます。

```
$ git mv README.rdoc README.md  
$ subl README.md
```

リスト 1.8 新しい README ファイル (**README.md**)。

```
# Ruby on Rails Tutorial: first application  
  
This is the first application for  
[*Ruby on Rails Tutorial: Learn Rails by Example*](http://railstutorial.org/)  
by [Michael Hartl](http://michaelhartl.com/).
```

コミット (commit)

変更が終わったら、ブランチの状態を確認してみましょう。

```
$ git status  
# On branch modify-README  
# Changes to be committed:  
#   (use "git reset HEAD <file>..." to unstage)  
#  
#       renamed:    README.rdoc -> README.md  
#  
# Changed but not updated:  
#   (use "git add <file>..." to update what will be committed)  
#   (use "git checkout -- <file>..." to discard changes in working directory)  
#  
#       modified:   README.md  
#
```

この時点では、**1.3.2** のように **git add .** を実行することもできますが、Git には現存するすべてのファイル（git mv で作成したファイルも含む）への変更を一括でコミットする **-a** フラグがあります。このフラグは非常によく使われます。なお、**git mv** で作成されたファイルは、Git では新規ファイルではなく変更として扱われます。

```
$ git commit -a -m "Improve the README file"  
2 files changed, 5 insertions(+), 243 deletions(-)  
delete mode 100644 README.rdoc  
create mode 100644 README.md
```

-a フラグは慎重に扱ってください。最後のコミット後に新しいファイルを追加した場合は、まず **git add** を実行してバージョン管理下に置く必要があります。

コミットメッセージは現在形で書くようにしましょう。Gitのモデルは、(单一のパッチではなく)一連のパッチとしてコミットされます。そのため、コミットメッセージを書くときは、そのコミットが「何をしたのか」と履歴スタイルで書くよりも「何をする」ためのものなのかを書く方が、後から見返したときにわかりやすくなります。さらに、現在形で書いておけば、Gitコマンド自身によって生成されるコミットメッセージとも時制が整合します。詳細については GitHub に投稿された「[最新のコミット方法](#)」(英語) を参照してください。

マージ (merge)

ファイルの変更が終わったので、マスター ブランチにこの変更をマージ (*merge*) します。

```
$ git checkout master
Switched to branch 'master'
$ git merge modify-README
Updating 34f06b7..2c92bef
Fast forward
 README.rdoc      | 243 -----
 README.md        |    5 +
 2 files changed, 5 insertions(+), 243 deletions(-)
 delete mode 100644 README.rdoc
 create mode 100644 README.md
```

Gitの出力には **34f06b7** のような文字列が含まれていることがあります。Gitはこれらをリポジトリの内部処理に使用しています。この文字列は環境の違いにより上記のものと少し異なるかもしれません、他の部分はほぼ同じはずです。

変更をマージした後は、**git branch -d** を実行してトピック ブランチを削除すれば終わりです。

```
$ git branch -d modify-README
Deleted branch modify-README (was 2c92bef).
```

トピック ブランチの削除は必須ではありません。実際、トピック ブランチを削除せずにそのままにしておくことはよく行われています。トピック ブランチを削除せずに残しておけば、トピック ブランチとマスター ブランチを交互に行き来して、きりの良い所で変更をマージすることができます。

上で述べたように、**git branch -D** でトピック ブランチ上の変更を破棄することもできます。

```
# これはあくまで例です。ブランチでミスをした時以外は実行しないで下さい。
$ git checkout -b topic-branch
$ <really screw up the branch>
$ git add .
$ git commit -a -m "Major screw up"
$ git checkout master
$ git branch -D topic-branch
```

-d フラグと異なり、**-D** フラグは変更をマージしていなくてもブランチを削除してくれます。

プッシュ (push)

README ファイルの更新が終わったので、GitHubに変更をプッシュして結果を見てみましょう。既に 1.3.4 で一度プッシュを行ったので、大抵のシステムでは **git push** を実行するときに **origin master** を省略できます。

```
$ git push
```

