everyday Rails

RSpecによるRailsテスト入門

テスト駆動開発の習得に向けた実践的アプローチ

Aaron Sumner 著

伊藤 淳一 訳

Everyday Rails - RSpec による Rails テスト入門

テスト駆動開発の習得に向けた実践的アプローチ

Aaron Sumner と Junichi Ito (伊藤淳一)

This book is for sale at http://leanpub.com/everydayrailsrspec-jp

この版は 2024-01-08 に発行されました。

本書は Leanpub の電子書籍です。Leanpub はリーンパブリッシングプロセスで著者や出版社を支援します。リーンパブリッシング は新しい出版スタイルです。軽量なツールを使って執筆中の電子書籍を出版し、読者のフィードバックをもらいながら魅力的な本に仕上がるまでピボットを繰り返すことができます。

Twitter でシェアしませんか？

本書に関するコメントをTwitter でシェアして Aaron Sumner と Junichi Ito (伊藤淳一) を応援してください！

本書のハッシュタグは #everydayrailsjp です。

本書に関するコメントを検索する場合は、次のリンクをクリックして下さい。Twitter のハッシュタグを使って検索できます。

#everydayrailsjp

Contents

この版のまえがき

> このまえがきは2017年の原著改訂時に作成されたものです。

改訂版の「Everyday Rails - RSpec による Rails テスト入門」を手にとっていただき、どうもありがとうございます。改訂版をリリースするまで、長い時間がかかりました。そして、内容も大きく変わりました。本書を読んだみなさんに「長い間待った甲斐があった」と思っていただけると幸いです。

なぜこんなに時間がかかったのでしょうか？前述のとおり、内容は大きく変わりました。本の内容そのものも変わりましたし、Rails における一般的なテストの考え方も変わっています。まず後者について説明しましょう。Rails 5.0の登場と同時に、Rails チームはコントローラのテストを事実上非推奨としました。個人的にこれは素晴らしいニュースでした。本書の前の版では説明に 3章 も使っていましたが、私も最初はコントローラのテストを理解するのに非常に苦労したのを思い出しました。そして、最近では以前ほどコントローラのテストを書かなくなりました。

その1年後、Rails 5.1がリリースされ、ついに高レベルのシステムテストが組み込まれました。このレベルのテストは本書を最初に出版したときから採用していたもので、かつては Rails に自分で組み込む必要がありました。システムテストは RSpec ではないので、本書で使っていたものとまったく同じではありません。ですが、Rails 標準の構成で開発したいと思う人たちが、アプリケーションを様々なレベルでテストできるようになったのは、とても素晴らしいことだと思います。

一方、RSpec の開発も進んでいます。RSpec にも数多くの新機能が実装され、より表現力豊かにテストを書けるようになりました。私自身を含め、多くの開発者が今なお RSpec を愛用しています。また、RSpec をアプリケーションに組み込むのに、いくつかの手順が必要になる点も変わっていません。

以上が自分ではコントロールできない、外部で起きた変化です。では次に、私が改訂版の「Everyday Rails - RSpec による Rails テスト入門」に加えた変更をご説明しましょう。きっと以前の版よりも充実した内容になっているはずです。今回の変更点の多くは、Rails や RSpec それぞれに起きた変化とは関係なく、私自身が「こうしたい」と思って加えた変更です。

本書はもともと「Everyday Rails[1]」というブログに書いていた記事から始まっています。5年前、私は Rails アプリケーションのテストの書き方について、自分が学んだことをブログに書き始めました。ブログ記事は人気を集め、私はそれを新たに書き下ろした内容や、完全なサンプルコードとともに一冊の本にまとめることにしました。その後、本書は私の期待をはるかに超えて多くの人たちに読まれ、私が初心者だった頃と同じようにテストの書き方で困っていた人たちを助けました。

それにしてもソフトウェアというのは面白いもので、その名の通り「ソフト（柔らかい）」です。対象となる問題を大小様々な観点から理解するにつれ、問題を解決するためのアプローチは少しずつ変わってきます。現在私が使っているテストのテクニックは、初期のブログ記事や本書の初版で書いたテクニックと根本的には同じです。しかし、私はこれまでにそのテクニックを増やし、テクニックを厳選し、さらにテクニックを磨き上げてきました。

この1年間で頭を悩ませたのは、どうやってテストにおける「次のレベルの学び方」を本書に落とし込むか、ということでした。つまり、初心者がテストを学び、それから自分自身のテクニックを増やし、厳選し、磨き上げる方法を考えるのに苦労しました。幸いなことに、本書でもともと採用していた学習フレームワークは、今でも正しかったようです。すなわち、最初は簡単なアプリケーションから始めて、それをブラウザでテストします（もしくは API であれば、最近は Postman のようなツールを使うこともあると思います）。それから、小さな単位でテストを書き始めます。最初は明白な仕様をテストします。それからもっと複雑なテストを書きます。今度はその順番をひっくり返します。まずテストを書き、それからコードを書くのです。こうやっていくうちに、効果的なテストの書き方が身に付いていきます。

そうは言うものの、私は前の版で使っていたサンプルアプリケーションに満足できていませんでした。とてもシンプルなアプリケーションなので、新しいテストテクニックを説明していても読者の頭からアプリの仕様が抜け落ちない点は良かったのですが、そのシンプルさゆえに意味のあるテストコードを追加しづらいのが難点でした。また、簡単な修正をコードに加えたいだけなのに、全部の章に渡って同じ修正を加えなければならず、バージョン管理システムで変更の衝突が頻繁に発生していたのも苦労した点の一つです。改訂版のサンプルアプリケーションは大きくなりましたが、それでも大きすぎるレベルではありません。これならずっと快適にテストを書くことができます。ちなみに、これは私が久々にテストファーストで 書かずに 作ったアプリケーションでもあります。

それはさておき、みなさんが本書を楽しみながら読んでくれることを願っています。テストを書いたことがないという方はもちろん、本書の初版から読んでくれている方で、テスト駆動開発に対する私の考え方やその他のテストテクニックがどのように変わってきたのか興

[1] https://everydayrails.com

味を持っている方も、楽しんで読んでもらえると嬉しいです。公開前に自分で何度も読み直し、問題が無いことを確認したつもりですが、もしかすると読者のみなさんは内容の誤りに気付いたり、別のもっと良い方法を知っていたりするかもしれません。間違いを見つけたり、何か良いアイデアがあったりする場合は、このリリース用の GitHub issues[2] にぜひ報告してください。できるだけ素早く対処します。(訳注: 日本語版のフィードバックはこちら[3]からお願いします)

改めてみなさんに感謝します。みなさんにこの改訂版を気に入ってもらえることを願っています。そして Github や Twitter、E メールでみなさんの感想が聞けることも楽しみにしています。

[2] https://github.com/everydayrails/everydayrails-rspec-2017/issues
[3] https://github.com/JunichiIto/everydayrails-rspec-jp-2024/issues

日本語版のまえがき

> このまえがきは日本語版の初版リリース時に作成されたものです。

私は好運です。

英語は私の母国語です。そして英語は技術文書の非公式な共通言語になっているようです。私は数多くの本やブログ、スクリーンキャストで勉強し、テスト駆動開発と RSpec を理解することができました。そしてついに、自分でその本を書くこともできました。私には想像することしかできませんが、世界中のソフトウェア開発者の多くは新しいプログラミング言語を学ぶだけでなく、関連する情報源が書かれている外国語もがんばって学ぶ必要があるんですよね。

そして、私はまたもや好運であり、大変嬉しく思っています。なぜかといえば、同じ Rubyist である伊藤淳一さん、魚振江さん、秋元利春さんが日本語で読みたがっているプログラマのために、*Everyday Rails Testing with RSpec* をがんばって翻訳してくれたからです。彼らは本書を新しい読者に届けてくれただけでなく、今後のバージョンの改善に役立つ貴重なフィードバックも返してくれました。

読者のみなさんが淳一さん、振江さん、利春さんの努力の成果を私と同じぐらい楽しんで、感謝することを願っています。そして、あなたの今後の Rails 開発にも好運が訪れることを願っています！

Aaron Sumner

Author

Everyday Rails Testing with RSpec

日本語版独自のアップデート内容について

本書は2017年11月に改訂された *Everyday Rails Testing with RSpec*[4] の内容をベースに、原著者の許可を得た上で日本語版独自のアップデートを加えたものです。具体的には以下の点が原著と異なります。

- サンプルアプリケーションを Rails 7.1で作り直している（原著は Rails 5.1）
- RSpec Rails 6.1を対象バージョンとして解説している（原著は RSpec Rails 3.6）
- フィーチャスペックの章（第6章）をはじめとして、フィーチャスペックで書かれていたテストをすべてシステムスペックで書き直している
- ファイルアップロード機能を Active Storage で実装している（原著は Paperclip gem）
- その他、2024年1月時点で最新の Rails や最新の gem の仕様に合わせて説明やサンプルコードを修正している

サンプルアプリケーションのソースコードも日本語版専用の GitHub リポジトリで公開しています。

https://github.com/JunichiIto/everydayrails-rspec-jp-2024

なお、2022年のアップデート以降、翻訳者が伊藤淳一、秋元利春、魚振江の3人体制から、伊藤淳一のみに変わっています。

[4] https://leanpub.com/everydayrailsrspec

1. イントロダクション

Ruby on Rails と自動テストは相性の良い組み合わせです。Rails にはデフォルトのテストフレームワークが付いてきます。ジェネレータを動かせば自動的にひな型となるテストファイルも作られるので、すぐに自分自身のテストコードを書き込むことができます。とはいえ、Rails でテストを全く書かずに開発する人や、書いたとしても大して役に立たない、もしくは書いてもほとんど意味のないスペックをちょこっと書いて終わらせるような人もたくさんいます。

これにはいくつかの理由があると私は考えています。人によっては Ruby や規約の厳しい web フレームワークを覚えることだけで精一杯になってしまい、そこへさらに新しい技術が増えるのは 余計な仕事 としか思えないのかもしれません。もしくは時間の制約が問題になっている可能性もあります。テストを書く時間が増えることによって、顧客や上司から要求されている機能に費やす時間が減ってしまうからです。もしくはブラウザのリンクをクリックするのが テスト であるという習慣から抜け出せなくなっているだけかもしれません。

私も同じでした。私は自分のことを正真正銘のエンジニアだとは思ったことはありませんが、解決すべき問題を持っているという点ではエンジニアと同じです。そしてたいていの場合、ソフトウェアを構築する中でそうした問題の解決策を見つけています。私は1995年から web アプリケーションを開発しており、予算の乏しい公共セクターのプロジェクトを長い間一人で担当しています。小さいころに BASIC をさわったり、大学で C++ をちょっとやったり、社会人になってから入った2社目の会社で役に立たない Java のトレーニングを一週間ほど受講したりしたことはありましたが、ソフトウェア開発のまともな教育というものは全く受けたことがありません。実際、私は2005年まで PHP で書かれたひどいスパゲティプログラム[5]をハックしていて、それからようやく web アプリケーションのもっと上手な開発方法を探し始めました。

私はかつて Ruby を触ったことはありましたが、真剣に使い始めたのは Rails が注目を集め出してからです。Ruby や Rails には学習しなければいけないことがたくさんありました。たとえば、新しい言語や アーキテクチャ 、よりオブジェクト指向らしいアプローチ等々です（Rails におけるオブジェクト指向を疑問に思う人がいるかもしれませんが、フレームワークを使っていなかったころに比べれば、私のコードはずっとオブジェクト指向らしくなりました）。このように新しいチャレンジはいくらか必要だったものの、それでもフレームワーク

[5] http://en.wikipedia.org/wiki/Spaghetti_code

を使わずに開発していた時代に比べると、ずっと短い時間で複雑なアプリケーションが作れました。こうして私は夢中になったのです。

とはいえ、Rails に関する初期の書籍やチュートリアルは、テストのような良いプラクティスよりも開発スピード（15分でブログアプリケーションを作る！）にフォーカスしていました。テストの説明は全くなかったか、説明されていたとしても、たいてい最後の方に一章だけしか用意されてませんでした。最近の書籍や web 上の情報源ではその欠点に対処しており、アプリケーション全体をテストする方法を初めから説明しているように思います。加えて、テストについて 専門的に 書かれた本はたくさんありますが、テストの実践方法をしっかり身につけていないと開発者（特にかつての私と同じような立場の開発者）の多くは一貫したテスト戦略を考えられないかもしれません。もしテストが多少あったとしても、そのようなテストは信頼できるものではなかったり、あまり意味のないテストだったりします。そんなテストでは 自信をもって開発する ことはできません。

本書の第一のゴールは 私の 役に立っている一貫した戦略をあなたに伝えることです。そしてその戦略を使って、あなたも 一貫したテスト戦略をとれるように願っています。私の戦略が正しく、本書でそれをうまく伝えることができれば、あなたは 自信をもってテストが書けるようになります。 そうすればコードに変更を加えるのも簡単になります。なぜなら、テストコードがアプリケーションをしっかり守り、何かがおかしくなったらあなたにすぐ知らせてくれるからです。

なぜ RSpec なのか？

誤解がないように言っておきますが、私は Ruby 用の他のテストフレームワークを悪く言うつもりはありません。実際、単体の Ruby ライブラリを書くときは MiniTest をよく使っています。しかし、Rails アプリケーションを開発し、テストするときは RSpec を使い続けています。

私にはコピーライティングとソフトウェア開発のバックグラウンドがあるせいかもしれませんが、RSpec を使うと読みやすいスペックが簡単に書けます。これが私にとって一番の決め手でした。のちほど本書でお話ししますが、技術者ではなくても大半の人々が RSpec で書かれたスペックを読み、その内容を理解できたのです。RSpec を使って自分のソフトウェアの期待する振る舞いを記述することは、もはや私の習慣になってしまったと言ってよいでしょう。RSpec の構文はスラスラと私の指先から流れ出してきます。そして、将来何か変更を加えたくなったときでも、相変わらず読みやすいです。

