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

謝辞

> この謝辞は2017年の原著改訂時に作成されたものです。

まず最初に why the lucky stiff 氏に感謝します。彼が今どこにいるのかはわかりませんが、彼のちょっと奇妙で面白いプロジェクトと著書のおかげで私は Ruby に出会うことができました。Railscasts[4]を制作してくれた Ryan Bates にも感謝します。彼は誰よりも詳しく私に Rails を教えてくれました。彼らなしでは今の Ruby コミュニティはあり得なかったと思います。

まだお目にかかったことのない Ruby コミュニティの偉大なエンジニアにも感謝します。あなた方のおかげで私は開発者として成長することができました。ただし、必ずしも私のコードに活かされているとは限らないですが。

私が Everyday Rails blog に書いた RSpec 関連の記事に素晴らしいフィードバックを送ってくれた読者のみなさんにも感謝します。フィードバックをいただいたおかげで、本書の内容がより正確なものになりました。本書が発行されてまもない頃に購入してくれたみなさんにも感謝します。本書の反響はびっくりするぐらい大きくて、みなさんからいただいたフィードバックは私にとって大変有益なものでした。

David Gnojek 氏は私が本書のために作成した十数件のカバーデザインに対して意見をくれました。おかげで良いカバーデザインを選ぶことができました。どうもありがとうございます。ぜひDESIGNOJEK[5]というサイトにある Dave の素晴らしいアートとデザインをチェックしてみてください。

本書を中国語と日本語に翻訳してくれた Andor Chen 氏、伊藤淳一氏、秋元利春氏、魚振江氏にも感謝します。彼らががんばってくれたおかげで、数え切れないぐらいたくさんの人たちに本書を読んでもらえました。これは私一人ではできなかったことで、とても嬉しく思っています。

このプロジェクトを応援してくれた家族と友人にも感謝します。もしかすると私が何を話しているのか、さっぱりわからなかったかもしれませんが。

そして最後に、何か新しいことを始めると止まらなくなる私にずっと我慢してくれた妻に感謝します。日によっては本当に遅い時間まで起きていたり、徹夜で何かしたりしたときも

[4] http://railscasts.com/
[5] http://www.designojek.com/

ありました。そして、そんな中でずっと私と一緒にいてくれた猫たちにも感謝します。

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*[6] 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。

[6] 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[7]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

[7] 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[8]、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

[8] 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

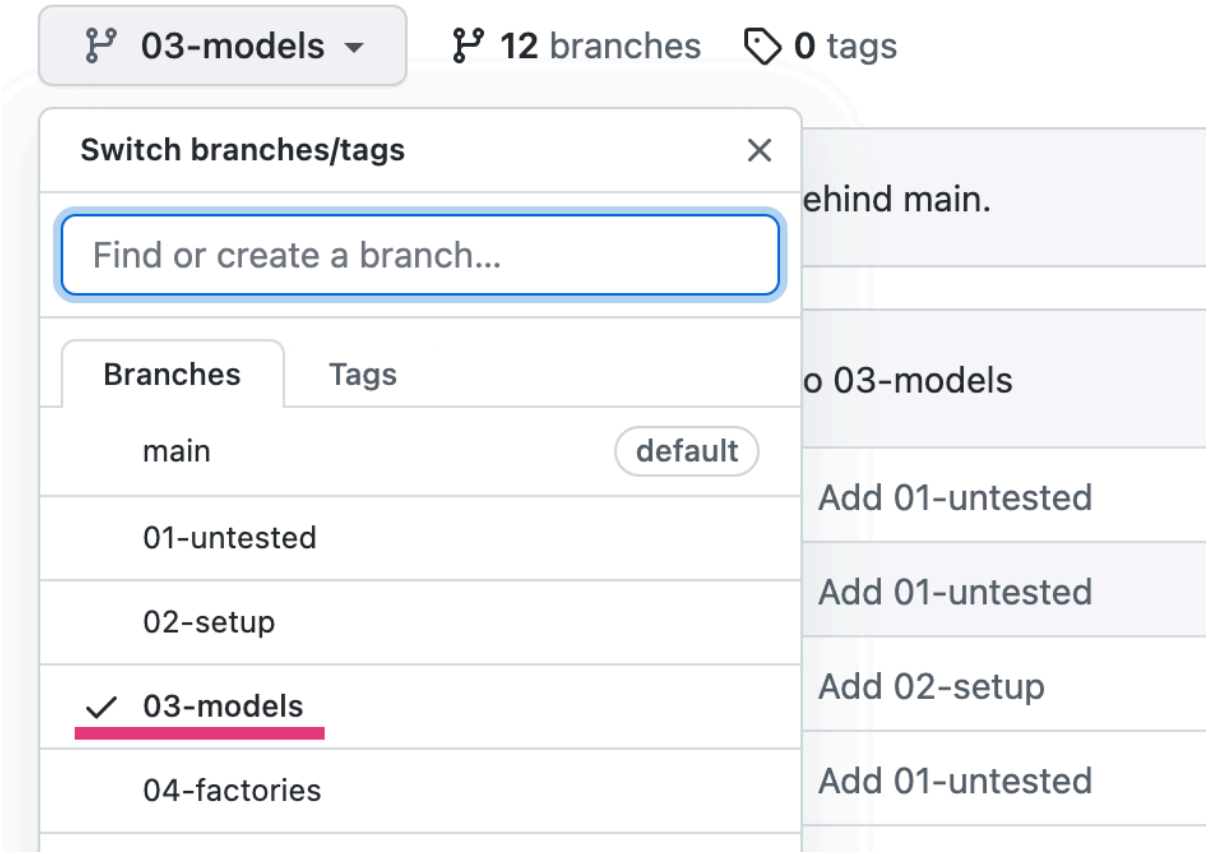
> 2022年に本書のアップデートを行ったタイミングで、日本語版は独自のサンプルコードを提供することになりました。原著のソースコードを参照したい場合は https://github.com/everydayrails/everydayrails-rspec-2017 にアクセスしてください。