先ほど説明したとおり、Markdownで記述された新しいファイルは GitHub できれいにフォーマットされます (図1.9)。



図1.9: Markdownを使用してフォーマットされた改良版**README** ファイル。(拡大)

1.4 展開する

この段階では空っぽのRailsアプリケーションしかありませんが、本番環境に展開(デプロイ: deploy)してしまいましょう。アプリケーションの展開は必須ではありませんが、頻繁に本番展開することによって、開発サイクルでの問題を早い段階で見つけることができます。開発環境のテストを繰り返すばかりで、いつまでも本番環境に展開しないままだと、アプリケーションを公開する時に思わぬ事態に遭遇する可能性が高まります²²。

かつてはRailsアプリの本番展開は大変な作業でしたが、ここ数年急速に簡単になってきており、さまざまな本番環境を選択できるようになりました。**Phusion Passenger** (ApacheやNginx²³ Webサーバ用のモジュール) を実行できるさまざまな共有ホストや仮想プライベートサーバ(VPS) の他に、フルサービスホスティングを提供する**Engine Yard**や**Rails Machine**、クラウドサービスを提供する**Engine Yard Cloud**や**Heroku**などがあります。

私のお気に入りはHerokuで、Railsを含むRuby Webアプリ用のホスティングプラットフォームです²⁴。Herokuは、ソースコードのバージョン管理にGitを使用していれば、Railsアプリケーションを簡単に本番環境に展開できます(Gitを導入したのは、まさにこのHerokuで使うためでもあります。まだGitをインストールしていなければ、1.3を参照してください)。この章では、私たちの最初のアプリケーションをHerokuに展開します。

1.4.1 Herokuのセットアップ

HerokuではPostgreSQLデータベースを使用します(ちなみに発音は“post-gres-cue-ell”で、よく“Postgres”と略されます)。そのためには、production(本番)環境にpg gemをインストールしてRailsがPostgreSQLと通信できるようにします。

```
group :production do
  gem 'pg', '0.12.2'
end
```

上のコードをリスト1.5の**Gemfile**に追加すると、リスト1.9のようになります。

リスト1.9 PostgreSQL用のpg gemを追加した**Gemfile**。

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

gem 'rails', '3.2.13'

group :development do
  gem 'sqlite3', '1.3.5'
end

# Gems used only for assets and not required
# in production environments by default.
group :assets do
  gem 'sass-rails',   '3.2.5'
  gem 'coffee-rails', '3.2.2'

  gem 'uglifier', '1.2.3'
end

gem 'jquery-rails', '2.0.2'

group :production do
  gem 'pg', '0.12.2'
end
```

インストールの際には、**bundle install**に特殊なフラグを追加します。

```
$ bundle install --without production
```

--without productionオプションを追加すると、production用のgem(この場合はpgのみ)はローカルの環境にはインストールされません。(今回追加したgemは本番環境でしか使用できないように制限されているので、このコマンドを実行しても、ローカルgemはローカルの環境にインストールされません。ただし**Gemfile.lock**については、Herokuがこれを使用して本番環境のRailsアプリケーションで必要なgemを推測するため、更新しておく必要があります。)

次にHerokuのアカウントを新規作成して設定します。最初にHerokuのユーザー登録を行います。アカウント作成後に完了通知メールが届いたら、Heroku Toolbeltを使用して必要なHerokuソフトウェアをインストールします²⁵。次にターミナルで以下の**heroku**コマンドを実行します(実行前にターミナルの終了と再起動が必要なことがあります)。

```
$ heroku login
```

最後に、元のRailsプロジェクトディレクトリに移動し、**heroku**コマンドを実行して、Herokuサーバー上にサンプルアプリケーション用の場所を作成します(リスト1.10)。

リスト1.10 Herokuに新しいアプリケーションを作成する。