本書の第二のゴールは日常的によく使う RSpec の機能と構文をあなたが使いこなせるよう

に手助けすることです。RSpec は複雑なフレームワークです。しかし、多くの複雑なシステムがそうであるように、8割の作業は2割の機能で済ませられるはずです。なので、本書は RSpec や Capybara のような周辺ライブラリの完全ガイドにはなっていません。そのかわり、Rails アプリケーションをテストする際に私が何年にもわたって使ってきたツールに焦点を絞って説明しています。また、本書ではよくありがちなパターンも説明します。本書では具体的に説明していない問題に遭遇したときも、あなたがこのパターンをちゃんと理解していれば、長時間ハマることなく、すぐに解決策をみつけることができるはずです。

対象となる読者

もし Rails があなたにとって初めての web アプリケーションだったり、これまでのプログラマ人生でテストの経験がそれほどなかったりするなら、本書はきっと良い入門書になると思います。もし、あなたが 全くの Rails 初心者なら、この「Everyday Rails - RSpec による Rails テスト入門」を読む前に、Michael Hartl の Rails チュートリアル や、Daniel Kehoe の *Learn Ruby on Rails* 、もしくは Sam Ruby の Rails によるアジャイル Web アプリケーション開発 といった書籍で Rails の開発や基礎的なテスト方法を学習しておいた方が良いかもしれません。なぜなら、本書はあなたがすでに Rails の基礎知識を身につけていることを前提としているからです。言い換えると、本書では Rails の使い方は説明しません。また、Rails に組みこまれているテストツールを最初から紹介することもしません。そうではなく、RSpec といくつかの追加ライブラリをインストールします。追加ライブラリはテストのプロセスをできるだけ理解しやすく、そして管理しやすくするために使います。というわけで、もしあなたが Rails 初心者なのであれば、まず先ほどの資料を読み、それから本書に戻ってきてください。

本書の巻末にある「Rails のテストに関するさらなる情報源」も参考にしてください。このページではここで紹介した資料や、その他の書籍、Web サイト、テスト関連のチュートリアルのリンクを載せています。

もしあなたが Rails の開発経験は多少あるものの、テストにはまだ馴染めていない開発者なのであれば、まさに本書は最適です！私もかつてはずっとあなたと同じでしたが、私は本書で紹介するようなテクニックでテストカバレッジを向上させ、テスト駆動開発者らしい考え方を身につけることができました。あなたも私と同じようにこうしたテクニックを身につけてくれることを私は願っています。

ところで、本書で前提としている読者の知識や経験を具体的に挙げると、次のようになります。

- Rails で使われているサーバーサイドの Model-View-Controller 規約
- gem の依存関係を管理する Bundler
- Rails コマンドの実行方法
- リポジトリのブランチを切り替えられるぐらいの Git 知識

もしあなたが Test::Unit や MiniTest、もしくは RSpec そのものに慣れていて、自信をもって開発できるワークフローを確立している場合、本書を読むことでテストのアプローチを多少改善できるかもしれません。私なりのテストのアプローチから、単にコードをテストするだけでなく、意図をもってテストする方法を学んでいただければ、と思います。

本書はテスト理論に関する本 ではありません。 また、長年使われてきたソフトウェアにありがちなパフォーマンス問題を深く掘り下げるわけでもありません。本書を読むよりも、他の書籍を読んだ方が役立つかもしれません。本書の巻末にある「Rails のテストに関するさらなる情報源」を参考にしてください。このページではここで紹介した資料や、その他の書籍、Web サイト、テスト関連のチュートリアルのリンクを載せています。

私が考えるテストの原則

どういった種類のテストが一番良いのですか？単体テストですか？それとも統合テストですか？私はテスト駆動開発（TDD）を練習すべきでしょうか？それとも振る舞い駆動開発（BDD）を練習すべきですか？（そして両者の違いは何ですか？）私はコードを書く前にテストを書くべきでしょうか？それとも、コードのあとに書くべきでしょうか？そもそもテストを書くのをサボってもいいのでしょうか？

Rails をテストする 正しい 方法というテーマで議論をすると、プログラマの間で大げんかが始まるかもしれません。まあ、Mac 対 PC や、Vim 対 Emacs のような論争ほど激しくないとは思いますが、それでも Rubyist の中で不穏な空気が流れそうです。実際、David Heinmeier-Hansen は Railsconf 2014でTDD は「死んだ」と発言し[6]、Rails のテストについて近年新たな議論を巻き起こしました。

確かにテストの正しい方法は存在します。しかし私に言わせれば、テストに関してはその 正しさ の度合いが異なるだけです。

私のアプローチでは次のような基本的な信条に焦点を当てています。

- テストは信頼できるものであること
- テストは簡単に書けること

[6] http://david.heinemeierhansson.com/2014/tdd-is-dead-long-live-testing.html

- テストは簡単に理解できること（今日も将来も）

つまり、テストはあなたに開発者としての 自信 を付けさせるものであるべきなのです。この三つの要素を意識しながら実践すれば、アプリケーションにしっかりしたテストスイートができあがっていきます。もちろん、あなたが本物のテスト駆動開発者に近づいていくことは言うまでもありません。

一方、それと引き替えに失うものもあります。具体的には次のようなことです。

- スピードは重視しません。ただし、これについてはのちほど説明します。
- テストの中では過度に DRY なコードを目指しません。なぜならテストにおいては DRY でないコードは必ずしも悪とは限らないからです。この点ものちほど説明します。

とはいえ結局、一番大事なことは テストが存在すること です。信頼性が高く、理解しやすいテストが書いてあることが大事な出発点になります。何から何まで完璧である必要はありません。かつて私はたくさんアプリケーション側のコードを書き、ブラウザをあちこちクリックすることで “テスト” し、うまく動くことを祈っていました。しかし、前述したアプローチによって、こうした問題をついに乗り越えることができました。完全に自動化されたテストスイートを利用すれば、開発を加速させ、潜在的なバグや境界値に潜む問題をあぶり出すことができるのです。

そして、このアプローチこそがこれから本書で説明していく内容です。

本書の構成

本書「Everyday Rails - RSpec による Rails テスト入門」では、標準的な Rails アプリケーションが全くテストされていない状態から、RSpec を使ってきちんとテストされるまでを順に説明していきます。本書では Rails 7.1と RSpec Rails 6.1（RSpec 本体のバージョンは3.12）を使用します。これはどちらも執筆時点の現行バージョンです。

本書は次のようなテーマに分けられています。

- あなたが今読んでいるのが第1章 イントロダクション です。
- 第2章 RSpec のセットアップ では、新規、もしくは既存の Rails アプリケーションで RSpec が使えるようにセットアップします。
- 第3章 モデルスペック では、シンプルでも信頼性の高い単体テストを通じてモデルをテストしていきます。
- 第4章 意味のあるテストデータの作成 では、テストデータを作成するテクニックを説明します。

- 第5章 コントローラスペック では、コントローラに対して直接テストを書いていきます。
- 第6章 システムスペックで UI をテストする では、システムスペックを使った統合テストを説明します。統合テストを使えば、アプリケーション内の異なるパーツがお互いにきちんとやりとりできることをテストできます。
- 第7章 リクエストスペックで API をテストする では、昔ながらの UI を使わずに直接 API をテストする方法を説明します。
- 第8章 スペックを DRY に保つ では、いつどのようにしてテストの重複を減らすのか、そしていつ、そのままにすべきなのかを議論します。
- 第9章 速くテストを書き、速いテストを書く では、効率的にテストを書くテクニックと、素早いフィードバックを得るために実行対象のテストを絞り込む方法を説明します。
- 第10章 その他のテスト ではメール送信やファイルアップロード、外部の Web サービスといった、これまでに説明してこなかった機能のテストについて説明します。
- 第11章 テスト駆動開発に向けて ではステップ・バイ・ステップ形式でテスト駆動開発の実践方法をデモンストレーションします。
- そして、第12章 最後のアドバイス で、これまで説明してきた内容を全部まとめます。

各章にはステップ・バイ・ステップ形式の説明を取り入れています。これは私が自分自身のソフトウェアでテストスキルを上達させたのと同じ手順になっています。また、多くの章では どう テストし、なぜ テストするのかをしっかり考えてもらうための Q&A セクションで締めくくり、そのあとに演習問題が続きます。この演習問題はその章で習ったテクニックを自分で使ってみるために用意しています。繰り返しますが、あなた自身のアプリケーションでこうした演習問題に取り組んでみることを私は強く推奨します。一つはチュートリアルの内容を復習するためで、もう一つはあなたが学んだことをあなた自身の状況で応用するためです。本書では一緒にアプリケーションを作っていくのではなく、単にコードのパターンやテクニックを掘り下げていくだけです。ここで学んだテクニックを使い、あなた自身のプロジェクトを改善させましょう！

サンプルコードのダウンロード

本書のサンプルコードは GitHub にあります。このアプリケーションではテストコードも完全に書かれています。

ソースを入手しましょう！

https://github.com/JunichiIto/everydayrails-rspec-jp-2024

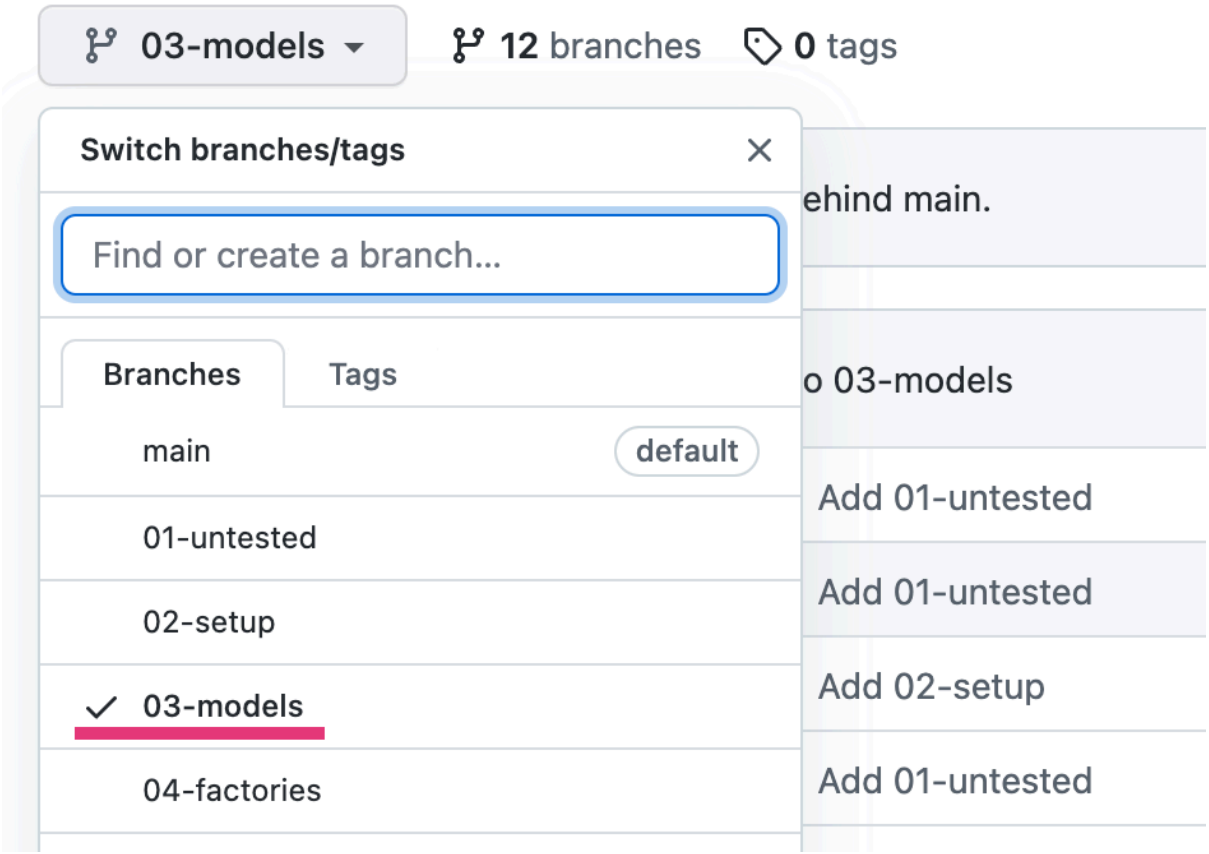
> 2022年に本書のアップデートを行ったタイミングで、日本語版は独自のサンプルコードを提供することになりました。原著のソースコードを参照したい場合は https://github.com/everydayrails/everydayrails-rspec-2017 にアクセスしてください。

もし Git の扱いに慣れているなら（Rails 開発者ならきっと大丈夫なはず）、サンプルコードをあなたのコンピュータにクローン（clone）することもできます。各章の成果物はそれぞれブランチを分けています。各章のソースを開くと完成後のコードが見られます。本書を読みながら実際に手を動かす場合は、一つ前の章のソースを開くと良いでしょう。各ブランチには章番号を振ってあります。各章の最初でチェックアウトすべきブランチをお伝えし、現在の章と一つ前の章で発生する変更点を確認できるリンクを紹介します。