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。

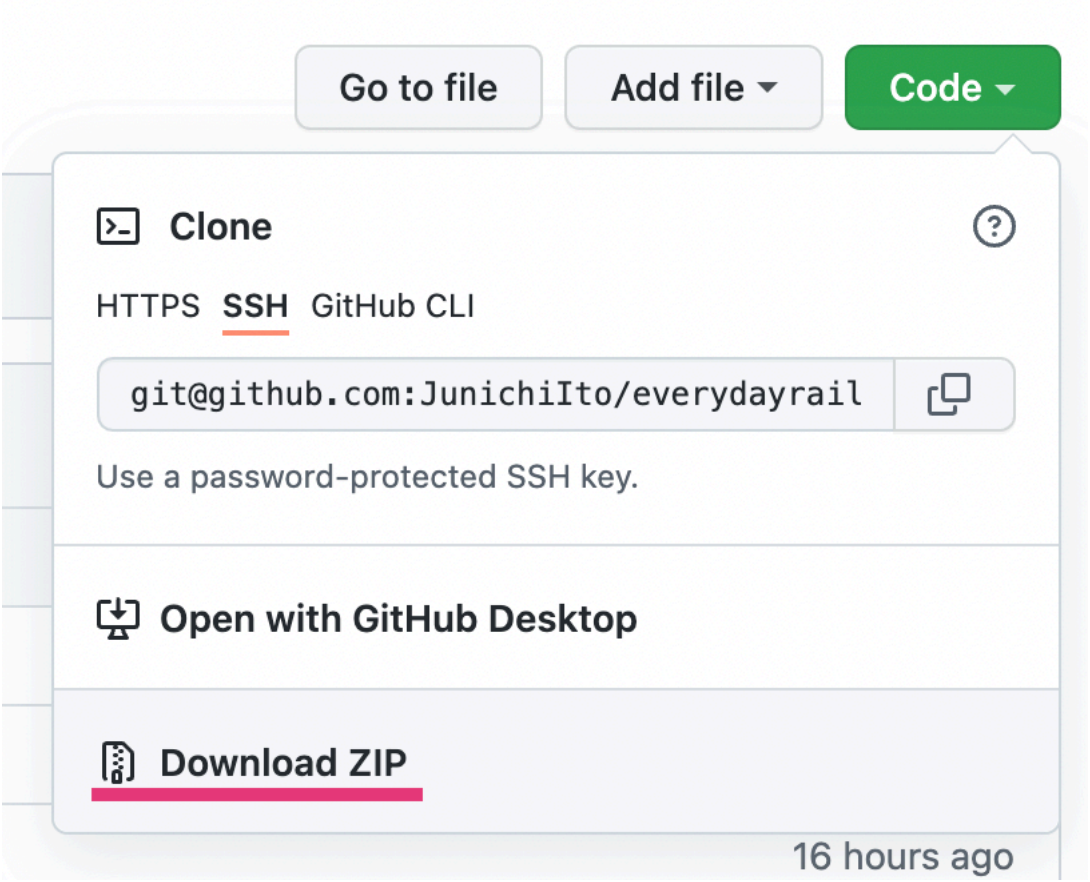
本書はどの章も一つ前の章のソースコードに変更を加えていく流れで構成されています。なので、現在の章のスタート地点として一つ前の章を使うことができます。たとえば、第5章のコードを最初から順に入力していきたいのであれば、第4章のコードから書き始めてください。

```
$ 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[9] 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[10] 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。

[9] http://gitimmersion.com/
[10] 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[11]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[12] 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）

[11] http://learncodethehardway.org/
[12] 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*[13] 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

[13] 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。

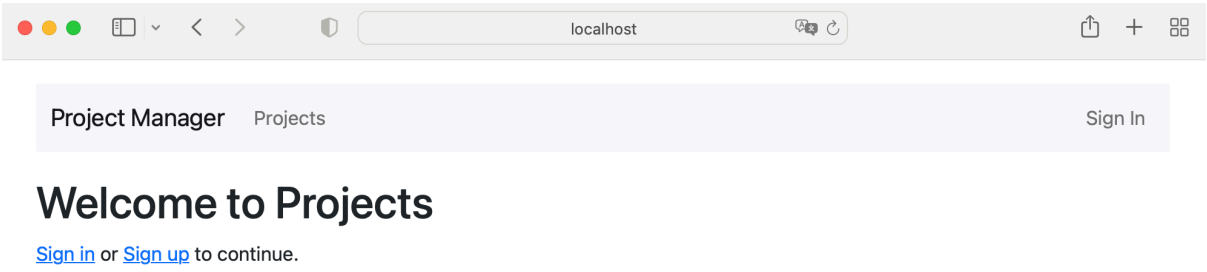
まず、サンプルアプリケーションには以下のツールやソフトウェアが必要です。不足している場合は適宜インストールしてください。

- 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[14] 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[15]で質問していただいて構いません。

[14] https://teratail.com/
[15] 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[16] 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
```

[16] https://github.com/JunichiIto/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[17]」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。

[17] 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[18]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。

[18] 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[19]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

[19] 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[20]から始めてみるのが良いと思います。

[20] 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[21] 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

[21] 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[22]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。

[22] 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[23]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。

[23] 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
```

```
  end
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

```
require 'rails_helper'

RSpec.describe Note, type: :model do

  before do
    # 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
  end

  # バリデーションのテストが並ぶ

  # 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

    before do
      # 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
    end

    # 一致するデータが見つかるとき
    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。

4. 意味のあるテストデータの作成

ここまで私たちはごく普通の Ruby オブジェクト（一般的には *plain old Ruby objects* の略語で *PORO* と呼ばれます）を使ってテスト用の一時データを作ってきました。この方法はシンプルですし、余計な gem を追加する必要もありません。もしあなたのアプリケーションにおいてテストデータを作るのにこの方法で事足りるのであれば、わざわざ余計なものを追加してテストスイートを複雑にする必要はありません。

ですが、テストシナリオが複雑になってもテストデータのセットアップはシンプルな方がいいですよね。複雑なテストシナリオになったときでも、私たちはデータよりもテストにフォーカスしたいと思うはずです。幸いなことに、テストデータを簡単にしてくれる Ruby ライブラリがいくつかあります。この章では有名な gem である *Factory Bot* に焦点を当てます。具体的には次のような内容を説明します。

- 他の方法と比較した場合のファクトリの利点と欠点について説明します。
- それから基本的なファクトリを作り、既存のスペックで使ってみます。
- 続いてファクトリを編集し、さらに便利で使いやすくします。
- さらに Active Record の関連を再現する、より高度なファクトリを見ていきます。
- 最後に、ファクトリを使いすぎるリスクについて説明します。

この章で発生する変更点全体は GitHub 上のこの diff[24] 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-04-factories origin/03-models
```