```
$ cd ~/rails_projects/first_app
$ heroku create
Created http://stormy-cloud-5881.herokuapp.com/ |
git@heroku.com:stormy-cloud-5881.herokuapp.com
Git remote heroku added
```

この`heroku`コマンドを実行すると、私たちのRailsアプリケーション専用のサブドメインが作成され、ただちにブラウザで表示可能になります。今はまだ何もありませんが、すぐに展開してWebページを表示させましょう。

1.4.2 Herokuに展開する(1)

Railsアプリケーションを実際にHerokuに展開するには、まずGitを使用してHerokuにリポジトリをプッシュします。

```
$ git push heroku master
```

1.4.3 Herokuに展開する(2)

失礼、その2はありません。これで終わりです(図1.10)。展開されたアプリケーションの表示は、`heroku create`(リスト1.10)を実行した際に生成されたアドレスをブラウザで開くだけです(もちろんここに出てくる私のアドレスではなく、あなたのアドレスを使って下さい)。`heroku`コマンドに以下の引数を与えるだけで、正しいアドレスでブラウザが起動します。

```
$ heroku open
```

なお、設定の関係で、“About your application’s environment”リンクはHeroku上ではリンク切れになります。これは正常です。5.3.2ではこのデフォルトのRailsページは削除されるので、完全なサンプルアプリケーションではこのエラーは消え去ります。

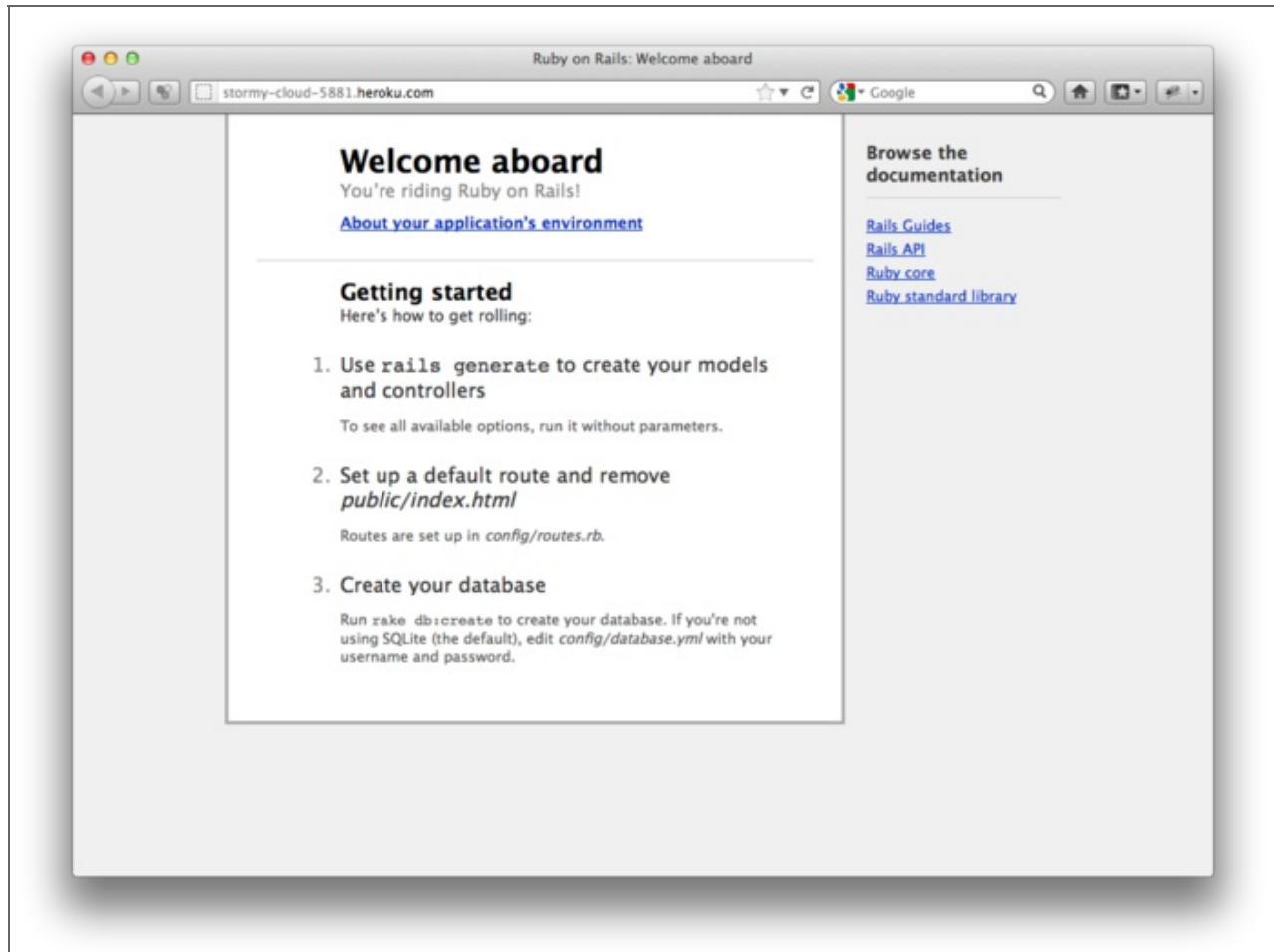


図1.10: Heroku上で動作しているRailsチュートリアルの最初のアプリケーション。(拡大)

Herokuへの展開が完了した後は、アプリケーションの管理と設定にこんな美しいユーザーインターフェースを使用できます(図1.11)。

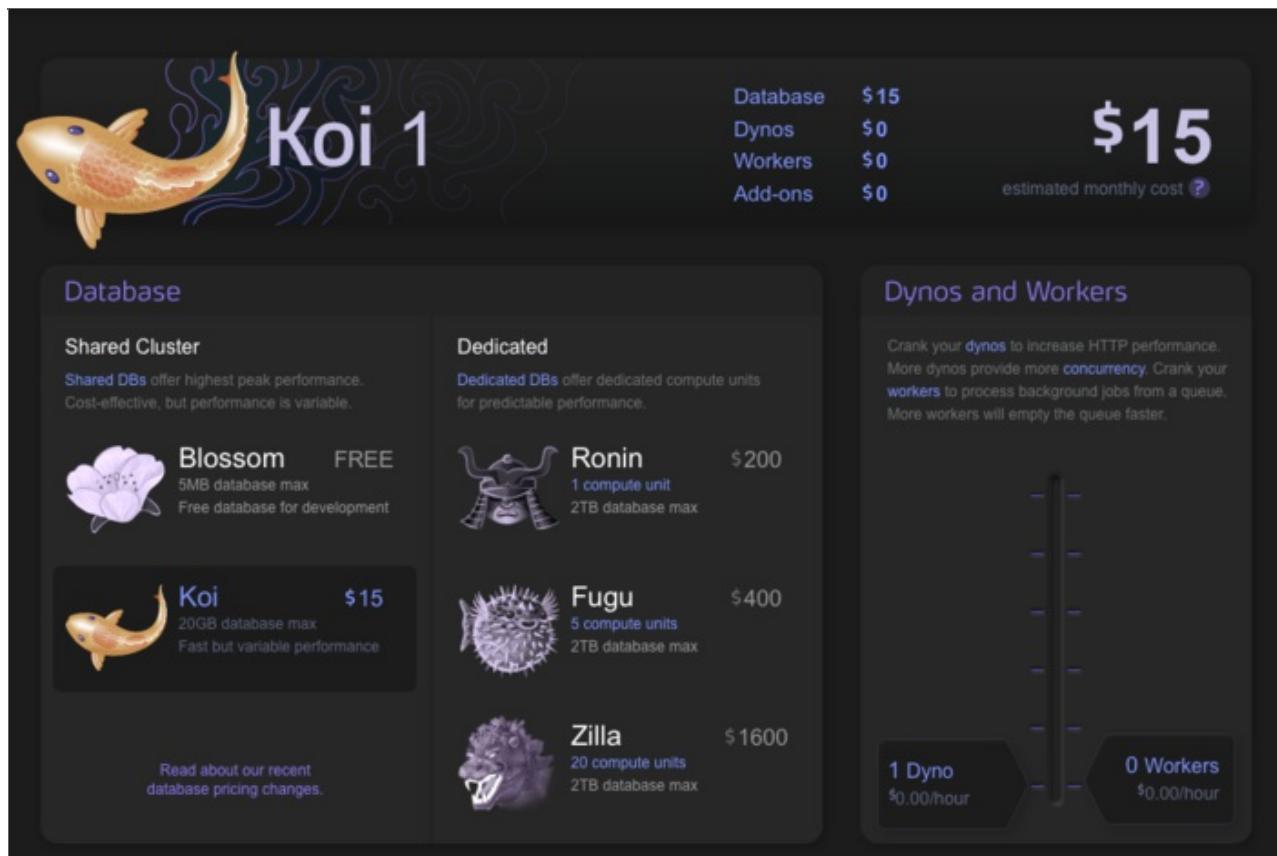


図1.11: 美しいHerokuのインターフェイス。(拡大)

1.4.4 Herokuコマンド

Herokuのコマンドはたくさんあるので、ここでは簡単に触れる程度にとどめますが、少しだけ使ってみましょう。アプリケーションの名前を変更してみます。