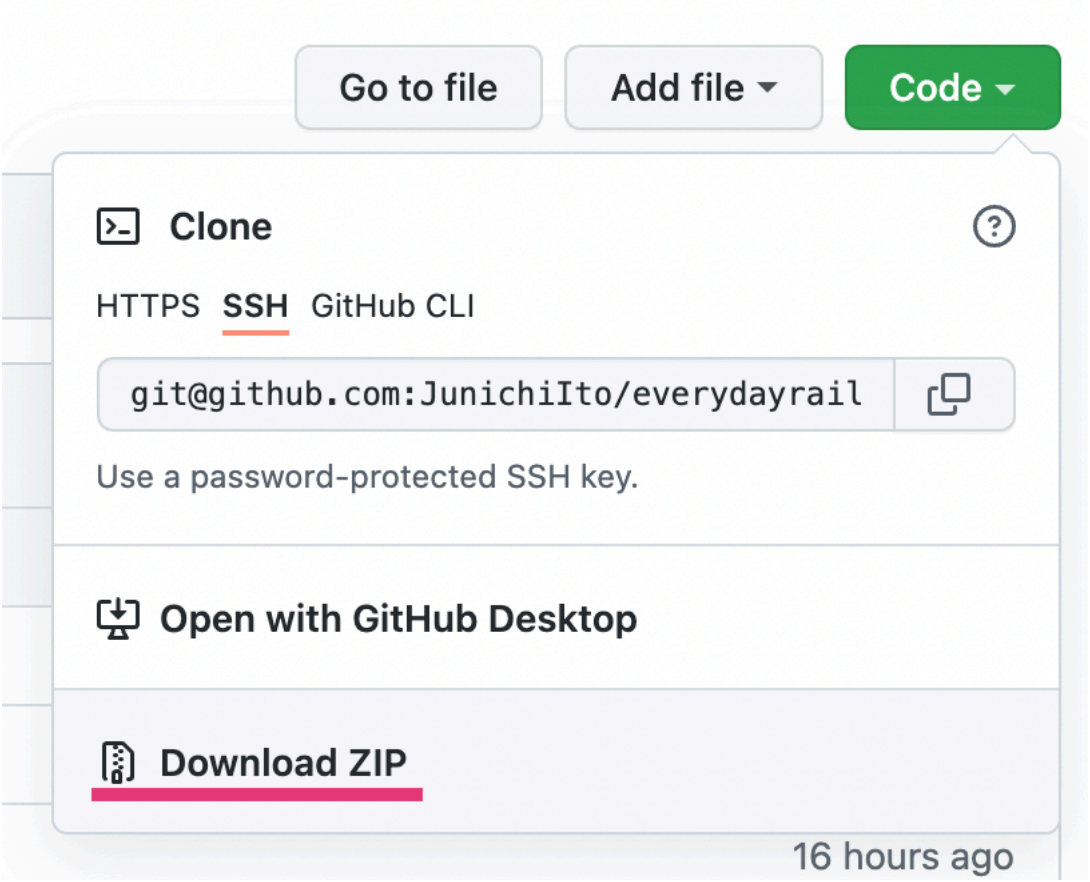
本書はどの章も一つ前の章のソースコードに変更を加えていく流れで構成されています。なので、現在の章のスタート地点として一つ前の章を使うことができます。たとえば、第5章のコードを最初から順に入力していきたいのであれば、第4章のコードから書き始めてください。

```
$ git checkout -b my-05-controllers origin/04-factories
```

Git の扱いに慣れていなくても各章のサンプルコードをダウンロードすることは可能です。まず GitHub のプロジェクトページを開いてください。それからブランチセレクタでその章のブランチを選択します。



最後に ZIP ダウンロードリンクをクリックします。クリックするとソースコードをコンピュータに保存できます。



Git Immersion[7] は実際に手を動かしながら Git のコマンドライン操作の基本を学ぶことができる素晴らしいサイトです。Try Git[8] もそのようなサイトです。

コードの方針

このアプリケーションは次の環境で動作します。

- **Rails 7.1**: 最新バージョンの Rails が本書のメインターゲットです。私が知る限り、Rails 5.1以上であれば本書で紹介しているテクニックは適用できるはずです。サンプルコードによっては違いが出るかもしれませんが、差異が出そうな箇所はできる限り伝えていきます。
- **Ruby 3.3**: Rails 7.1では Ruby 2.7以上が必須です。本書では執筆時点の最新バージョンである Ruby 3.3を使用します。

[7] http://gitimmersion.com/
[8] http://try.github.io

- **RSpec Rails 6.1と RSpec 3.12:** RSpec は Rails 専用の機能を提供する RSpec Rails（rspec-rails）と、RSpec の本体である RSpec（rspec-core）がそれぞれ独立した gem としてリリースされています。どちらも本書執筆時点の最新バージョンです。以前は RSpec と RSpec Rails はバージョン番号を統一してリリースされていましたが、RSpec Rails 4.0からバージョン番号は別々に更新されるようになりました。

本書で使用しているバージョン固有の用法等があれば、できる限り伝えていきます。もし Rails、RSpec、Ruby のどれかで古いバージョンを使っているなら、本書の以前の版をダウンロードしてください。以前の版は Leanpub からダウンロードできます。個々の機能はきれいに対応しませんが、バージョン違いに起因する基本的な差異は理解できるもしれません。

もう一度言いますが、本書はよくありがちなチュートリアルではありません！ 本書に載せているコードはアプリケーションをゼロから順番に作ることを想定していません。本書ではテストのパターンと習慣を学習し、あなた自身の Rails アプリケーションに適用してもらうことを想定しています。言いかえると、コードをコピー&ペーストすることができるとはいえ、そんなことをしても全くあなたのためにならないということです。あなたはこのような学習方法を Zed Shaw のLearn Code the Hard Way シリーズ[9]で知っているかもしれません。

「Everyday Rails - RSpec による Rails テスト入門」はそれと全く同じスタイルではありませんが、私は Zed の考え方に同意しています。すなわち、何か学びたいものがあるときは Stack Overflow や電子書籍からコピー&ペーストするのではなく、自分でコードをタイピングした方が良い、ということです。

間違いを見つけた場合

私はたくさんの時間と労力をつぎ込んで「Everyday Rails - RSpec による Rails テスト入門」をできる限り間違いのない本にしようとしてきましたが、私が見落とした間違いにあなたは気付くかもしれません。そんなときは GitHub の issues ページで間違いを報告したり詳細を尋ねたりしてください。https://github.com/JunichiIto/everydayrails-rspec-jp-2024/issues（訳注: この URL は日本語版専用の issues ページなので、日本語で質問できます。ただし、「サンプルアプリケーションがうまく動かない」といった技術的な質問はこの issues ページではなく、Teratail[10] のようなプログラマ向け Q&A サイトで質問してもらえると助かります）

[9] http://learncodethehardway.org/
[10] https://teratail.com/

gem のバージョンに関する注意点

本書と本書のサンプルアプリケーションで使用している gem のバージョンは、この RSpec Rails 6.1/Rails 7.1版を執筆していたとき（2024年1月）の現行バージョンです。当然、どの gem も頻繁にアップデートされますので、Rubygems.org や GitHub、またはお気に入りの Ruby 新着情報フィードでアップデート情報をチェックしてください。

サンプルアプリケーションについて

本書で使用するサンプルアプリケーションはプロジェクト管理アプリです。Trello や Basecamp ほど多機能でカッコいいものではありませんが、テストを書き始めるためには十分な機能を備えています。

はじめに、このアプリケーションは次のような機能を持っています。

- ユーザーはプロジェクトを追加できる。追加したプロジェクトはそのユーザーにだけ見える。
- ユーザーはタスクとメモと添付ファイルをプロジェクトに追加できる。
- ユーザーはタスクを完了済みにできる。
- 開発者はパブリック API を使って、外部のクライアントアプリケーションを開発できる。

私はここまで意図的に Rails のデフォルトのジェネレータだけを使ってアプリケーション全体を作成しました（*01_untested*[11] ブランチにあるサンプルコードを参照）。つまりこれは *test* ディレクトリに何も変更していないテストファイルとフィクスチャがそのまま格納されているということです。この時点で `bin/rails test` を実行すると、いくつかのテストはそのままでパスするかもしれません。しかし、本書は RSpec の本ですので、test フォルダは用無しになります。RSpec が使えるように Rails をセットアップし、信頼できるテストスイートを構築していきましょう。これが今から私たちが順を追って見ていく内容です。

まず最初にすべきことは、RSpec を使うようにアプリケーションの設定を変更することです。では始めましょう！

サンプルアプリケーションのセットアップ手順

[11] https://github.com/JunichiIto/everydayrails-rspec-jp-2024/tree/01-untested

> この項は日本語版独自の補足説明です。

サンプルアプリケーションを実際に動かす場合は以下の手順でセットアップしてください。

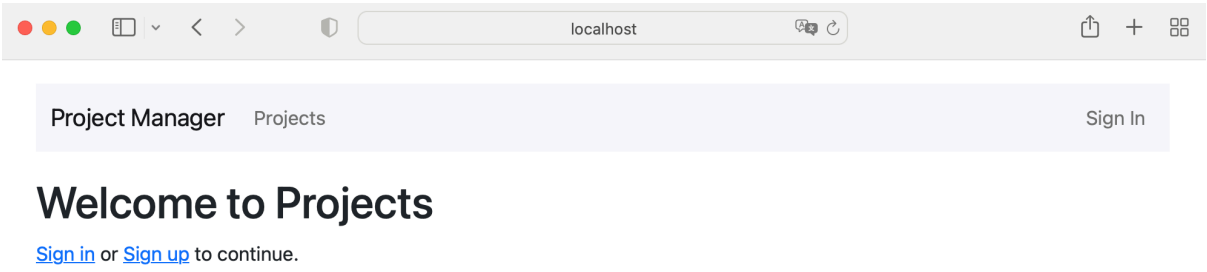
まず、サンプルアプリケーションには以下のツールやソフトウェアが必要です。不足している場合は適宜インストールしてください。

- Ruby 3.3.0（ただし、Ruby 3.0.0以上であれば動作することを確認しています）
- Git
- Google Chrome（執筆時点のバージョンは120）

インストールが済んだらターミナル上で次のコマンドを入力してセットアップします。

```
# ソースコードのダウンロード
git clone https://github.com/JunichiIto/everydayrails-rspec-jp-2024.git

# ディレクトリの移動
cd everydayrails-rspec-jp-2024

# 使用する Ruby バージョンを指定（本書では3.3.0を推奨。下記コマンドは rbenv を使用する場合）
rbenv local 3.3.0

# gem のインストールやデータベースのセットアップ等
bin/setup

# サーバーの起動
bin/rails s
```

`Listening on http://127.0.0.1:3000` のような表示がターミナルに表示されれば OK です。http://localhost:3000 にブラウザでアクセスするとホームページが表示されます。



“Sign up” のリンクを開くとユーザー登録ができ、サンプルアプリケーションを使えるようになります。サーバーを停止する場合は Ctrl-C で停止できます。

上記の手順でセットアップした場合は現在のブランチが main ブランチになっているはずです。以下のコマンドを実行して既存のテストがすべてパスすることも確認しておきましょう。

```
bundle exec rspec
```

以下のような表示で終了していれば RSpec も正常に動作しています。

```
Finished in 2.88 seconds (files took 2.39 seconds to load)
70 examples, 0 failures
```

サーバーが起動しない、テストが正常に動作しない、といった技術的な質問は teratail[12] のようなプログラマ向け Q&A サイトで質問してください。それでも問題が解決しない場合はGitHub の issues ページ[13]で質問していただいて構いません。

[12] https://teratail.com/
[13] https://github.com/JunichiIto/everydayrails-rspec-jp-2024/issues

2. RSpec のセットアップ

第1章で述べたように、プロジェクト管理アプリケーションは 動いています 。少なくとも 動いている とは言えるのですが、その根拠は何かと言われたら、リンクをクリックし、ダミーのアカウントとプロジェクトをいくつか作り、ブラウザを使ってデータを追加したり編集したりできた、ということだけです。もちろん、機能を追加するたびに毎回こんなやり方を繰り返しているといつか破綻します。このアプリケーションへ新しい機能を追加する前に、私たちはいったん手を止め、RSpec を使って 自動化されたテストスイート を追加する必要があります。私たちはここから先のいくつかの章にわたってこのアプリケーションにどんどんテストを追加していきます。最初は RSpec だけで始め、それからその他のテスト用のライブラリを必要に応じてテストスイートに追加していきます。

最初に、RSpec をインストールし、RSpec を使ってテストできるようにアプリケーションを設定しなければなりません。ちょっと前までは RSpec と Rails を組み合わせて使うためにそこそこの労力が必要でしたが、今はもう違います。とはいえ、それでもスペックを書く前にいくつかの gem をインストールし、ちょっとした設定を行う必要はあります。

この章では次のようなタスクを完了させます。

- まず Bundler を使って、RSpec をインストールするところから始めます。
- 必要に応じてテスト用データベースの確認とインストールを行います。
- 次にテストしたい項目をテストできるように RSpec を設定します。
- 最後に、新しい機能を追加するときにテスト用のファイルを自動生成できるよう、Rails アプリケーションを設定します。

この章で発生する変更点全体は GitHub 上のこの diff[14] で確認できます。

最初から一緒にコードを書いていきたい場合は第1章 の指示に従ってリポジトリをクローンし、それから1つ前の章のブランチからスタートしてください。

```
git checkout -b my-02-setup origin/01-untested
```

[14] https://github.com/Junichilto/everydayrails-rspec-jp-2024/compare/01-untested...02-setup

Gemfile

RSpec は Rails アプリケーションにデフォルトでは含まれていないため、まずインストールする必要があります。RSpec をインストールするためには Bundler を使います。*Gemfile* を開き、RSpec をそこに追加しましょう。

Gemfile

```
group :development, :test do
  # Rails で元から追加されている gem は省略

  gem 'rspec-rails'
end
```

もしあなたが自分で作った既存のアプリケーションで作業している場合は、Gemfile の書き方が上のコードと若干異なるかもしれません。このように書く理由は 開発環境 と テスト環境 の両方で、*rspec-rails* を読み込むためです。ただし、本番環境 では読み込みません。言い換えるなら、あなたはサーバー上ではテストを実行しない、ということです。

> 日本語版のサンプルアプリケーションでは最初から Gemfile に *rspec-rails* を追加してあるので、*Gemfile* を編集する必要はありません。これは執筆時点のバージョンと異なるバージョンの *rspec-rails* がインストールされるのを避けるためです。`bundle` コマンドを実行すれば *Gemfile.lock* に記載されたバージョンの *rspec-rails* がインストールされます。