ファクトリ対フィクスチャ

Rails ではサンプルデータを生成する手段として、フィクスチャと呼ばれる機能がデフォルトで提供されています。フィクスチャは YAML 形式のファイルです。このファイルを使ってサンプルデータを作成します。たとえば、`Project` モデルのフィクスチャなら次のようになります。

[24] https://github.com/JunichiIto/everydayrails-rspec-jp-2024/compare/03-models...04-factories

projects.yml

```yaml
death_star:
  name: "Death Star"
  description: "Create the universe's ultimate battle station"
  due_on: 2016-08-29

rogue_one:
  name: "Steal Death Star plans"
  description: "Destroy the Empire's new battle station"
  due_on: 2016-08-29
```

それからテストの中で `projects(:rogue_one)` と呼び出すだけで、全属性がセットされた新しい `Project` が使えるようになります。とても素晴らしいですね。

フィクスチャを好む人もたくさんいます。フィクスチャは比較的速いですし、Rails に最初から付いてきます。あなたのアプリケーションやテストスイートに余計なものを追加する必要がありません。

とはいえ、私が初心者だった頃はフィクスチャで困ったことを覚えています。私は実行中のテストでどんなデータが作成されたのかを見たかったのですが、フィクスチャを使うとテストとは別のフィクスチャファイルに保存された値を覚えておく必要がありました。実際、私は今でもテストデータはすぐに確認できる方が好きです。テストデータのセットアップはテストの一部として目に見える方がいいですし、テストが実行されたときに何が起きるのかも理解しやすくなります。

フィクスチャにはもろくて壊れやすいという性質もあります。これはすなわち、テストコードやアプリケーションコードを書くのと同じぐらいの時間をテストデータファイルの保守に時間をかけなくてはならない、ということを意味しています。最後に、Rails はフィクスチャのデータをデータベースに読み込む際に Active Record を使いません。これはどういうことでしょうか？これはつまり、モデルのバリデーションのような重要な機能が無視されるということです。私に言わせれば、これは望ましくありません。なぜなら、本番環境のデータと一致しなくなる可能性があるからです。もし同じデータを Web フォームやコンソールから作ろうとすると、失敗することもあるわけです。これは困りますよね？

このような理由からテストデータのセットアップが複雑になってきたときは、私は ファクトリ を使っています。ファクトリはシンプルで柔軟性に富んだテストデータ構築用のブロックです。もし私がテストを理解するのに役立ったコンポーネントを一つだけ挙げなければならないとしたら、Factory Bot[25]を挙げると思います。最初は少しトリッキーで理解しにくいか

[25] https://github.com/thoughtbot/factory_bot

もしれませんが、実際に使って一度基本を覚えれば比較的シンプルに使うことができます。

ファクトリは適切に（つまり賢明に）使えば、あなたのテストをきれいで読みやすく、リアルなものに保ってくれます。しかし、多用しすぎると遅いテストの原因になります（と、お利口で声の大きい Rubyist たちが言っています[26]）。彼らが言うことはわかりますし、気軽にファクトリを使うとスピードの面で高コストになる、というのも理解できます。ですが、それでも私は、遅いテストは何もテストがない状態よりもずっと良いと信じています。特に初心者にとっては絶対そうだと思います。あなたがテストスイートを作りあげ、テストに慣れてきたと思えるようになれば、あなたはいつでもファクトリをもっと効率の良いアプローチに置き換えることができます。

Factory Bot をインストールする

Factory Bot は新しく追加する必要があるテストツールです。というわけで、`Gemfile` の `rspec-rails` の下に gem を追加しましょう。

Gemfile

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

  gem "rspec-rails"
  gem "factory_bot_rails"
end
```

> 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 *factory_bot_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 *factory_bot_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 *factory_bot_rails* がインストールされます。

それから `bundle` コマンドをコマンドラインから実行して、gem をインストールしてください。gem の名前を見ればわかると思いますが、ここで実際にインストールしたのは *factory_bot_rails* です。この gem は Factory Bot に Rails 向けの統合機能を持たせたものです。rspec-rails が RSpec を Rails で便利に使えるようにしているのと同じ関係ですね。Rails 以外の Ruby コードで Factory Bot を単体で使うこともできますが、この内容は本書の範疇を超えるので省略します。

[26] https://groups.google.com/forum/?fromgroups#!topic/rubyonrails-core/_lcjRRgyhC0

せっかくなので、これからジェネレータを使って作られるモデルに対して、自動的にファクトリを作成するように Rails を設定しておきましょう。*confing/application.rb* に戻り、`fixtures: false` の行を削除してください。`config.generators` のブロックは次のようになるはずです。

config/application.rb

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

> 日本語版のサンプルアプリケーションでは第2章で `g.factory_bot false` の行を追加していました。この行も不要になるので一緒に削除してください。

アプリケーションにファクトリを追加する

factory_bot_rails が提供する統合機能のひとつが、新しいファクトリを作成するためのコード生成機能です。これを使って User モデルのファクトリを追加してみましょう。

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

このコマンドを実行すると、*spec* ディレクトリ内に *factories* という新しいディレクトリが作られます。そしてその中には *users.rb* という名前のファイルが以下のような内容で作られます。

spec/factories/users.rb

```
FactoryBot.define do
  factory :user do

  end
end
```

ではここに不足している情報を追加して新しく作ったファクトリを便利にしましょう。追加するデータは第3章 から持ってきます。

spec/factories/users.rb

```
FactoryBot.define do
  factory :user do
    first_name { "Aaron" }
    last_name  { "Sumner" }
    email { "tester@example.com" }
    password { "dottle-nouveau-pavilion-tights-furze" }
  end
end
```

データを囲んでいる中括弧（{ }）は Ruby のブロックです。FactoryBot 4.11から記法が変わり、データを囲む中括弧が必須になりました[27]。ですので、この中括弧を忘れないようにしてください。

こうすると、テスト内で `FactoryBot.create(:user)` と書くだけで、簡単に新しいユーザーを作成できます。ユーザーの名前は *Aaron Sumner* です。メールアドレスは *tester@example.com* です。

このコードを書くとスペック全体で ファクトリ が使えるようになります。`FactoryBot.create(:user)` を使ってテストデータを作れば、そのユーザーの名前は毎回基本的に *Aaron Sumner* になります。メールアドレスやパスワードも最初から設定された状態になります。

この例の属性はすべて文字列ですが、ファクトリで使えるのは文字列だけではありません。整数やブーリアン、日付など、属性に渡せるものなら何でも渡すことができます。詳しい話はまたあとで説明します。

ではセットアップがちゃんと完了していることを確認しましょう。一つ前の章で作成した *user_spec.rb* ファイルに戻り、次のような簡単な example を追加してください。

[27] https://qiita.com/jnchito/items/81637bbdf66c2662eacf

spec/models/user_spec.rb