```
$ heroku rename railstutorial
```

注意: この名前は、私のサンプルアプリケーションで既に使用していますので、他の名前を使用してください。実際は、Herokuで生成されたデフォルトのアドレスでも十分です。本当にアプリケーションの名前を変えてみたい場合は、次のようなランダムなサブドメイン名を設定し、この章の冒頭で説明したアプリケーションのセキュリティを実装してみる方法もあります。

hwpcbmze.herokuapp.com
seyjhflo.herokuapp.com
jhyicevg.herokuapp.com

このようなでたらめのサブドメイン名なら、アドレスを渡さない限りサイトがアクセスされる心配もありません。(ちなみに、Rubyの威力の一端をお見せするために、ランダムなサブドメイン名を生成するためのコンパクトなコードを以下に記します。

```
('a'..'z').to_a.shuffle[0..7].join
```

最高ですね。)

Herokuでは、サブドメインの他に独自ドメインも使用できます。(実は、このRuby on RailsチュートリアルWebサイトもHeroku上にあるのです。あなたが本書をオンラインで読んでいるということは、まさにHerokuにホスティングされたWebサイトを見ているということになります。)カスタムドメインや他のHeroku関連の情報については、[Heroku ドキュメント](#)を参照してください。

1.5 最後に

この章ではインストール、開発環境の設定、バージョン管理、本番環境への展開など、多くの課題を達成しました。ここまで進歩をTwitterに投稿したりFacebookで通知するには以下のリンクからどうぞ。

[@railstutorialでRuby on Railsを学習中!http://railstutorial.org/](#)

後は、Railsを実際に勉強するだけです。一緒に頑張りましょう。

第2章 デモアプリケーション»

1. *URI*はUniform Resource Identifierの略です。それよりやや一般性の低い*URL*はUniform Resource Locatorの略です。URIは、要するに「ブラウザのアドレスバーにあるあれ」と考えればだいたい合っています。[↑](#)
2. <http://tryruby.org/> [↑](#)
3. <http://railsforzombies.org/> [↑](#)
4. <http://railstutorial.org/screencasts> [↑](#)
5. *Rails*チュートリアルを読んでいて、チュートリアル内部の別セクション番号へのリンクをクリックして移動したら、なるべくすぐに元の場所に戻ることをお勧めします。Webページで読んでいる場合は、ブラウザの [戻る] ボタンで戻れます。Adobe ReaderやOS XのプレビューでPDF版を読んでいる場合でも、同様に戻る方法があります。Adobe Readerの場合は、ドキュメント画面を右クリックして [Previous View] をクリックします。OS X Previewの場合はメニューの [移動] > [戻る] で戻れます。[↑](#)
6. **sudo**コマンドを実行するとデフォルトでroot(スーパーユーザー)に切り替わるためか、多くの人が**sudo**コマンドを "superuser do" の略だと誤って信じています。正しくは、**sudo**は**su**コマンドと英語の "do" をつなげたものです。そして**su**コマンドは "substitute user" (ユーザーの切替) の略なのです。ターミナルで**man su**と入力すればこのことを確認できます。[↑](#)
7. <http://railstutorial.org/help> [↑](#)
8. https://github.com/perfectionist/sample_project/wiki [↑](#)
9. viは、Unixで古くから使用されているコマンドベースの強力なエディタです。Vimは "vi improved" の略です。[↑](#)