少し詳しい話をすると、ここでは *rspec-rails* ライブラリをインストールしようとしています。*rspec-rails* には *rspec-core* とその他の独立した gem が含まれます。もしあなたが Sinatra アプリケーションやその他の非 Rails アプリケーションをテストするために RSpec を使いたい場合は、そうした gem を個別にインストールしなければなりません。rspec-rails は必要な gem をひとつにパッケージングし、インストールを楽にしてくれます。また、それに加えて、このあとで説明する Rails 向けの便利機能も提供してくれます。

コマンドラインから `bundle` コマンドを実行して、rspec-rails と関連する gem をシステムにインストールしてください。これで私たちのアプリケーションは、堅牢なテストスイートを構築するために必要な最初のブロックを手に入れました。続いて、テスト用のデータベースを作成しましょう。

テストデータベース

もし既存の Rails アプリケーションにスペックを追加するのであれば、もうすでにテストデータベースを作ってあるかもしれません。作っていないのなら、追加する方法を今から説明します。

config/database.yml というファイルを開き、あなたのアプリケーションがどのデータベースにアクセスできるか確認してください。もしこのファイルを全く変更していなければ、次のようになっているはずです。たとえば SQLite であればこうなっています。

config/database.yml

```
1 test:
2   <<: *default
3   database: db/test.sqlite3
```

MySQL や PostgreSQL を使っている場合はこうなります。

config/database.yml

```
1 test:
2   <<: *default
3   database: projects_test
```

こうしたコードが見つからなければ、必要なコードを *config/database.yml* に追加しましょう。`projects_test` の部分は自分のアプリケーションにあわせて適切に置き換えてください。

もしあなたのデータベース設定が上の例と異なっている場合は、Rails ガイドの「Rails アプリケーションを設定する[15]」を参照してください。

最後に、次の rake タスクを実行して接続可能なデータベースを作成しましょう。

```
$ bin/rails db:create:all
```

もしテストデータベースをまだ作っていないなら、今実行してみてください。すでに作成済みなら、`rails` タスクはテストデータベースがすでにあることをちゃんと教えてくれるはずです。既存のデータベースを間違って消してしまう心配はいりません。では RSpec 自体の設定に進みましょう。

[15] https://railsguides.jp/configuring.html

RSpec の設定

これでアプリケーションに spec フォルダを追加し、RSpec の基本的な設定ができるようになりました。次のようなコマンドを使って RSpec をインストールしましょう。

```
$ bin/rails generate rspec:install
```

するとジェネレータはこんな情報を表示します。

```
create  .rspec
create  spec
create  spec/spec_helper.rb
create  spec/rails_helper.rb
```

ここで作成されたのは、RSpec 用の設定ファイル（*.rspec* ）と、私たちが作成したスペックファイルを格納するディレクトリ（spec ）、それと、のちほど RSpec の動きをカスタマイズするヘルパーファイル（*spec/spec_helper.rb* と *spec/rails_helper.rb* ）です。最後の2つのファイルにはカスタマイズできる内容がコメントで詳しく書かれています。今はまだ全部読む必要はありませんが、あなたにとって RSpec が Rails 開発に欠かせないものになってきた頃に、設定をいろいろ変えながらコメントを読んでみることを強くお勧めします。設定の役割を理解するためにはそうするのが一番です。

次に、必須ではありませんが、私は RSpec の出力をデフォルトの形式から読みやすいドキュメント形式に変更するのが好みです。これによってテストスイートの実行中にどのスペックがパスし、どのスペックが失敗したのかがわかりやすくなります。それだけでなく、スペックのアウトラインが美しく出力されます。予想していたかもしれませんが、この出力を仕様書のように使うこともできます。先ほど作成された *.rspec* ファイルを開き、以下のように変更してください。

.rspec

```
--require spec_helper
--format documentation
```

このほかにも `--warnings` フラグをこのファイルに追加することもできます。*warnings* が有効になっていると、RSpec はあなたのアプリケーションや使用中の gem から出力された警告をすべて表示します。確かに、警告の表示は実際のアプリケーションを開発するときは便利です。テストの実行中に出力された非推奨メソッドの警告にはいつでも注意を払うべきでしょう。しかし、学習目的なのであれば、警告は非表示にしてテストの実行結果からノイズを減らすことをお勧めします。この設定はあとからいつでも戻せます。

試してみよう！

私たちはまだ一つもテストを書いていませんが、RSpec が正しくインストールできているかどうかは確認できます。以下のコマンドを使って RSpec を起動してみましょう。

```
$ bundle exec rspec
```

ちゃんとインストールされていれば、次のように出力されるはずです。

```
No examples found.


Finished in 0.00074 seconds (files took 0.14443 seconds to load)
0 examples, 0 failures
```

出力結果が異なる場合は、もう一度読み直してちゃんと手順どおりに作業したかどうかを確認してください。*Gemfile* に gem を追加し、それから `bundle` コマンドを実行することをお忘れなく。

`rspec` binstub を使って短いコマンドで実行できるようにする

Rails アプリケーションは Bundler の使用が必須であるため、RSpec を実行する際は `bundle exec rspec` のように毎回 `bundle exec` を付ける必要があります。binstub を作成すると `bin/rspec` のように少しタイプ量を減らすことができます。binstub を作成する場合は以下のコマンドを実行します。

```
$ bundle binstubs rspec-core
```

こうするとアプリケーションの *bin* ディレクトリ内に *rspec* という名前の実行用ファイルが作成されます。ただし、binstub を使うかどうかは読者のみなさんにお任せします。このほかにも `bundle exec` に独自のエイリアスを設定してコマンドを短くする方法もあります（例 `be rspec` など）。ですが、本書では標準的な `bundle exec rspec` を使うことにします。

ちなみに、Rails 6.1まではアプリケーションの起動時間を短くするSpring[16]を使うために binstub を作成することがありましたが、Rails 7.0からは Spring はデフォルトではインストールされなくなりました。

[16] https://github.com/rails/spring

ジェネレータ

さらに、もうひとつ手順があります。`rails generate` コマンドを使ってアプリケーションにコードを追加する際に、RSpec 用のスペックファイルも一緒に作ってもらうよう Rails を設定しましょう。

RSpec はもうインストール済みなので、Rails のジェネレータを使っても、もともとデフォルトだった Minitest のファイルは *test* ディレクトリに作成されなくなっています。その代わり、RSpec のファイルが *spec* ディレクトリに作成されます。しかし、好みに応じてジェネレータの設定を変更することができます。たとえば、`scaffold` ジェネレータを使ってコードを追加するときに気になるのは、本書であまり詳しく説明しない不要なスペックがたくさん作られてしまう点かもしれません。そこで最初からそうしたファイルを作成しないようにしてみましょう。

config/application.rb を開き、次のコードを Application クラスの内部に追加してください。

config/application.rb

```
require_relative "boot"

require "rails/all"

# Rails が最初から書いているコメントは省略 ...
Bundler.require(*Rails.groups)

module Projects
  class Application < Rails::Application
    config.load_defaults 7.1
    config.autoload_lib(ignore: %w(assets tasks))

    config.generators do |g|
      g.test_framework :rspec,
        fixtures: false,
        view_specs: false,
        helper_specs: false,
        routing_specs: false
    end
  end
end
```

このコードが何をしているかわかりますか？今から説明していきます。

- `fixtures: false` はテストデータベースにレコードを作成するファイルの作成をスキップします。この設定は第4章 で `true` に変更します。第4章以降ではファクトリを使ってテストデータを作成します。
- `view_specs: false` はビュースペックを作成しないことを指定します。本書ではビュースペックを説明しません。代わりに システムスペック で UI をテストします。
- `helper_specs: false` はヘルパーファイル用のスペックを作成しないことを指定します。ヘルパーファイルは Rails がコントローラごとに作成するファイルです。RSpec を自在に操れるようになってきたら、このオプションを true にしてヘルパーファイルをテストするようにしても良いでしょう。
- `routing_specs: false` は `config/routes.rb` 用のスペックファイルの作成を省略します。あなたのアプリケーションが本書で説明するものと同じぐらいシンプルなら、このスペックを作らなくても問題ないと思います。しかし、アプリケーションが大きくなってルーティングが複雑になってきたら、ルーティングスペックを導入するのは良い考えです。

> 日本語版のサンプルアプリケーションでは次のように `g.factory_bot false` の行も一緒に追加してください。これは第4章以降で利用する Factory Bot のジェネレータを一時的に無効化するためです。
>
> ```
> config.generators do |g|
>   g.test_framework :rspec,
>     fixtures: false,
>     view_specs: false,
>     helper_specs: false,
>     routing_specs: false
>   g.factory_bot false
> end
> ```

モデルスペックとリクエストスペックの定型コードはデフォルトで自動的に作成されます。もし自動的に作成されたくなければ、同じように設定ブロックに記述してください。たとえば、リクエストスペックの生成をスキップしたいのであれば、`request_specs: false` を追加します。

ただし、次の内容を忘れないでください。RSpec はいくつかのファイルを自動生成しないというだけであって、そのファイルを手作業で追加したり、自動生成された使う予定のないファイルを削除してはいけない、という意味ではありません。たとえば、もしヘルパースペックを追加する必要があるなら、次に説明するスペックファイルの命名規則に従ってファイルを作成し、*spec/helpers* ディレクトリに追加してください。命名規則は以下のとおりです。もし、*app/helpers/projects_helper.rb* をテストするのであれば、*spec/helpers/projects_helper_spec.rb* を追加します。*lib/my_library.rb* という架空のライブラリをテストしたいなら、*spec/lib/my_library_spec.rb* を追加します。その他のファイルについても同様です。

まとめ

この章では RSpec をアプリケーションの開発環境とテスト環境に追加し、テスト実行時に接続するテスト用のデータベースを設定しました。また、RSpec 用にいくつかのデフォルト設定ファイルを追加し、Rails がアプリケーション側のファイルに応じてテストファイルを自動的に作成する（または作成しない）ように設定することもしました。

さあ、これでテストを書く準備が整いました！次の章ではモデル層からアプリケーションの機能テストを書いていきます。

Q&A

- **test ディレクトリは削除しても良いのですか？**ゼロから新しいアプリケーションを作るのであればイエスです。これまでにアプリケーションをある程度作ってきたのであれば、まず `rails test:all` コマンドを実行し、既存のテストがないことを確認してください。既存のテストがあるなら、それらを RSpec に移行させる必要があるかもしれません。
- **なぜビューはテストしないのですか？**信頼性の高いビューのテストを作ることは非常に面倒だからです。さらにメンテナンスしようと思ったらもっと大変になります。ジェネレータを設定する際に私が述べたように、UI 関連のテストは統合テストに任せようとしています。これは Rails 開発者の中では標準的なプラクティスです。

演習問題

既存のコードベースから始める場合は次の課題に取り組んでください。

- *rspec-rails* を *Gemfile* に追加し、`bundle` コマンドでインストールしてください。本書はRails 7.1と RSpec Rails 6.1を対象にしていますが、テストに関連しているコードとテクニックはそれより古いバージョンでも大半が使えるはずです。
- アプリケーションが正しく設定され、テストデータベースと接続できることを確認してください。必要であればテストデータベースを作成してください。
- 新しくアプリケーションコードを追加するときは RSpec を使うように Rails の `generate` コマンドを設定してください。*rspec-rails* が提供しているデフォルトの設定をそのまま使うこともできます。そのままにすると、余分な定型コードも一緒に作成されます。使わないコードは手で消してもいいですし、無視しても構いません（使わないコードは削除することをお勧めします）。
- 既存のアプリケーションで必要となるテスト項目をリストアップしてください。このリストにはアプリケーションで必要不可欠な機能、過去に修正した不具合、既存の機能を壊した新機能、境界値の挙動を検証するテストなどが含まれます。こうしたシナリオはすべて次章以降で説明していきます。

新しくてきれいなコードベースから始める場合は次の課題に取り組んでください。

- 前述の説明に従い、Bundler を使って RSpec をインストールしてください。
- あなたの *database.yml* ファイルはテストデータベースを使うように設定されているかもしれません。SQLite 以外のデータベースを使っているなら、まずデータベースを作る必要があるかもしれません。まだ作っていなければ、`bin/rails db:create:all` で作成してください。
- 必須ではありませんが、Rails のジェネレータが RSpec を使うように設定してみましょう。新しいモデルとコントローラをアプリケーションに追加する際は、開発のワークフローとしてジェネレータを使えるようにし、スペックが自動的に生成されるようにしてください。

ボーナス課題

もしあなたがたくさん新しい Rails アプリケーションを作るなら、Rails アプリケーションテンプレート[17]を作ることもできます。テンプレートを使うと自動的に RSpec や関連する設

[17] https://railsguides.jp/rails_application_templates.html

定を *Gemfile* に追加したり、設定ファイルに追加したりすることができます。もちろんテストデータベースも作れます。好みのアプリケーションテンプレートを作りたい場合は、Daniel Kehoe のRails Composer[18]から始めてみるのが良いと思います。

[18] https://github.com/RailsApps/rails-composer

3. モデルスペック

RSpec のインストールが完了し、これで信頼性の高いテストスイートを構築する準備が整いました。まずアプリケーションのコアとなる部分、すなわちモデルから始めてみましょう。

本章では次のようなタスクを完了させます。

- まず既存のモデルに対してモデルスペックを作ります。
- それからモデルのバリデーション、クラスメソッド、インスタンスメソッドのテストを書きます。テストを作りながらスペックの整理もします。

既存のモデルがあるので、最初のスペックファイルは手作業で追加します。それから新しいモデルをアプリケーションに追加します。こうすると第2章で設定した便利な RSpec のジェネレータが仮のファイルを作成してくれます。

この章で発生する変更点全体は GitHub 上のこの diff[19] で確認できます。

最初から一緒にコードを書いていきたい場合は第1章 の指示に従ってリポジトリをクローンし、それから1つ前の章のブランチからスタートしてください。

```
git checkout -b my-03-models origin/02-setup
```

モデルスペックの構造

私はモデルレベルのテストが一番学習しやすいと思います。なぜならモデルをテストすればアプリケーションのコアとなる部分をテストすることになるからです。このレベルのコードが十分にテストされていれば土台が堅牢になり、そこから信頼性の高いコードベースを構築できます。