```
require 'rails_helper'

describe User do
  # 有効なファクトリを持つこと
  it "has a valid factory" do
    expect(FactoryBot.build(:user)).to be_valid
  end

  # 他のスペックが並ぶ ...
end
```

ここではFactoryBot.buildを使ったので、新しいユーザーはインスタンス化されるだけで、保存はされません。ユーザーの属性値はファクトリによって設定されます。それから第3章で説明したbe_validマッチャを使ってユーザーの有効性をテストしています。一つ前の章で作ったスペックと比較してみましょう。このときはPOROを使ってテストデータを作成していました。

```
# 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" do
  user = User.new(
    first_name: "Aaron",
    last_name:  "Sumner",
    email:      "tester@example.com",
    password:   "dottle-nouveau-pavilion-tights-furze",
  )
  expect(user).to be_valid
end
```

ファクトリ版の方がより簡潔ですね。なぜなら詳細が別のファイルに移動しているからです。一方、PORO版はより自己文書的（self-documenting）です。テストをパスさせるのに何が必要になるのかが目で見てわかります。

実際にはどちらのアプローチにもそれぞれ適切なユースケースがあります。詳細な情報が必要になるときもあれば、詳細は隠蔽しても問題ないときもあります。テストの経験が増えるにつれ、どちらのテクニックをいつ使うべきか、感覚的にわかるようになるはずです。ですが、ここでは新しいツールの説明のためにファクトリを使い続けることにします。

第3章で作ったUserバリデーションのスペックをもう一度見てみましょう。今度はファク

トリから作られたデータの属性を一つ以上オーバーライドします。ただし、オーバーライドするのは特定の属性だけです。

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 = FactoryBot.build(:user, first_name: nil)
  user.valid?
  expect(user.errors[:first_name]).to include("can't be blank")
end

# 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 = FactoryBot.build(:user, last_name: nil)
  user.valid?
  expect(user.errors[:last_name]).to include("can't be blank")
end

# 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" do
  user = FactoryBot.build(:user, email: nil)
  user.valid?
  expect(user.errors[:email]).to include("can't be blank")
end
```

ここに挙げた example はとても単純です。見ればわかる通り、ここではすべて Factory Bot の `build` メソッドを使って新しい（ただし保存されていない）`User` を作成しています。最初の example では `first_name` が空になっている User を `user` 変数にセットしています。二番目の例でも同様にデフォルトの `last_name` を *nil* で置き換えています。User モデルは `first_name` と `last_name` の両方を必須としているので、どちらの example でもエラーになるのが期待されるテストの結果です。`email` のバリデーションのテストも全く同じパターンになっています。

次に示すスペックは少し違います。ですが、基本的なツールを使う点は同じです。ここでは `first_name` と `last_name` に特定の値をセットして新しい User を構築しています。テスト内では `email` の値が何か知っておく必要はありません。なので、ファクトリが設定した値をそのまま使います。それから変数に入れた `user` の `name` メソッドが、期待する文字列を返すことを確認しています。

spec/models/user_spec.rb

```
# 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" do
  user = FactoryBot.build(:user, first_name: "John", last_name: "Doe")
  expect(user.name).to eq "John Doe"
end
```

ですが、次の example ではちょっと新しい内容が出てきます。

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
  FactoryBot.create(:user, email: "aaron@example.com")
  user = FactoryBot.build(:user, email: "aaron@example.com")
  user.valid?
  expect(user.errors[:email]).to include("has already been taken")
end
```

この example ではテストオブジェクトの email 属性が重複しないことを確認しています。これを検証するためには二つ目（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）の User がデータベースに保存されている必要があります。そこでエクスペクテーションを実行する前に、FactoryBot.create を使って同じメールアドレスの user を最初に保存しているのです。

次のことを覚えてください。FactoryBot.build を使うと新しいテストオブジェクトをメモリ内に保存します。FactoryBot.create を使うとアプリケーションのテスト用データベースにオブジェクトを永続化します。

シーケンスを使ってユニークなデータを生成する

現在の User ファクトリにはちょっと問題があります。もし、先ほどの example を次のように書いていたら何が起きると思いますか？

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
  FactoryBot.create(:user)
  user = FactoryBot.build(:user)
  user.valid?
  expect(user.errors[:email]).to include("has already been taken")
end
```

うーん、このテストもやっぱりパスしますね。なぜなら、明示的に異なる値を設定しない限り、ファクトリが常にユーザーのメールアドレスを *tester@example.com* に設定するからです。これはここまでに書いてきたスペックでは問題になっていませんが、ファクトリで複数のユーザーをセットアップする必要が出てきた場合は実際のテストコードが走る前に例外が発生します。たとえば以下のような場合です。

```
# 複数のユーザーで何かする
it "does something with multiple users" do
  user1 = FactoryBot.create(:user)
  user2 = FactoryBot.create(:user)
  expect(true).to be_truthy
end
```

すると次のようなバリデーションエラーが発生します。

```
Failures:

  1) User does something with multiple users
     Failure/Error: user2 = FactoryBot.create(:user)

     ActiveRecord::RecordInvalid:
       Validation failed: Email has already been taken
```

Factory Bot では シーケンス を使ってこのようなユニークバリデーションを持つフィールドを扱うことができます。シーケンスはファクトリから新しいオブジェクトを作成するたびに、カウンタの値を1つずつ増やしながら、ユニークにならなければいけない属性に値を設定します。ファクトリ内にシーケンスを作成して実際に使ってみましょう。

spec/factories/users.rb

```
FactoryBot.define do
  factory :user do
    first_name { "Aaron" }
    last_name  { "Sumner" }
    sequence(:email) { |n| "tester#{n}@example.com" }
    password { "dottle-nouveau-pavilion-tights-furze" }
  end
end
```

メール文字列に n の値がどのように挟み込まれるかわかりますか？こうすれば新しいユーザーを作成するたびに、*tester1@example.com*、*tester2@example.com* というように、ユニークで連続したメールアドレスが設定されます。