10. https://github.com/mhartl/rails_tutorial_sublime_text [↑](#)
11. IRCが初めてであれば、まず“irc client <あなたのプラットフォーム>”で検索することをお勧めします。OS X用のネイティブクライアントとしてはColloquyとLimeChatがお勧めです。もちろんWeb版のインターフェイスもあります。<http://webchat.freenode.net/?channels=rvm> [↑](#)

12. 場合によっては、動作させるためにSubversionというバージョン管理システムをインストールしておく必要があるかもしれません。[↑](#)
13. <http://mxcl.github.com/homebrew/> [↑](#)
14. この手順が必要となるのは、Rails gemのバージョンを変更した場合に限られます。おそらくRailsインストーラを使用している場合にしかこういうことは起こらないでしょう。他の場合にこの手順を実行しても大丈夫です。[↑](#)
15. <https://developer.apple.com/downloads/> [↑](#)
16. Windowsユーザーの場合は代わりに**ruby rails server**と入力する必要がある場合があると[1.1.3](#)に書いてあったことを思い出してください。[↑](#)
17. 通常、Webサイトは80番ポートで受信待ちしますが、このポートを使用するには特別な権限が必要になることが多いので、Railsの開発用サーバーでは制限の少ない、番号の大きいポート(いわゆるハイナンバーポート)を使用します。[↑](#)
18. GUIエディタの起動後もターミナルを使用し続けることはできます。ただし、Gitはデタッチ時にコミットメッセージが空のままファイルを閉じたとみなすため、コミットは中断されます。Gitのエディタオプションで**subl**や**gvim**にフラグを付けないと、このあたりの動作で頭が混乱するかもしれません(誤注:gitのエディタ設定はGUIエディタとあまり相性がよくないらしく、vimやnanoのようなコマンドベースのエディタを選択するのが無難なようです)。この注釈の意味がよくわからない場合は、無視してくださいってよいです。[↑](#)
19. **.gitignore**がディレクトリに見当たらない場合は、ファイルブラウザやエクスプローラで隠しファイルを表示するよう設定を変更する必要があるかもしれません。[↑](#)
20. **git status**を実行したときに表示して欲しくないファイルがあれば、それらのファイルを[1.7](#)の**.gitignore**ファイルに追加してください。[↑](#)
21. 詳細については*Pro Git*の「Gitのブランチ機能」を参照してください。[↑](#)
22. Railsチュートリアルのサンプル・アプリケーションでは気にする必要はありません。作りかけの恥ずかしいWebアプリケーションをネットにうっかり公開してしまわないだろうかと心配する方も多いいらっしゃるかと思いますが、それを防ぐための方法はいくつもありますのでご安心ください。[1.4.4](#)はその方法の1つです。[↑](#)
23. “Engine X”と発音します。[↑](#)
24. Herokuは、Ruby WebプラットフォームにRackミドルウェアが導入されていれば動作します。このミドルウェアは、WebフレームワークとWebサーバーの標準インターフェイスを提供します。Rackインターフェイスを導入したことでのRubyコミュニティは大きな力を得ることができました。Rackはさまざまなフレームワークに導入されており、**Sinatra**、**Ramaze**、**Camping**、そしてRubyがRackを採用しています。つまり、RubyのWebアプリケーションは基本的にHerokuでサポートされているということです。[↑](#)
25. <https://toolbelt.heroku.com/> [↑](#)