はじめに、モデルスペックには次のようなテストを含めましょう。

- 有効な属性で初期化された場合は、モデルの状態が有効（valid）になっていること
- バリデーションを失敗させるデータであれば、モデルの状態が有効になっていないこと
- クラスメソッドとインスタンスメソッドが期待通りに動作すること

[19] https://github.com/JunichiIto/everydayrails-rspec-jp-2024/compare/02-setup...03-models

良い機会なので、ここでモデルスペックの基本構成を見てみましょう。スペックの記述をアウトラインと考えるのが便利です。たとえば、メインとなる User モデルの要件を見てみましょう。

```
describe User do
  # 姓、名、メール、パスワードがあれば有効な状態であること
  it "is valid with a first name, last name, email, and password"
  # 名がなければ無効な状態であること
  it "is invalid without a first name"
  # 姓がなければ無効な状態であること
  it "is invalid without a last name"
  # メールアドレスがなければ無効な状態であること
  it "is invalid without an email address"
  # 重複したメールアドレスなら無効な状態であること
  it "is invalid with a duplicate email address"
  # ユーザーのフルネームを文字列として返すこと
  it "returns a user's full name as a string"
end
```

このアウトラインはすぐあとに展開していきますが、初心者はここからたくさんのことが学べます。これは本当にシンプルなモデルのシンプルなスペックです。しかし、次のような4つのベストプラクティスを示しています。

- **期待する結果をまとめて記述（describe）している**。このケースでは User モデルがどんなモデルなのか、そしてどんな振る舞いをするのかということを説明しています。
- **example（`it` で始まる1行）一つにつき、結果を一つだけ期待している**。私が `first_name` 、`last_name` 、`email` のバリデーションをそれぞれ分けてテストしている点に注意してください。こうすれば、example が失敗したときに問題が起きたバリデーションを 特定できます。原因調査のために RSpec の出力結果を調べる必要はありません。少なくともそこまで細かく調べずに済むはずです。
- **どの example も明示的である**。技術的なことを言うと、`it` のあとに続く説明用の文字列は必須ではありません。しかし、省略してしまうとスペックが読みにくくなります。
- **各 example の説明は動詞で始まっている。should ではない**。期待する結果を声に出して読んでみましょう。*User is invalid without a first name*（名がなければユーザーは無効な状態である）、*User is invalid without a last name*（姓がなければユーザーは無効な状態である）、*User returns a user's full name as a string*（ユーザーは文字列としてユーザーのフルネームを返す）。可読性は非常に重要であり、RSpec のキーとなる機能です！

こうしたベストプラクティスを念頭に置きながら *User* モデルのスペックを書いてみましょう。

モデルスペックを作成する

第2章ではモデルやコントローラを追加するたびに定型のテストファイルが自動的に作成されるように RSpec をセットアップしました。ジェネレータはいつでも起動できます。最初のモデルスペックを作成するため、この作業の出発地点となるファイルを実際に生成してみましょう。

まず、*rspec:model* ジェネレータをコマンドラインから実行してください。

```
$ bin/rails g rspec:model user
```

RSpec は新しいファイルを作成したことを報告します。

```
    create  spec/models/user_spec.rb
```

作成されたファイルを開き、内容を確認しましょう。

spec/models/user_spec.rb

```
1 require 'rails_helper'
2
3 RSpec.describe User, type: :model do
4   pending "add some examples to (or delete) #{__FILE__}"
5 end
```

この新しいファイルを見れば、RSpec の構文と規約がわかります。まず、このファイルでは *rails_helper* を *require* しています。この記述はテストスイート内のほぼすべてのファイルで必要になります。この記述で RSpec に対し、ファイル内のテストを実行するために Rails アプリケーションの読み込みが必要であることを伝えています。次に、*describe* メソッドを使って、*User* という名前の モデル のテストをここに書くことを示しています。*pending* 機能については第9章 で説明しますが、とりあえずここでは `bundle exec rspec` を使ってこのテストを実行してみましょう。いったい何が起こるでしょうか？

```
User
  add some examples to (or delete)
  /Users/asumner/code/examples/projects/spec/models/user_spec.rb
  (PENDING: Not yet implemented)

Pending: (Failures listed here are expected and do not affect your
suite's status)

  1) User add some examples to (or delete)
  /Users/asumner/code/examples/projects/spec/models/user_spec.rb
     # Not yet implemented
     # ./spec/models/user_spec.rb:4


Finished in 0.00107 seconds (files took 0.43352 seconds to load)
1 example, 0 failures, 1 pending
```

> 必ずしもスペックファイルを作成するためにジェネレータを利用する必要はありません。しかし、ジェネレータの使用はタイプミスによるつまらないエラーを防止するための良い方法です。

describe の外枠はそのままにして、その内側を先ほど作成したアウトラインに置き換えてみましょう。

spec/models/user_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe User, type: :model do
4    # 姓、名、メール、パスワードがあれば有効な状態であること
5    it "is valid with a first name, last name, email, and password"
6    # 名がなければ無効な状態であること
7    it "is invalid without a first name"
8    # 姓がなければ無効な状態であること
9    it "is invalid without a last name"
10   # メールアドレスがなければ無効な状態であること
11   it "is invalid without an email address"
12   # 重複したメールアドレスなら無効な状態であること
```

```
13    it "is invalid with a duplicate email address"
14    # ユーザーのフルネームを文字列として返すこと
15    it "returns a user's full name as a string"
16  end
```

詳細はこのあと追加していきますが、この状態でコマンドラインからスペックを実行すると（コマンドラインから bundle exec rspec とタイプしてください）、出力結果は次のようになります。

```
User
  is valid with a first name, last name, email, and password (PENDING:
  Not yet implemented)
  is invalid without a first name (PENDING: Not yet implemented)
  is invalid without a last name (PENDING: Not yet implemented)
  is invalid without an email address (PENDING: Not yet implemented)
  is invalid with a duplicate email address (PENDING: Not yet implemented)
  returns a user's full name as a string (PENDING: Not yet implemented)

Pending: (Failures listed here are expected and do not affect your
suite's status)

  1) User is valid with a first name, last name, email, and password
     # Not yet implemented
     # ./spec/models/user_spec.rb:5

  2) User is invalid without a first name
     # Not yet implemented
     # ./spec/models/user_spec.rb:7

  3) User is invalid without a last name
     # Not yet implemented
     # ./spec/models/user_spec.rb:9

  4) User is invalid without an email address
     # Not yet implemented
     # ./spec/models/user_spec.rb:11

  5) User is invalid with a duplicate email address
```

```
    # Not yet implemented
    # ./spec/models/user_spec.rb:13

  6) User returns a user's full name as a string
    # Not yet implemented
    # ./spec/models/user_spec.rb:15


Finished in 0.00176 seconds (files took 2.18 seconds to load)
6 examples, 0 failures, 6 pending
```

すばらしい！6つの保留中（pending）のスペックができあがりました。私たちはまだ実行可能なテストを何も書いていないので、RSpec はここで作成したスペックを *pending* と表示しています。それでは実際にテストを書いていきましょう。まずは一番最初の example から始めます。

古いバージョンの Rails では、Rake タスクを使って開発用データベースの構造を手作業でテストデータベースコピーにする必要がありました。しかし、現在ではマイグレーションを実行すると、Rails がこのコピー作業を (ほぼ毎回) 自動的に処理してくれます。もしテスト環境でマイグレーションが未実行になっているというエラーが出た場合は、`bin/rails db:migrate RAILS_ENV=test` というコマンドを実行してデータベースの構造を最新にしてください。

RSpec の構文

その昔、RSpec は「〜が期待した結果と一致すべきだ/すべきでない（something `should` or `should_not` match expected output）」と読むことができる `should` 構文を使っていました。

しかし、2012年にリリースされた RSpec 2.11からは「私は〜が〜になる/ならないことを期待する（I `expect` something `to` or `not_to` be something else）」と読むことができる `expect` 構文に変わりました。構文が変わったのは、古い構文でときどき発生していた技術的な問題を回避するため[20]です。

2つの構文を比較するために、簡単なテスト、つまりエクスペクテーション（expectation、期待する内容）の使用例を見てみましょう。この example の場合、2 + 1はいつでも3に等しいはずですよね？古い RSpec の構文ではこのように書きます。

[20] http://rspec.info/blog/2012/06/rspecs-new-expectation-syntax/

```
# 2と1を足すと3になること
it "adds 2 and 1 to make 3" do
  (2 + 1).should eq 3
end
```

現行の expect 構文ではテストする値を expect() メソッドに渡し、それに続けてマッチャを呼び出します。

```
# 2と1を足すと3になること
it "adds 2 and 1 to make 3" do
  expect(2 + 1).to eq 3
end
```

Google や Stack Overflow で RSpec に関する質問を検索したり、古い Rails アプリケーションを開発したりすると、古い should 構文を使ったコードを今でも見かけることがあるかもしれません。この構文は現行バージョンの RSpec でも動作しますが、使うと非推奨であるとの警告が出力されます。設定を変更すればこの警告を出力しないようにすることも できますが、そんなことはせずに新しい expect() 構文を学習した方が良いと思います。

では、実際の example ではどうなるでしょうか？User モデルの最初のエクスペクテーションで使ってみましょう。

spec/models/user_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe User, type: :model do
4    # 姓、名、メール、パスワードがあれば有効な状態であること
5    it "is valid with a first name, last name, email, and password" do
6      user = User.new(
7        first_name: "Aaron",
8        last_name:  "Sumner",
9        email:      "tester@example.com",
10       password:   "dottle-nouveau-pavilion-tights-furze",
11     )
12     expect(user).to be_valid
13   end
14
15   # 名がなければ無効な状態であること
16   it "is invalid without a first name"
17   # 姓がなければ無効な状態であること
```

```
18    it "is invalid without a last name"
19    # メールアドレスがなければ無効な状態であること
20    it "is invalid without an email address"
21    # 重複したメールアドレスなら無効な状態であること
22    it "is invalid with a duplicate email address"
23    # ユーザーのフルネームを文字列として返すこと
24    it "returns a user's full name as a string"
25  end
```

この単純な example は be_valid という RSpec のマッチャを使って、モデルが有効な状態を理解できているかどうかを検証しています。まずオブジェクトを作成し（このケースでは新しく作られているが保存はされていない User クラスのインスタンスを作成し、user という名前の変数に格納しています)、それからオブジェクトを expect に渡して、マッチャと比較しています。

それでは bundle exec rspec をコマンドラインから再実行してみましょう。すると、1つの example がパスしたと表示されるはずです。

```
User
  is valid with a first name, last name and email, and password
  is invalid without a first name (PENDING: Not yet implemented)
  is invalid without a last name (PENDING: Not yet implemented)
  is invalid without an email address (PENDING: Not yet implemented)
  is invalid with a duplicate email address (PENDING: Not yet implemented)
  returns a user's full name as a string (PENDING: Not yet implemented)

Pending: (Failures listed here are expected and do not affect your
suite's status)

  1) User is invalid without a first name
     # Not yet implemented
     # ./spec/models/user_spec.rb:16

  2) User is invalid without a last name
     # Not yet implemented
     # ./spec/models/user_spec.rb:18

  3) User is invalid without an email address
     # Not yet implemented
```

```
     # ./spec/models/user_spec.rb:20

  4) User is invalid with a duplicate email address
     # Not yet implemented
     # ./spec/models/user_spec.rb:22

  5) User returns a user's full name as a string
     # Not yet implemented
     # ./spec/models/user_spec.rb:24


Finished in 0.02839 seconds (files took 0.28886 seconds to load)
6 examples, 0 failures, 5 pending
```

おめでとうございます。これで最初のテストが完成しました！ではこれからもっとコードをテストしていって、保留中のテストを完全になくしてしまいましょう。

バリデーションをテストする

バリデーションはテストの自動化に慣れるための良い題材です。バリデーションのテストはたいてい1〜2行で書けます。では first_name バリデーションのスペックについて詳細を見てみましょう。

spec/models/user_spec.rb

```
# 名がなければ無効な状態であること
it "is invalid without a first name" do
  user = User.new(first_name: nil)
  user.valid?
  expect(user.errors[:first_name]).to include("can't be blank")
end
```

今回は新しく作ったユーザー（first_name には明示的に nil をセットします）に対して valid? メソッドを呼び出すと有効（valid）に ならず 、ユーザーの first_name 属性にエラーメッセージが付いていることを 期待（expect） します。RSpec をもう一度実行すると、二番目までのスペックがパスするはずです。ここで RSpec の include マッチャについて確認しましょう。このマッチャは繰り返し可能な値（enumerable value）の中に、ある値が存在するかどうかをチェックします。では RSpec をもう一度実行します。すると、今回は2つのスペックがパスするはずです。

ここまでのアプローチにはちょっとした問題があります。現時点で2つのテストがパスしていますが、私たちはまだテストが 失敗 するところを見ていません。これは警告すべき兆候です。特に、テストを書き始めたタイミングであればなおさらです。私たちはテストコードが意図した通りに動いていることを確認しなければなりません。これは「テスト対象のコードでいろいろ試すアプローチ（exercising the code under test）」としても知られています。

誤判定ではないことを証明するためには二つのやり方があります。ひとつめは、to を to_not に変えてエクスペクテーションを反転させてみます。

spec/models/user_spec.rb

```
1  # 名がなければ無効な状態であること
2  it "is invalid without a first name" do
3    user = User.new(first_name: nil)
4    user.valid?
5    expect(user.errors[:first_name]).to_not include("can't be blank")
6  end
```

当然のごとく、RSpec はテストの失敗を報告します。

```
Failures:

  1) User is invalid without a first name
     Failure/Error: expect(user.errors[:first_name]).to_not
     include("can't be blank")
       expected ["can't be blank"] not to include "can't be blank"
     # ./spec/models/user_spec.rb:17:in `block (2 levels) in <main>'

Finished in 0.06211 seconds (files took 0.28541 seconds to load)
6 examples, 1 failure, 5 pending

Failed examples:

rspec ./spec/models/user_spec.rb:14 # User is invalid without a first name
```