ファクトリで関連を扱う

ちょっと面倒なユニークなメールアドレスの要件は別にしても、ユーザーファクトリはそれほど難しいことをしていません。実際、ここまでに紹介したような使い方だとわざわざ PORO のかわりにファクトリを使うというのはあまり魅力的な選択肢ではないかもしれません。ですが、Factory Bot は他のモデルと関連を持つモデルを扱うのにとても便利です。というわけで、メモとプロジェクトのファクトリを作ってみましょう。ファクトリの作成はジェネレータを使います。`bin/rails g factory_bot:model note` のコマンドで始めてください。

spec/factories/notes.rb

```
FactoryBot.define do
  factory :note do
    message { "My important note." }
    association :project
    association :user
  end
end
```

次に `bin/rails g factory_bot:model project` でプロジェクトのファクトリを作成します。

spec/factories/projects.rb

```
FactoryBot.define do
  factory :project do
    sequence(:name) { |n| "Project #{n}" }
    description { "A test project." }
    due_on { 1.week.from_now }
    association :owner
  end
end
```

先へ進む前に、ユーザーファクトリにちょっとだけ情報を追加しましょう。ファクトリに名前を付けている2行目に、以下に示すような *owner* という別名（alias）を付けてください。なぜこうする必要があるのかはすぐにわかります。

spec/factories/users.rb

```
FactoryBot.define do
  factory :user, aliases: [:owner] do
    first_name { "Aaron" }
    last_name  { "Sumner" }
    sequence(:email) { |n| "tester#{n}@example.com" }
    password { "dottle-nouveau-pavilion-tights-furze" }
  end
end
```

メモはプロジェクトとユーザーの両方に属しています。しかし、テストのたびにいちいち手作業でプロジェクトとユーザーを作りたくありません。私たちが作りたいのはメモだけです。こちらの使用例を見てください。

spec/models/note_spec.rb

```
require 'rails_helper'

RSpec.describe Note, type: :model do
  # ファクトリで関連するデータを生成する
  it "generates associated data from a factory" do
    note = FactoryBot.create(:note)
    puts "This note's project is #{note.project.inspect}"
    puts "This note's user is #{note.user.inspect}"
  end
end
```

ここでは Factory Bot を1回しか呼んでいないにもかかわらず、テストの実行結果を見ると必要なデータが全部作成されています。

```
Note
This note's project is #<Project id: 1, name: "Test Project 1",
description: "Sample project for testing purposes", due_on:
"2017-01-17", created_at: "2017-01-10 04:01:24", updated_at:
"2017-01-10 04:01:24", user_id: 1>
This note's user is #<User id: 2, email: "tester2@example.com", created_at: "2017\
-01-10 04:01:24", updated_at: "2017-01-10 04:01:24",
first_name: "Aaron", last_name: "Sumner">
```

ですが、この例はファクトリで関連を扱う際の潜在的な落とし穴も示しています。みなさんはわかりますか？ユーザーのメールアドレスをよく見てください。なぜ *tester1@example.com* ではなく、*tester2@example.com* になっているのでしょうか？この理由はメモのファクトリが関連するプロジェクトを作成する際に関連するユーザー（プロジェクトに関連する owner）を作成し、それから2番目のユーザー（メモに関連するユーザー）を作成するからです。

この問題を回避するためにメモのファクトリを次のように更新します。こうするとデフォルトでユーザーが1人しか作成されなくなります。

spec/factories/notes.rb

```
FactoryBot.define do
  factory :note do
    message { "My important note." }
    association :project
    user { project.owner }
  end
end
```

スペックの結果を見てもユーザーは1人だけです。

```
Note
This note's project is #<Project id: 1, name: "Test Project 1", description: "Sam\
ple project for testing purposes", due_on: "2017-01-17", created_at: "2017-01-10 \
04:18:03", updated_at: "2017-01-10 04:18:03", user_id: 1>
This note's user is #<User id: 1, email: "tester1@example.com", created_at: "2017\
-01-10 04:18:03", updated_at: "2017-01-10 04:18:03", first_name: "Aaron", last_na\
me: "Sumner">
```

ここで確認してほしいのは、ファクトリを使うとたまにびっくりするような結果が生まれるということです。また、あなたの予想よりも多いテストデータが作成されることもあります。このセクションで使ったようなちょっと不自然なテストコードであれば大した問題になりませんが、ユーザーの件数を検証するようなテストを書いているときはデバッグするときに原因を突き止めるのに苦労するかもしれません。

さて、ちょっと前に User ファクトリに追加した *alias* に戻りましょう。*Project* モデルを見ると、User の関連は *owner* という名前になっているのがわかると思います。

app/models/project.rb

```
class Project < ApplicationRecord
  validates :name, presence: true, uniqueness: { scope: :user_id }

  belongs_to :owner, class_name: User, foreign_key: :user_id
  has_many :notes, dependent: :destroy
  has_many :tasks, dependent: :destroy
end
```

このように Factory Bot を使う際はユーザーファクトリに対して owner という名前で参照される場合があると伝えなくてはいけません。そのために使うのが alias です。alias の付いたユーザーファクトリのコード全体を再度載せておきます。

spec/factories/users.rb

```
FactoryBot.define do
  factory :user, aliases: [:owner] do
    first_name { "Aaron" }
    last_name  { "Sumner" }
    sequence(:email) { |n| "tester#{n}@example.com" }
    password { "dottle-nouveau-pavilion-tights-furze" }
  end
end
```

ファクトリ内の重複をなくす

Factory Bot では同じ型を作成するファクトリを複数定義することもできます。たとえば、スケジュールどおりのプロジェクトとスケジュールから遅れているプロジェクトをテストしたいのであれば、別々の名前を付けてプロジェクトファクトリの引数に渡すことができます。その際はそのファクトリを使って作成するインスタンスのクラス名と、既存のファクトリと異なるインスタンスの属性値（この例でいうと due_on 属性の値）も指定します。

spec/factories/projects.rb

```
FactoryBot.define do
  factory :project do
    sequence(:name) { |n| "Test Project #{n}" }
    description { "Sample project for testing purposes" }
    due_on { 1.week.from_now }
    association :owner
  end

  # 昨日が締め切りのプロジェクト
  factory :project_due_yesterday, class: Project do
    sequence(:name) { |n| "Test Project #{n}" }
    description { "Sample project for testing purposes" }
    due_on { 1.day.ago }
    association :owner
  end

  # 今日が締め切りのプロジェクト
  factory :project_due_today, class: Project do
```

```
    sequence(:name) { |n| "Test Project #{n}" }
    description { "Sample project for testing purposes" }
    due_on { Date.current.in_time_zone }
    association :owner
  end

  # 明日が締め切りのプロジェクト
  factory :project_due_tomorrow, class: Project do
    sequence(:name) { |n| "Test Project #{n}" }
    description { "Sample project for testing purposes" }
    due_on { 1.day.from_now }
    association :owner
  end
end
```