RSpec ではこうしたエクスペクテーションを書くために to_not と not_to が提供されています。役割はどちらも全く同じです。本書では RSpec のドキュメントで広く使われている to_not を使います。

もうひとつ、アプリケーション側のコードを変更して、テストの実行結果にどんな変化が起きるか確認する方法もあります。先ほどのテストコードの変更を元に戻し（to_not を to

に戻す)、それから User モデルを開いて first_name のバリデーションをコメントアウトしてください。

app/models/user.rb

```
class User < ApplicationRecord
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  # validates :first_name, presence: true
  validates :last_name, presence: true

  # 残りのコードは省略 ...
```

スペックを再実行すると、再度失敗が表示されるはずです。これはすなわち、私たちは RSpec に対して名を持たないユーザーは無効であると伝えたのに、アプリケーション側がその仕様を実装していないことを意味しています。

この二つの方法は、自分の書いたテストが期待どおりに動いているかどうか確認する簡単な方法です。シンプルなバリデーションからもっと複雑なロジックに進むときであれば、特に有効です。また、この方法は既存のアプリケーションをテストするためにも有効です。もしテストの出力結果に何も変化がなければ、それはよいチャンスです。変化がない場合は、テストがアプリケーション側のコードと連携していなかったり、コードが期待した動きと異なっていたりすることを意味しています。

では、:last_name のバリデーションも同じアプローチでテストしてみましょう。

spec/models/user_spec.rb

```
# 姓がなければ無効な状態であること
it "is invalid without a last name" do
  user = User.new(last_name: nil)
  user.valid?
  expect(user.errors[:last_name]).to include("can't be blank")
end
```

「こんなテストは役に立たない。モデルに含まれるすべてのバリデーションを確認しようとしたらどれくらい大変になるのかわかっているのか?」そんなふうに思っている人もいるかもしれません。ですが、実際はあなたが考えている以上にバリデーションは書き忘れやすいものです。しかし、それよりもっと大事なことは、テストを書いている最中に モデルが持

つべきバリデーションについて考えれば、バリデーションの追加を忘れにくくなるということです。（このプロセスはテスト駆動開発でコードを書くのが理想的ですし、最後は実際そうします。）

ここまでに得た知識を使って、もう少し複雑なテストを書いてみましょう。今回は email 属性のユニークバリデーションをテストします。

spec/models/user_spec.rb

```
# 重複したメールアドレスなら無効な状態であること
it "is invalid with a duplicate email address" do
  User.create(
    first_name:  "Joe",
    last_name:  "Tester",
    email:      "tester@example.com",
    password:   "dottle-nouveau-pavilion-tights-furze",
  )
  user = User.new(
    first_name:  "Jane",
    last_name:  "Tester",
    email:      "tester@example.com",
    password:   "dottle-nouveau-pavilion-tights-furze",
  )
  user.valid?
  expect(user.errors[:email]).to include("has already been taken")
end
```

ここではちょっとした違いがあることに注意してください。このケースではテストの前にユーザーを保存しました（User に対して new の代わりに create を呼んでいます）。それから2件目のユーザーをテスト対象のオブジェクトとしてインスタンス化しました。もちろん、最初に保存されたユーザーは有効な状態（姓、名、メール、パスワードが全部ある）であり、なおかつ、同一のメールアドレスも設定されている必要があります。第4章ではこのプロセスをもっと効率よく処理する方法を説明します。では、`bundle exec rspec` を実行して新しいテストの出力結果を確認してください。

続いてもっと複雑なバリデーションをテストしましょう。User モデルの話はいったん横に置いて、今度は Project モデルに着目します。たとえば、ユーザーは同じ名前のプロジェクトを作成できないという要件があったとします。つまり、プロジェクト名はユーザーごとにユニークでなければならない、ということです。別の言い方をすると、私は *Paint the house*（家を塗る）という複数のプロジェクトを持つことはできないが、あなたと私はそれぞれ *Paint*

the house というプロジェクトを持つことができる、ということです。あなたならどうやってテストしますか？

では Project モデル用に新しいスペックファイルを作成しましょう。

```
$ bin/rails g rspec:model project
```

続いて、作成されたファイルに二つの example を追加します。ここでテストしたいのは、一人のユーザーは同じ名前で二つのプロジェクトを作成できないが、ユーザーが異なるときは同じ名前のプロジェクトを作成できる、という要件です。

spec/models/project_spec.rb

```
require 'rails_helper'

RSpec.describe Project, type: :model do
  # ユーザー単位では重複したプロジェクト名を許可しないこと
  it "does not allow duplicate project names per user" do
    user = User.create(
      first_name: "Joe",
      last_name:  "Tester",
      email:      "joetester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )

    user.projects.create(
      name: "Test Project",
    )

    new_project = user.projects.build(
      name: "Test Project",
    )

    new_project.valid?
    expect(new_project.errors[:name]).to include("has already been taken")
  end

  # 二人のユーザーが同じ名前を使うことは許可すること
  it "allows two users to share a project name" do
    user = User.create(
      first_name: "Joe",
```

```
29        last_name:  "Tester",
30        email:      "joetester@example.com",
31        password:   "dottle-nouveau-pavilion-tights-furze",
32      )
33
34      user.projects.create(
35        name: "Test Project",
36      )
37
38      other_user = User.create(
39        first_name: "Jane",
40        last_name:  "Tester",
41        email:      "janetester@example.com",
42        password:   "dottle-nouveau-pavilion-tights-furze",
43      )
44
45      other_project = other_user.projects.build(
46        name: "Test Project",
47      )
48
49      expect(other_project).to be_valid
50    end
51  end
```

今回は User モデルと Project モデルが Active Record のリレーションで互いに関連するため、そのぶん多くの情報を記述する必要があります。最初の example では両方のプロジェクトを割り当てられた一人のユーザーがいます。二つ目の example では二つの別々のプロジェクトに同じ名前が割り当てられ、それらが別々のユーザーに属しています。ここでは以下の点に注意してください。二つの example はどちらもユーザーを create してデータベースに保存しています。これはユーザーをテスト対象のプロジェクトに割り当てる必要があるためです。

Project モデルには以下のようなバリデーションが設定されています。

app/models/project.rb

```
validates :name, presence: true, uniqueness: { scope: :user_id }
```

今回作成したスペックは問題なくパスします。ですが、例のチェックをお忘れなく。一時的にバリデーションをコメントアウトしたり、テストを書き換えたりして、結果が変わることを確認してください。テストはちゃんと失敗するでしょうか？

もちろん、バリデーションは scope が一つしかないような単純なものばかりではなく、もっと複雑になる場合があります。もしかするとあなたは複雑な正規表現やカスタムバリデータを使っているかもしれません。こうしたバリデーションもテストする習慣を付けてください。正常系のパターンだけでなく、エラーが発生する条件もテストしましょう。たとえば、これまでに作ってきた example ではオブジェクトが `nil` で初期化された場合の実行結果もテストしました。もし数値しか受け付けない属性のバリデーションがあるなら、文字列を渡してください。もし4文字から8文字の文字列を要求するバリデーションがあるなら、3文字と9文字の文字列を渡してください。

インスタンスメソッドをテストする

それでは User モデルのテストに戻ります。このサンプルアプリケーションでは、ユーザーの姓と名を毎回連結して新しい文字列を作るより、`@user.name` を呼び出すだけでフルネームが出力されるようにした方が便利です。というわけでこんなメソッドが `User` クラスに作ってあります。

app/models/user.rb

```
def name
  [first_name, last_name].join(' ')
end
```

バリデーションの example と同じ基本的なテクニックでこの機能の example を作ることができます。

spec/models/user_spec.rb

```
it "returns a user's full name as a string" do
  user = User.new(
    first_name: "John",
    last_name:  "Doe",
    email:      "johndoe@example.com",
  )
  expect(user.name).to eq "John Doe"
end
```

RSpec で等値のエクスペクテーションを書くときは == ではなく eq を使います。

テストデータを作り、それからあなたが期待する振る舞いを RSpec に教えてあげてください。簡単ですね。では続けましょう。

クラスメソッドとスコープをテストする

このアプリケーションには渡された文字列でメモ（note）を検索する機能を用意してあります。念のため説明しておくと、この機能は Note モデルにスコープとして実装されています。

app/models/note.rb

```
scope :search, ->(term) {
  where("LOWER(message) LIKE ?", "%#{term.downcase}%")
}
```

では Note モデル用に3つめのファイルをテストスイートに追加しましょう。`rspec:model` ジェネレータでファイルを作ったら、最初のテストを追加してください。

spec/models/note_spec.rb

```ruby
require 'rails_helper'

RSpec.describe Note, type: :model do
  # 検索文字列に一致するメモを返すこと
  it "returns notes that match the search term" do
    user = User.create(
      first_name: "Joe",
      last_name:  "Tester",
      email:      "joetester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )

    project = user.projects.create(
      name: "Test Project",
    )

    note1 = project.notes.create(
      message: "This is the first note.",
      user: user,
    )
    note2 = project.notes.create(
      message: "This is the second note.",
      user: user,
    )
    note3 = project.notes.create(
      message: "First, preheat the oven.",
      user: user,
    )

    expect(Note.search("first")).to include(note1, note3)
    expect(Note.search("first")).to_not include(note2)
  end
end
```

search スコープは検索文字列に一致するメモのコレクションを返します。返されたコレクションは一致したメモだけが含まれるはずです。その文字列を含まないメモはコレクションに含まれません。

このテストでは次のような実験ができます。to を to_not に変えたらどうなるでしょうか？もしくは検索文字列を含むメモをさらに追加したらどうなるでしょうか？

失敗をテストする

正常系のテストは終わりました。ユーザーが文字列検索すると結果が返ってきます。しかし、結果が返ってこない文字列で検索したときはどうでしょうか？そんな場合もテストした方が良いです。次のスペックがそのテストになります。

spec/models/note_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe Note, type: :model do
4    # 検索結果を検証するスペック...
5
6    # 検索結果が1件も見つからなければ空のコレクションを返すこと
7    it "returns an empty collection when no results are found" do
8      user = User.create(
9        first_name: "Joe",
10       last_name:  "Tester",
11       email:      "joetester@example.com",
12       password:   "dottle-nouveau-pavilion-tights-furze",
13     )
14
15     project = user.projects.create(
16       name: "Test Project",
17     )
18
19     note1 = project.notes.create(
20       message: "This is the first note.",
21       user: user,
22     )
23     note2 = project.notes.create(
24       message: "This is the second note.",
25       user: user,
26     )
27     note3 = project.notes.create(
28       message: "First, preheat the oven.",
```

```
29        user: user,
30      )
31
32      expect(Note.search("message")).to be_empty
33    end
34  end
```

このスペックでは `Note.search("message")` を実行して返却された配列をチェックします。この配列は 確かに 空なのでスペックはパスします！これで理想的な結果、すなわち結果が返ってくる文字列で検索した場合だけでなく、結果が返ってこない文字列で検索した場合もテストしたことになります。

マッチャについてもっと詳しく

これまで四つのマッチャ（`be_valid` 、`eq` 、`include` 、`be_empty` ）を実際に使いながら見てきました。最初に使ったのは `be_valid` です。このマッチャは *rspec-rails* gem が提供するマッチャで、Rails のモデルの有効性をテストします。`eq` と `include` は *rspec-expectations* で定義されているマッチャで、前章で RSpec をセットアップしたときに *rspec-rails* と一緒にインストールされました。

RSpec が提供するデフォルトのマッチャをすべて見たい場合はGitHub にある rspec-expectations リポジトリ[21]の *README* が参考になるかもしれません。この中に出てくるマッチャのいくつかは本書全体を通して説明していきます。また、第8章 では自分でカスタムマッチャを作る方法も説明します。

describe、context、before、after を使ってスペックを DRY にする

ここまでに作成したメモ用のスペックには冗長なコードが含まれます。具体的には、各 example の中ではまったく同じ4つのオブジェクトを作成しています。アプリケーションコードと同様に、DRY 原則はテストコードにも当てはまります（いくつか例外もあるので、のちほど説明します）。では RSpec の機能をさらに活用してテストコードをきれいにしてみましょう。

[21] https://github.com/rspec/rspec-expectations

先ほど作った Note モデルのスペックに注目してみましょう。まず最初にやるべきことは describe ブロックを describe Note ブロックの 中に 作成することです。これは検索機能にフォーカスするためです。アウトラインを抜き出すと、このようになります。

spec/models/note_spec.rb

```
require 'rails_helper'

RSpec.describe Note, type: :model do

  # バリデーション用のスペックが並ぶ

  # 文字列に一致するメッセージを検索する
  describe "search message for a term" do
    # 検索用の example が並ぶ ...
  end
end
```

二つの context ブロックを加えてさらに example を切り分けましょう。一つは「一致するデータが見つかるとき」で、もう一つは「一致するデータが1件も見つからないとき」です。

spec/models/note_spec.rb

```
require 'rails_helper'