こうすると上で定義した新しいファクトリを Project モデルのスペックで使うことができます。ここでは魔法のマッチャ、be_late が登場します。be_late は RSpec に定義されているマッチャではありません。ですが RSpec は賢いので、project に late または late? という名前の属性やメソッドが存在し、それが真偽値を返すようになっていれば be_late はメソッドや属性の戻り値が true になっていることを検証してくれるのです。すごいですね。

spec/models/project_spec.rb

```
# 遅延ステータス
describe "late status" do
  # 締切日が過ぎていれば遅延していること
  it "is late when the due date is past today" do
    project = FactoryBot.create(:project_due_yesterday)
    expect(project).to be_late
  end

  # 締切日が今日ならスケジュールどおりであること
  it "is on time when the due date is today" do
    project = FactoryBot.create(:project_due_today)
    expect(project).to_not be_late
  end

  # 締切日が未来ならスケジュールどおりであること
  it "is on time when the due date is in the future" do
```

```
17      project = FactoryBot.create(:project_due_tomorrow)
18      expect(project).to_not be_late
19    end
20  end
```

ですが、新しく作ったファクトリには大量の重複があります。新しいファクトリを定義するときは毎回プロジェクトの全属性を再定義しなければいけません。これはつまり、Project モデルの属性を変更したときは毎回複数のファクトリ定義を変更する必要が出てくる、ということを意味しています。

Factory Bot には重複を減らすテクニックが二つあります。一つ目は ファクトリの継承 を使ってユニークな属性だけを変えることです。

spec/factories/projects.rb

```
1   FactoryBot.define do
2     factory :project do
3       sequence(:name) { |n| "Test Project #{n}" }
4       description { "Sample project for testing purposes" }
5       due_on { 1.week.from_now }
6       association :owner
7
8       # 昨日が締め切りのプロジェクト
9       factory :project_due_yesterday do
10        due_on { 1.day.ago }
11      end
12
13      # 今日が締め切りのプロジェクト
14      factory :project_due_today do
15        due_on { Date.current.in_time_zone }
16      end
17
18      # 明日が締め切りのプロジェクト
19      factory :project_due_tomorrow do
20        due_on { 1.day.from_now }
21      end
22    end
23  end
```

見た目には少しトリッキーかもしれませんが、これが継承の使い方です。:project_due_-

yesterday と :project_due_today と :project_due_tomorrow の各ファクトリは継承元となる :project ファクトリの内部で入れ子になっています。構造だけを抜き出すと次のようになります。

```
factory :project
  factory :project_due_yesterday
  factory :project_due_today
  factory :project_due_tomorrow
```

継承を使うと class: Project の指定もなくすことができます。なぜならこの構造から Factory Bot は子ファクトリで Project クラスを使うことがわかるからです。この場合、スペック側は何も変更しなくてもそのままでパスします。

重複を減らすための二つ目のテクニックは トレイト（trait） を使ってテストデータを構築することです。このアプローチでは属性値の 集合 をファクトリで定義します。まず、プロジェクトファクトリの中身を更新しましょう。

spec/factories/projects.rb

```ruby
FactoryBot.define do
  factory :project do
    sequence(:name) { |n| "Test Project #{n}" }
    description { "Sample project for testing purposes" }
    due_on { 1.week.from_now }
    association :owner

    # 締め切りが昨日
    trait :due_yesterday do
      due_on { 1.day.ago }
    end

    # 締め切りが今日
    trait :due_today do
      due_on { Date.current.in_time_zone }
    end

    # 締め切りが明日
    trait :due_tomorrow do
      due_on { 1.day.from_now }
    end
```

```
  end
end
```

トレイトを使うためにはスペックを変更する必要があります。利用したいトレイトを使って次のようにファクトリから新しいプロジェクトを作成してください。

spec/models/project_spec.rb

```
describe "late status" do
  # 締切日が過ぎていれば遅延していること
  it "is late when the due date is past today" do
    project = FactoryBot.create(:project, :due_yesterday)
    expect(project).to be_late
  end

  # 締切日が今日ならスケジュールどおりであること
  it "is on time when the due date is today" do
    project = FactoryBot.create(:project, :due_today)
    expect(project).to_not be_late
  end

  # 締切日が未来ならスケジュールどおりであること
  it "is on time when the due date is in the future" do
    project = FactoryBot.create(:project, :due_tomorrow)
    expect(project).to_not be_late
  end
end
```

トレイトを使うことの本当の利点は、複数のトレイトを組み合わせて複雑なオブジェクトを構築できる点です。トレイトについてはこの後の章でテストデータに関する要件がもっと複雑になってきたときに再度説明します。

コールバック

Factory Bot の機能をもうひとつ紹介しましょう。コールバックを使うと、ファクトリがオブジェクトを create する前、もしくは create した後に何かしら追加のアクションを実行できます。また、create されたときだけでなく、build されたり、stub されたりしたときも同じように使えます。適切にコールバックを使えば複雑なテストシナリオも簡単にセットアップで

きるので、強力な時間の節約になります。ですが、一方でコールバックは遅いテストや無駄に複雑なテストの原因になることもあります。注意して使ってください。

そのことを頭の片隅に置きつつ、コールバックのよくある使い方を見てみましょう。ここでは複雑な関連を持つオブジェクトを作成する方法を説明します。Factory Bot にはこうした処理を簡単に行うための create_list メソッドが用意されています。コールバックを利用して、新しいオブジェクトが作成されたら自動的に複数のメモを作成する処理を追加してみましょう。今回は必要なときにだけコールバックを利用するよう、トレイトの中でコールバックを使います。

spec/factories/projects.rb

```ruby
FactoryBot.define do
  factory :project do
    sequence(:name) { |n| "Test Project #{n}" }
    description { "Sample project for testing purposes" }
    due_on { 1.week.from_now }
    association :owner

    # メモ付きのプロジェクト
    trait :with_notes do
      after(:create) { |project| create_list(:note, 5, project: project) }
    end

    # 他のトレイトが並ぶ ...
  end
end
```

create_list メソッドではモデルを作成するために関連するモデルが必要です。今回はメモの作成に必要な Project モデルを使っています。

プロジェクトファクトリに新しく定義した with_notes トレイトは、新しいプロジェクトを作成した後にメモファクトリを使って5つの新しいメモを追加します。それではスペック内でこのトレイトを使う方法を見てみましょう。最初はトレイトなしのファクトリを使ってみます。

spec/models/project_spec.rb

```
# たくさんのメモが付いていること
it "can have many notes" do
  project = FactoryBot.create(:project)
  expect(project.notes.length).to eq 5
end
```

このテストは失敗します。なぜならメモの数が5件ではなくゼロだからです。

```
Failures:

  1) Project can have many notes
     Failure/Error: expect(project.notes.length).to eq 5

       expected: 5
            got: 0

       (compared using ==)
     # ./spec/models/project_spec.rb:69:in `block (2 levels) in <top
       (required)>'
     # 以下略
```

そこで with_notes トレイトでセットアップした新しいコールバックを使って、このテストをパスさせましょう。

spec/models/project_spec.rb

```
# たくさんのメモが付いていること
it "can have many notes" do
  project = FactoryBot.create(:project, :with_notes)
  expect(project.notes.length).to eq 5
end
```

これでテストがパスします。なぜなら、コールバックによってプロジェクトに関連する5つのメモが作成されるからです。実際のアプリケーションでこういう仕組みを使っていると、ちょっと情報量の乏しいテストに見えるかもしれません。ですが、今回の使用例はコールバックが正しく設定されているか確認するのに役立ちますし、この先でもっと複雑なテストを作り始める前のちょうどいい練習にもなります。とくに、Rails のモデルが入れ子になった他のモデルを属性として持っている場合、コールバックはそうしたモデルのテストデータを作るのに便利です。

ここではFactory Botのコールバックについてごく簡単な内容しか説明していません。コールバックの詳しい使い方についてはFactory Botの公式ドキュメントにあるコールバックの欄[28]を参照してください。本書でもこのあとでさらにコールバックを使っていきます。

ファクトリを安全に使うには

ファクトリはテスト駆動開発で強力なパワーを発揮します。ほんの数行コードを書くだけで、私たちのソフトウェアを検証するために必要なサンプルデータをあっという間に作ってくれます。セットアップ用のコードも短くなるので、テストコードの読みやすさも妨げません。

ですが、他のパワフルなツールと同様に、ファクトリを使うときにはちょっと注意した方がいいと忠告しておきます。前述のとおり、ファクトリを使うとテスト中に予期しないデータが作成されたり、無駄にテストが遅くなったりする原因になります。

上記のような問題がテストで発生した場合はまず、ファクトリが必要なことだけを行い、それ以上のことをやっていないことを確認してください。コールバックを使って関連するデータを作成する必要があるなら、ファクトリを使うたびに呼び出されないよう、トレイトの中でセットアップするようにしましょう。可能な限り`FactoryBot.create`よりも`FactoryBot.build`を使ってください。こうすればテストデータベースにデータを追加する回数が減るので、パフォーマンス面のコストを削減できます。

最初の頃はここでファクトリを使う必要はあるだろうか？と自問するのもいいでしょう。もしモデルの`new`メソッドや`create`メソッドでテストデータをセットアップできるなら、ファクトリをまったく使わずに済ませることもできます。PORO で作ったデータとファクトリで作ったデータをテスト内に混在させることもできます。このように、テストの読みやすさと速さを保つためにはいろんな方法が使えます。

> テストの経験がある程度ある人であれば、テストのスピードを上げる方法として、なぜモックやダブルを説明しないのか、と不思議に思っている人もいるかもしれません。モックを使うとさまざまな種類の複雑さをテストに持ち込むことになります。なので、初心者のうちはここで説明したような方法でデータを作る方が良いと私は考えています。モックやスタブについては本書の後半で説明します。

[28] https://github.com/thoughtbot/factory_bot/blob/main/GETTING_STARTED.md#callbacks

まとめ

この章では Factory Bot を使ってごちゃごちゃしていたスペックをきれいにしました。それだけでなく、データを作成する際の柔軟性も上がり、リアルなシナリオをテストしやすくなりました。この章で説明した機能は私が普段よく使う機能です。こうした機能はみなさんも大半のテストで使うことになるはずです。また、ここで紹介した内容と同じぐらい、たくさんの機能が Factory Bot には用意されています。Factory Bot の基礎を理解したら、ぜひFactory Bot のドキュメント[29]を読んで、たくさんの便利機能を使いこなせるようになってください。

本書ではこのあとの章でも Factory Bot を使っていきます。実際、Factory Bot はこの次に出てくるテストコードでも重要な役割を演じます。さて、次に出てくるコードはコントローラです。コントローラはモデルとビューの間でデータをやりとりするためのコンポーネントです。そしてこのコントローラが次章のメイントピックになります。

演習問題

- 第3章で追加したモデルスペックをもう一度確認してください。ファクトリを使ってきれいにできる箇所が他にもありませんか？ちょっとヒントを出しましょう。現状の *spec/models/project_spec.rb* では User オブジェクトを作成するコードがたくさんあります。かわりにユーザーファクトリを使って書きかえることはできますか？
- もしあなた自身のアプリケーションにまだファクトリを追加していなければ、ファクトリを追加してください。
- あなたが作ったアプリケーションのファクトリを見てください。継承やトレイトを使ってファクトリをリファクタリングできませんか？
- コールバック付きのトレイトを作成してみてください。それから簡単なテストを書いて期待どおりに動いているか確認してください。

[29] https://github.com/thoughtbot/factory_bot/blob/main/GETTING_STARTED.md

5. コントローラスペック

Rails チームについて私がすごいと思うのは、フレームワークでもういらないと思われる機能はどんどん切り捨てていく精神です。Rails 5.0では過剰に使われていた二つのテストヘルパー（訳注：`assigns` メソッドと `assert_template` メソッド）がクビになりました。さらに、コントローラ層のテストが公式に格下げ（正式な用語を使うなら *soft-deprecated* ）されました。

正直にいうと、私はここ数年、自分のアプリケーションであまりコントローラのテストを書いてきませんでした。コントローラのテストはすぐ壊れやすくなりますし、アプリケーション内の他の実装の詳細へ過剰に焦点が当てられることも多いです。