RSpec.describe Note, type: :model do

  # 他のスペックが並ぶ

  # 文字列に一致するメッセージを検索する
  describe "search message for a term" do

    # 一致するデータが見つかるとき
    context "when a match is found" do
      # 一致する場合の example が並ぶ ...
    end

    # 一致するデータが1件も見つからないとき
    context "when no match is found" do
      # 一致しない場合の example が並ぶ ...
    end
```

```
19    end
20  end
```

describe と context は技術的には交換可能なのですが、私は次のように使い分けるのが好きです。すなわち、describe ではクラスやシステムの機能に関するアウトラインを記述し、context では特定の状態に関するアウトラインを記述するようにします。このケースであれば、状態は二つあります。一つは結果が返ってくる検索文字列を渡された状態で、もう一つは結果が返ってこない検索文字列が渡された状態です。

お気づきかもしれませんが、このように example のアウトラインを作ると、同じような example をひとまとめにして分類できます。こうするとスペックがさらに読みやすくなります。では最後に、before フックを利用してスペックのリファクタリングを完了させましょう。before ブロックの中に書かれたコードは内側の各テストが実行される前に実行されます。また、before ブロックは describe や context ブロックによってスコープが限定されます。たとえばこの例で言うと、before ブロックのコードは "search message for a term" ブロックの内側にある全部のテストに先立って実行されます。ですが、新しく作った describe ブロックの外側にあるその他の example の前には実行されません。

spec/models/note_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe Note, type: :model do
4
5    before do
6      # このファイルの全テストで使用するテストデータをセットアップする
7    end
8
9    # バリデーションのテストが並ぶ
10
11   # 文字列に一致するメッセージを検索する
12   describe "search message for a term" do
13
14     before do
15       # 検索機能の全テストに関連する追加のテストデータをセットアップする
16     end
17
18     # 一致するデータが見つかるとき
19     context "when a match is found" do
```

```
20        # 一致する場合の example が並ぶ ...
21      end
22
23      # 一致するデータが1件も見つからないとき
24      context "when no match is found" do
25        # 一致しない場合の example が並ぶ ...
26      end
27    end
28  end
```

RSpec の before フックはスペック内の冗長なコードを認識し、きれいにするための良い出発点になります。これ以外にも冗長なテストコードをきれいにするテクニックはありますが、before を使うのが最も一般的かもしれません。before ブロックは example ごとに、またはブロック内の各 example ごとに、またはテストスイート全体を実行するごとに実行されます。

- before(:each) は describe または context ブロック内の 各（each） テストの前に実行されます。好みに応じて before(:example) というエイリアスを使ってもいいですし、上のサンプルコードで書いたように before だけでも構いません。もしブロック内に4つのテストがあれば、before のコードも4回実行されます。
- before(:all) は describe または context ブロック内の 全（all） テストの前に一回だけ実行されます。かわりに before(:context) というエイリアスを使っても構いません。こちらは before のコードは一回だけ実行され、それから4つのテストが実行されます。
- before(:suite) はテストスイート全体の全ファイルを実行する前に実行されます。

before(:all) と before(:suite) は時間のかかる独立したセットアップ処理を1回だけ実行し、テスト全体の実行時間を短くするのに役立ちます。ですが、この機能を使うとテスト全体を汚染してしまう原因にもなりかねません。可能な限り before(:each) を使うようにしてください。

上で示したような書き方で before ブロックを定義すると、各（each） テストの前にブロック内のコードが実行されます。before のかわりに before :each のように定義すれば、より明示的な書き方になります。どちらを使っても構わないので、あなた自身やあなたのチームの好みに応じてお好きな方を使ってください。

もし example の実行後に後片付けが必要になるのであれば（たとえば外部サービスとの接続を切断する場合など）、after フックを使って各 example のあと（after）に後片付けすることもできます。before と同様、after にも each 、all 、suite のオプションがあります。RSpec

の場合、デフォルトでデータベースの後片付けをやってくれるので、私は after を使うことはほとんどありません。

さて、整理後の全スペックを見てみましょう。

spec/models/note_spec.rb

```
require 'rails_helper'

RSpec.describe Note, type: :model do
  before do
    @user = User.create(
      first_name: "Joe",
      last_name:  "Tester",
      email:      "joetester@example.com",
      password:   "dottle-nouveau-pavilion-tights-furze",
    )

    @project = @user.projects.create(
      name: "Test Project",
    )
  end

  # ユーザー、プロジェクト、メッセージがあれば有効な状態であること
  it "is valid with a user, project, and message" do
    note = Note.new(
      message: "This is a sample note.",
      user: @user,
      project: @project,
    )
    expect(note).to be_valid
  end

  # メッセージがなければ無効な状態であること
  it "is invalid without a message" do
    note = Note.new(message: nil)
    note.valid?
    expect(note.errors[:message]).to include("can't be blank")
  end

```

```
34    # 文字列に一致するメッセージを検索する
35    describe "search message for a term" do
36      before do
37        @note1 = @project.notes.create(
38          message: "This is the first note.",
39          user: @user,
40        )
41        @note2 = @project.notes.create(
42          message: "This is the second note.",
43          user: @user,
44        )
45        @note3 = @project.notes.create(
46          message: "First, preheat the oven.",
47          user: @user,
48        )
49      end
50
51      # 一致するデータが見つかるとき
52      context "when a match is found" do
53        # 検索文字列に一致するメモを返すこと
54        it "returns notes that match the search term" do
55          expect(Note.search("first")).to include(@note1, @note3)
56        end
57      end
58
59      # 一致するデータが1件も見つからないとき
60      context "when no match is found" do
61        # 空のコレクションを返すこと
62        it "returns an empty collection" do
63          expect(Note.search("message")).to be_empty
64        end
65      end
66    end
67  end
```

みなさんはもしかするとテストデータのセットアップ方法が少し変わったことに気づいたかもしれません。セットアップの処理を各テストから before ブロックに移動したので、各ユーザーはインスタンス変数にアサインする必要があります。そうしないとテストの中で変

数名を指定してデータにアクセスできないからです。

これらのスペックを実行すると、こんなふうに素敵なアウトラインが表示されます（第2章でドキュメント形式を使うようにRSpecを設定したからです）。

```
Note
  is valid with a user, project, and message
  is invalid without a message
  search message for a term
    when a match is found
      returns notes that match the search term
    when no match is found
      returns an empty collection

Project
  does not allow duplicate project names per user
  allows two users to share a project name

User
  is valid with a first name, last name and email, and password
  is invalid without a first name
  is invalid without a last name
  is invalid with a duplicate email address
  returns a user's full name as a string

Finished in 0.22564 seconds (files took 0.32225 seconds to load)
11 examples, 0 failures
```

開発者の中には入れ子になった`describe`ブロックで説明文の代わりにメソッド名を書くのが好きな人もいます。たとえば、私が`search for first name, last name, or email`と書いたラベルは`#search`になります。個人的にはこう書くのは好きではありません。なぜならこのラベルはコードの振る舞いを定義するものであり、メソッドの名前を書く場所ではないと思うからです。しかし、私はこの考え方についてそこまで強くこだわっているわけではありません。

どれくらいDRYだとDRYすぎるのか？

本章では長い時間をかけてスペックを理解しやすいブロックに分けて整理しました。しかし、これは弊害を起こしやすい機能です。

example のテスト条件をセットアップする際、可読性を考えて DRY 原則に違反するのは問題ありません。私はそう考えています。もし自分がテストしている内容を確認するために、大きなスペックファイルを頻繁にスクロールしているようなら（もしくはあとで説明する外部のサポートファイルを大量に読み込んでいるようなら）、テストデータのセットアップを小さな `describe` ブロックの中で重複させることを検討してください。`describe` ブロックの中だけでなく、example の中でも OK です。

とはいえ、そんな場合でも変数とメソッドに良い名前を付けるのは大変効果的です。たとえば上のスペックでは `@note1` や `@note2` 、`@note3` のような名前をテスト用のメモに使いました。しかし、場合によっては `@matching_note` （一致するメモ）や `@note_with_numbers_only`（数字だけのメモ）といった変数名を使いたくなるかもしれません。何が適切かはテストする内容に依りますが、一般論としてはわかりやすい変数名とメソッド名を付けるように心がけてください！

このトピックについては第8章 でさらに詳しく説明します。

まとめ

本章ではモデルのテストにフォーカスしましたが、このあとに登場するモデル以外のスペックでも使えるその他の重要なテクニックもたくさん説明しました。

- **期待する結果は能動形で明示的に記述すること。** example の結果がどうなるかを動詞を使って説明してください。チェックする結果は example 一つに付き一個だけにしてください。
- **起きて ほしい ことと、起きて ほしくない ことをテストすること。** example を書くときは両方のパスを考え、その考えに沿ってテストを書いてください。
- **境界値テストをすること。**もしパスワードのバリデーションが4文字以上10文字以下なら、8文字のパスワードをテストしただけで満足しないでください。4文字と10文字、そして3文字と11文字もテストするのが良いテストケースです。（もちろん、なぜそんなに短いパスワードを許容し、なぜそれ以上長いパスワードを許容しないのか、と自問するチャンスかもしれません。テストはアプリケーションの要件とコードを熟考するための良い機会でもあります。）
- **可読性を上げるためにスペックを整理すること。** `describe` と `context` はよく似た example を分類してアウトライン化します。`before` ブロックと `after` ブロックは重複を取り除きます。しかし、テストの場合は DRY であることよりも読みやすいことの方が

重要です。もし頻繁にスペックファイルをスクロールしていることに気付いたら、それはちょっとぐらいリピートしても問題ないというサインです。

アプリケーションに堅牢なモデルスペックを揃えたので、あなたは順調にコードの信頼性を上げてきています。

Q&A

describe と context はどう使い分けるべきでしょうか？ RSpec の立場からすれば、あなたはいつでも好きなときに `describe` が使えます。RSpec の他の機能と同じく、`context` はあなたのスペックを読みやすくするためにあります。私が本章でやったように、一つの条件をまとめるために `context` を使うのも良いですし、アプリケーションの状態（たとえば「発射準備完了」状態のロケットと、「準備未完了」状態のロケットなど）をまとめるために `context` を使うこともできます。

演習問題

サンプルアプリケーションにモデルのテストをさらに追加する。私はモデルが持つ一部の機能にしかテストを追加していません。たとえば、Project モデルのスペックにはバリデーションのスペックが欠けています。それを今、追加してみてください。もしあなたが RSpec を使ってテストできるように設定された自分自身のアプリケーションを持っているなら、そこにもモデルスペックを追加してみてください。

訳者あとがき

> このあとがきは日本語版の初版リリース時に作成されたものです。

伊藤淳一

Rails も RSpec も、日本では比較的ポピュラーな web フレームワーク/テストツールです。しかし、日本語で書かれた Rails の技術書や RSpec の技術書は発売されていても、「RSpec で Rails をテストする」というテーマだけにフォーカスを当てた日本の技術書はおそらくないと思います。きっと、日本の多くの技術者は web 上に散らばった情報を参考にしたり、職場のメンバーに教えてもらったりしながら、各自で「RSpec で Rails をテストする方法」を模索し続けていたのではないでしょうか。実際、私がそうでしたから。

本書、「Everyday Rails - RSpec による Rails テスト入門」（原題: *Everyday Rails Testing with RSpec*）はそんな日本の Rails プログラマの状況をきっと変えてくれる一冊になると私は信じています。非常に初歩的な話から中級者でも知らないような高度なテクニックまで、これほど体系立てて実践的に説明してくれる技術書は他にないからです。本書を読んで内容を理解し、Aaron が言うとおりに自分のアプリケーションで自動テストを組みこんで練習すれば、全くの RSpec 初心者でも一気に自動テストのスキルを向上させることができるはずです。目を使って 読む だけではなく、ぜひ自分の手と頭を動かして本書の内容を 身体で理解 してください！

最後に、私の家族へ向けて感謝の気持ちを。スーパーマンではなく、ただの凡人である私は、翻訳の作業時間を作るために家族との時間を削ることぐらいしかできませんでした。妻にも子どもたちにも、ここ数ヶ月はちょっと淋しい思いをさせていたかもしれません。今まで我慢してくれてどうもありがとう。この翻訳の仕事が落ち着いたら、みんなでどこかのんびりと旅行にでも行きましょう！

日本語版の謝辞

改訂版（2017年）の謝辞

改訂版の翻訳レビューはソニックガーデンのプログラマである、宋大羽さん[22]、木原忠大さん[23]、森田高士さん[24]、遠藤大介さん[25]、安達輝雄さん[26]に協力してもらいました。短いレビュー期間の中で多くのフィードバックを上げてくれたことに感謝します。

初版の謝辞

本書を翻訳するにあたって、お世話になった方々のお名前を挙げさせてください。翻訳者チームの力だけでは本書の翻訳を完成させることは決してできませんでした。

まず、著者の Aaron とは Facebook グループや GitHub の Issue 上で何度もやりとりを交わしました。忙しい中、毎回丁寧に応対してくれたことを非常に感謝しています。ちなみに、Aaron のラストネームは「サマー（Summer）」ではありません。「サムナー（Sumner）」ですのでお間違いなく。

技術評論社の傳智之さん[27]には技術書を出版する際の進め方についてアドバイスをいただきました。楽天株式会社の藤原大さん[28]には翻訳者としての経験を元に貴重なアドバイスをいただきました。

Leanpub[29]は海外発のサービスということもあって、時々電子書籍中の日本語表示がおかしくなることがありました。そんなときに何度も辛抱強く我々の修正リクエストに応対してくれた Leanpub の Mike, Scott, Peter にも感謝しています。おかげでとてもきれいな日本語の電子書籍が完成しました。また、電子書籍で使えそうな日本語フォントを探しているときに Twitter で「あおぞら明朝」の存在を教えてくれたがんじゃさん[30]と、このフォントを作られた bluskis さんにも大変感謝しています。

[22] https://github.com/shirogani
[23] https://github.com/tkiha
[24] https://github.com/moritamori
[25] https://twitter.com/ruzia
[26] https://twitter.com/interu
[27] https://twitter.com/dentomo
[28] https://twitter.com/daipresents
[29] https://leanpub.com
[30] https://twitter.com/thc_o0

お忙しい中、ベータ版のレビューをしてくれた橋立友宏さん[31]、西川茂伸さん[32]、遠藤大介さん[33]にも大変助けられました。どなたも我々だけでは気付かなかった翻訳の問題点や技術的な誤りを指摘していただきました。そして、西脇.rb[34]のイギリス人プログラマ、マイケル（P. Michael Holland）[35]はほとんど4人目の翻訳者と呼んでも良いぐらいの活躍をしてもらいました。彼が英語と日本語の橋渡しをしてくれていなければ、この翻訳がもっともっと辛い作業になっていたことは間違いありません。

最後に、本書を購入してくださったみなさんに感謝します。本書のベータ版を発売する前は、こんなにたくさんの方が本書を購入して下さるとは思いませんでした。翻訳者一同、本当に感謝しています。また、「本書を読んだおかげで Rails のテスト力が上がった」なんていう声があちこちから聞こえてくることを楽しみにしています。ぜひ、ご自身の Twitter やブログ等で感想を聞かせて下さい。ご意見やご質問でも構いません。本書は引き続きバージョンアップを繰り返していく予定です。「読み終わったからこれでおしまい」ではなく、今後もまたみなさんと紙面で（画面で？）再会できることを楽しみにしています。ではそのときまで、ごきげんよう。

翻訳者一同

[31] https://twitter.com/joker1007
[32] https://twitter.com/shishi4tw
[33] https://twitter.com/ruzia
[34] http://nishiwaki-higashinadarb.doorkeeper.jp/
[35] https://twitter.com/maikeruhorando

Everyday Rails について

Everyday Rails は Ruby on Rails に関する Tips やアイデアを紹介するブログです。あなたのアプリケーション開発に役立つ素晴らしいツールやテクニック等も紹介しています。Everyday Rails の URL はこちらです。https://everydayrails.com/

著者について

Aaron Sumner は20年以上 Web アプリケーションを開発しています。その間彼は CGI を AppleScript で（本当です）、Perl で、PHP で、そして Ruby と Rails で作ってきました。仕事を終えてテキストエディタの前から離れると、Aaron は写真や野球（Cardinals を応援しています）、カレッジスポーツ（カンザス大学 Jayhawks のファンです）、アウトドアクッキング、木工制作、ボーリングなどを楽しんでいます。彼は妻の Elise と5匹の猫、それに1匹の犬と一緒に、オレゴン州のアストリアに住んでいます。

Aaron の個人ブログは https://www.aaronsumner.com/ です。「Everyday Rails - RSpec による Rails テスト入門」（原題: *Everyday Rails Testing with RSpec* ）は彼が書いた最初の本です。

訳者紹介

伊藤淳一

株式会社ソニックガーデン[36]に勤務する Rails プログラマ。プログラミングスクール「フィヨルドブートキャンプ[37]」のメンターでもある。ブログ[38]やQiita[39]などで公開したプログラミング関連の記事多数。著書に「プロを目指す人のための Ruby 入門[40]」（技術評論社）がある。Twitter アカウントは@jnchito[41]。

[36] https://www.sonicgarden.jp
[37] https://bootcamp.fjord.jp/
[38] https://blog.jnito.com
[39] https://qiita.com/jnchito
[40] https://gihyo.jp/book/2021/978-4-297-12437-3
[41] https://twitter.com/jnchito

カバーの説明

カバーで使用した実用的で信頼性の高そうな赤いピックアップトラックの写真[42]はiStockphoto の投稿者であるHabman_18[43]によって撮影されたものです。私は長い時間をかけて（もしかすると長すぎたかも）カバー用の写真を探しました。私がこの写真を選んだ理由は、この写真が Rails のテストに対する私の姿勢を表していると思ったからです。つまり、どちらも派手ではなく、目的地に到達する最速の手段になるとは限りませんが、頑丈で頼りになります。そして、この車は Ruby のような赤色です。いや、もしかすると緑色の方が良かったかもしれません。スペックがパスするときの色みたいに。むむむ。

[42] http://www.istockphoto.com/stock-photo-16071171-old-truck-in-early-morning-light.php?st=1e7555f
[43] https://www.istockphoto.com/jp/portfolio/Habman_18

変更履歴

2024/01/09

- Rails 7.1および RSpec Rails 6.1に対応。
- 推奨 Ruby バージョンを3.3.0に変更。
- サンプルアプリケーションを Rails 7.1で作り直したため、リポジトリ URL を変更。
- 新しいサンプルアプリケーションのコードや挙動と一致するように本書の記述を修正。
- リンク切れしていたいくつかのリンクを新しい URL に修正。

2023/08/06

- Webdrivers gem が Chrome 115以降をサポートしなくなったため、Webdrivers の代わりに selenium-webdriver の ChromeDriver 自動ダウンロード機能を使うように本文の説明とサンプルコードを修正。（第6章および第10章）
- selenium-webdriver の ChromeDriver の自動ダウンロード機能は Ruby 3.0以上が必須であるため、本書の動作確認バージョンも Ruby 3.0以上に変更。（第1章）
- 上記の修正にあわせて「本書執筆時点」の記述を2023年8月に変更。

2023/04/01

- Relish の閉鎖に伴い、「Rails のテストの関するさらなる情報源」にあった RSpec の公式ドキュメントの説明文と URL を変更。

2023/03/03

- 第10章の「メール送信をテストする」にあった誤字を修正。

2023/01/05

- サンプルアプリケーションの Ruby バージョンを3.2.0にアップデートしたことに伴い、本書内に記述していた Ruby のバージョンも3.2.0に変更（バージョン番号の変更のみで、サンプルコードや本文の修正はなし）。
- これにあわせて「本書執筆時点」の記述を2023年1月に変更。

2022/11/03

- サンプルアプリケーションの gem をアップデートしたことに伴い、本書内に記述していた RSpec Rails のバージョンを6.0に、RSpec 本体のバージョンを3.12に変更（バージョン番号の変更のみで、サンプルコードや本文の修正はなし）。
- これにあわせて「本書執筆時点」の記述を2022年11月に変更。
- 訳者紹介にあったリンクの設定ミスを修正。

2022/07/30

- 第11章の「レッドからグリーンへ」にあった誤字を修正。

2022/05/11

- 第8章の「サポートモジュール」にあった出力結果を一部修正。加えて訳注を追記。
- 第8章の「カスタムマッチャ」にあった記述ミスを修正。
- 第9章の「タグ」で `filter_run` と `run_all_when_everything_filtered` を組み合わせる代わりに、RSpec 3.5から追加された `filter_run_when_matching` を使うように変更。
- 第11章に残っていた `login_as` を `sign_in` に修正。

2022/04/20

- サンプルアプリケーションを importmap-rails に移行したことに伴い、第1章のセットアップ手順を一部修正。

2022/03/05

- 第10章の「ファイルアップロードのテスト」にあった、ファイルを自動的に削除するコードを修正。
-「Rails のテストに関するさらなる情報源」にあった、日本語版 Better Specs に関する記述を削除（日本語版ページが削除されていたため）。

2022/02/07

- 第5章の「GET リクエストをテストする」にあった誤字を修正。
- 第11章の「外から中へ進む（Going outside-in）」にあったサンプルコードの記述ミスを修正。

2022/01/28

- 第5章の「ユーザー入力のエラーをテストする」にあったサンプルコードの記述ミスを修正。

2022/01/17

- 日本語版独自のアップデートを実施。Rails 7.0および RSpec Rails 5.0に対応。
- その他、具体的な修正点については「日本語版独自のアップデート内容について」を参照。
- 伊藤淳一、秋元利春、魚振江の3人体制から、伊藤淳一のみへ翻訳体制を変更。

2021/11/12

- 第3章の「インスタンスメソッドをテストする」にあったサンプルコードの記述ミスを修正。

2020/03/07

- 第1章に日本語版独自の補足説明として「サンプルアプリケーションに関する補足説明」を追記。

2019/10/01

- 第10章にあったジェネレータコマンドの脱字を修正。

2019/09/20

- 第4章の「アプリケーションにファクトリを追加する」の項に、ファクトリの定義方法に関する訳注を追記。

2019/09/05

- 第4章の「Factory Bot をインストールする」の項に訳注を追記。

2019/04/08

原著の2019/04/07版に追従。具体的には以下の内容を修正。
第6章

- `chromedriver-helper` の代わりに `webdrivers` を使うように説明内容を変更。

2019/02/25

- 第3章にあった軽微な誤字を修正。

2019/01/07

- 第11章にあった軽微な脱字を修正。

2018/09/12

- 第4章に訳注（原文は「二つ目」になっていますが、「一つ目」が正だと思われます）を追加。

2018/09/06

原著の2018/08/22版に追従。
新規追加

- 付録 A「システムスペックに移行する」を追加。

全般（GitHub のサンプルコード）

- README にあった”work in progress”（作成中）の文言を削除。
- Ruby 2.5で表示されていたシンタックスエラーを修正するために Devise をアップデート（サンプルコードへの影響はなし）。

第8章

- Warden が提供している `login_as` ヘルパーを、Devise が提供している `sign_in` ヘルパーに変更。

第10章

- geocoder gem が IP ベースのジオコーディングに IPInfo.io をデフォルトで使うように変更されたことに伴い、ジオコーディングのコード例を修正。
- Warden が提供している `login_as` ヘルパーを、Devise が提供している `sign_in` ヘルパーに変更。

第11章

- Warden が提供している `login_as` ヘルパーを、Devise が提供している `sign_in` ヘルパーに変更。

2018/08/02

- 第8章にあった typo を修正。

2018/07/11

- 第6章にあった軽微なフォーマット問題を修正。

2018/07/10

原著の2018-06-04版に追従。具体的には以下の内容を修正。

第3章

- `be_empty` マッチャ周辺の説明を変更（原著で発生していた Leanpub のフォーマット問題を回避するため）

第4章

- Factory Girl を Factory Bot に変更（本書全体）

第6章

- chromedriver をインストールするために chromedriver-helper を使用するように変更

第8章

- Warden のヘルパーメソッドではなく、Devise の feature helper を使用するように変更

第10章

- メール送信の統合テストに関する説明を修正
- `receive_message_chain` に関する説明を改善

第11章

- `focus` と `focus: true` の関係性について、説明を追加
- プロジェクトモデルの `completed?` メソッドに関する説明を改善
- モデルやコントローラの外部に置いたロジックのテストに関する説明を追加

その他

- 軽微な記述間違いの修正
- 原著に反映された訳注の削除

2018/03/31

- 翻訳の見直し・改善
- 誤字脱字の修正
- シンタックスハイライトや行番号のフォーマット修正

2018/02/21

- Rails 5.1 + RSpec 3.6版に全面改訂。（原著の2017/11/27版に追従）

2017/05/05

- 第5章の「整理」で提示したコードを実装コードと一致するように修正。

2017/04/29

- 第4章と第5章にあった誤記を修正。

2017/03/21

- 「追加コンテンツ「RSpec ユーザのための Minitest チュートリアル」について」にあった typo を修正。

2016/10/10

- 第5章の細かい文章表現を改善。

2016/06/08

- 第10章にあった typo を修正。

2015/09/15

- 第9章にあった typo を修正。

2015/06/30

- 追加コンテンツ「RSpec ユーザのための Minitest チュートリアル」に関する説明を追加。

2014/12/29

- 原著の2014-12-19版に追従。
  - binstub を使用するサンプルコードの修正
  - DatabaseCleaner に関する説明をアップデート（第8章）
  - タグに関する情報を追加（第9章）
  - `pending` から `skip` への変更とその理由の説明（第9章）
  - ファイルアップロードのサンプルコードを修正（第10章）
  - API のテストで `have_http_status` マッチャを使うように変更（第10章）

  - `rails scaffold` を使用する際に、不要な assets やヘルパーを作成しない方法を説明（第11章）
  - 情報源リストのアップデート
  - その他、細かい文章の改善

2014/11/23

- 第3章にあった軽微な翻訳ミスを修正。

2014/10/24

- RSpec 3.x と Rails 4.1に対応したメジャーアップデート版を公開。
- 第10章に外部サービスのテスト、API のテスト等を加筆.
- RSpec 2.99に関する章を削除。（必要であれば一つ前の版をダウンロードしてください。）
- その他、細かい情報のアップデートや表現の修正等を実施。

2014/07/17

- iPad 版の Kindle で表示したときに、鍵マークやエクスクラメーションマークのアイコンが大きく表示される問題を修正。

2014/05/22

- 第12章「RSpec 3に向けて」を新しく追加。

2014/04/22

- 第4章にあった軽微な誤字を修正。

2014/04/17

- 第5章にあった軽微な誤字を修正。

2014/02/28

- 正式版第1版作成。
- 「サンプル」の訳を「example」に変更。
- 「共有サンプル」の訳を「shared examples」に変更。
- 「テストの主語」となっていた箇所を「テストの対象」に変更。
- 訳者あとがき、日本語版の謝辞、および訳者紹介のページを追加。
- 翻訳全体に関して、日本語としての読みやすさを改善。

- 誤字脱字、表記の揺れ、フォーマット崩れ、段落先頭の字下げの文字数、シンタックスハイライトの不一致等を修正。
- 原著に合わせる形でサンプルコードに行番号を表示（表示されていない部分は原著通り）。
- 原著の2014-02-24版に追従。
  - `selenium-webdriver` gem のバージョンを 2.35.1 から 2.39 に変更（第2章）。
  - 場所の重要性が本書の後半で重要になる理由を追記（第3章）。
  - `eq` と `include` が定義されている gem に関する情報の誤りを修正（第3章）。
  - `before` メソッドの初期値に関する説明を追加（第3章）。
  - `/contacts/:contact_id/phones/:id` のパスが `phone` になっていたミスを修正（第5章）。
  - カスタムマッチャの定義例に関するリンクを変更（第7章）。
  - Selenium の依存関係に関する説明を追記（第8章）。
  - Guard を使った CSS コンパイルが Sass や LESS を指していることを追記（第9章）。
  - モックのサンプルコードで `before` ブロックを2回記述してたミスを修正（第9章）。
  -「古い習慣に戻らないでください！」のセクションで抜けていた後半部分の記述を追加（第12章）。
  - Relish が古いバージョンの RSpec をサポートしなくなったため、「Relish の RSpec ドキュメント」のセクションにあった後半の記述を削除（Rails のテストの関するさらなる情報源）。
  - いくつかのサンプルコードにおいて小規模なリファクタリングを実施。

2014/02/10

- *rspec-rails* の typo を修正（第2章）。
- PDF で見た場合に、一部の文章がページの外にはみ出してしまう問題を修正（第2章、および Rails のテストの関するさらなる情報源）。

2014/02/07

- ベータ版第1版作成（原著の2013/10/07版を翻訳）。