Rails チームと RSpec チームの双方が、コントローラのテスト（機能テスト層とも呼ばれます）を削除するか、またはモデルのテスト（単体テスト）か、より高いレベルの統合テストと置き換えることを推奨しています。こんな話を聞くと気分が滅入ってしまう人もいるかもしれませんが、心配はいりません。この変化は状況を改善するための変化です！みなさんはモデルをテストする方法はすでに理解していますし、統合テストについても次の章以降で説明します。

ですが、コントローラのテストもまったく無視するわけにはいきません。なぜなら、移行期間中はコントローラのテストもちゃんと意味をもって存在しているからです。また、私はなぜ Rails チームがこのレベルのテストを格下げしたのか、その理由を理解することも重要だと考えています。加えて、もしあなたに RSpec でテストされているレガシーな Rails アプリケーションを保守する機会があれば、コントローラスペックを見かけることもきっとあると思います。

この章ではコントローラのテストの基礎について次のようなことを学びます。

- まず、コントローラのテストとモデルのテストの違いを確認します。
- それからコントローラのアクションをいくつかテストします。
- 次に、認証が必要なアクションについて説明します。
- そのあとで、ユーザーの入力をテストします。入力値が不正な場合もテストします。
- 最後に CSV や JSON のような非 HTML の出力を持つコントローラのメソッドをテストします。

この章で発生する変更点全体は GitHub 上のこの diff[30] 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-05-controllers origin/04-factories
```

コントローラスペックの基本

では一番シンプルなコントローラから始めましょう。Home コントローラは一つの仕事しかしません。まだサインインしていない人のために、アプリケーションのホームページを返す仕事です。

app/controllers/home_controller.rb

```
1 class HomeController < ApplicationController
2
3   skip_before_action :authenticate_user!
4
5   def index
6   end
7 end
```

コントローラのテストを作成するために、RSpec が提供しているジェネレータを使います。

```
$ bin/rails g rspec:controller home --controller-specs --no-request-specs
```

> 以前は `bin/rails g rspec:controller home` というコマンドでコントローラのテストが作成できましたが、RSpec Rails 4.0.0以降ではリクエストスペックが優先的に作成されるようになったため、`--controller-specs --no-request-specs` というオプションを付ける必要があります。なお、リクエストスペックについては第7章 で説明します。

こうすると次のような定型コードが作成されます。

[30] https://github.com/JunichiIto/everydayrails-rspec-jp-2024/compare/04-factories...05-controllers

spec/controllers/home_controller_spec.rb

```
require 'rails_helper'

RSpec.describe HomeController, type: :controller do

end
```

ここまでは特に面白いところはありません。実際、ここまではtype: controllerだけがこれまでに作ったモデルスペックとの唯一の違いです。ここにシンプルなテストを追加していきましょう。まず、コントローラがブラウザのリクエストに対して正常にレスポンスを返すことを確認します。このアクション（#index）のためにdescribeブロックを作り、それから新しいスペックをその中に追加してください。

spec/controllers/home_controller_spec.rb

```
require 'rails_helper'

RSpec.describe HomeController, type: :controller do
  describe "#index" do
    # 正常にレスポンスを返すこと
    it "responds successfully" do
      get :index
      expect(response).to be_successful
    end
  end
end
```

responseはブラウザに返すべきアプリケーションの全データを保持しているオブジェクトです。この中にはHTTPレスポンスコードも含まれます。be_successfulはレスポンスステータスが成功（200レスポンス）か、それ以外（たとえば500エラー）であるかをチェックします。

このテストを実行して何が起きるか見てみましょう。ここではコントローラのテストだけを実行したいので、bundle exec rspec spec/controllersを使って実行します。

```
HomeController
  #index
    responds successfully

Finished in 0.01775 seconds (files took 0.3492 seconds to load)
1 example, 0 failures
```

ちょっと退屈ですか？ですが、ここには重要なポイントが含まれています。コントローラのテストは退屈であるべきなのです。なぜなら コントローラ 自体が退屈であるべきだからです。コントローラはいくつかのパラメータをブラウザから受け取り、それを使って面白いことをするオブジェクトに渡します。それからレスポンスをブラウザに返します。何はともあれ、そうあるべきなのです。完璧な世界においては。

テストが正しく機能していることを確認するため、わざとテストを失敗させましょう。最も簡単な方法は `expect(response).to be_successful` の `to` を `to_not` に変えて、レスポンスが成功しないように期待することです。

spec/controllers/home_controller_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe HomeController, type: :controller do
4    describe "#index" do
5      # 正常にレスポンスを返すこと
6      it "responds successfully" do
7        get :index
8        expect(response).to_not be_successful
9      end
10   end
11 end
```

当然、失敗します。

```
Failures:

1) HomeController#index responds successfully
 Failure/Error: expect(response).to_not be_successful
   expected `#<ActionDispatch::TestResponse:0x007f9b16cdd350
   @mon_owner=nil, @mon_count=0,
   @mon_mutex=#<Thread::Mu...:Headers:0x007f9b16cc7230
   @req=#<ActionController::TestRequest:0x007f9b16cdd508 ...>>,
   @variant=[]>>.successful?` to return false, got true
```

特定の HTTP レスポンスコードが返ってきているかどうかも確認できます。この場合であれば200 OK のレスポンスが返ってきてほしいはずです。

spec/controllers/home_controller_spec.rb

```
1 # 200レスポンスを返すこと
2 it "returns a 200 response" do
3   get :index
4   expect(response).to have_http_status "200"
5 end
```

これもちゃんとパスします。

繰り返しになりますが、こんなテストは一見すると退屈に見えます。ですが、面白い要素も実は隠れています。コントローラをもう一度見てください。よく見ると、アプリケーション全体で使われているユーザー認証用の before_action をスキップしていますね。この行をコメントアウトしてテストを実行すると、何が起きるでしょうか？

```
Failures:

      1) HomeController returns a 200 response
         Failure/Error: get :index

         Devise::MissingWarden:
           Devise could not find the `Warden::Proxy` instance on your
           request environment.
           Make sure that your application is loading Devise and Warden
           as expected and that the `Warden::Manager` middleware is
           present in your middleware stack.
           If you are seeing this on one of your tests, ensure that your
           tests are either executing the Rails middleware stack or that
```

```
your tests are using the `Devise::Test::ControllerHelpers`
module to inject the `request.env['warden']` object for you.
```

興味深いことに、Devise のテスト用ヘルパーが見つからないためにテストが失敗しました（訳注: 失敗メッセージの中に Devise::Test::ControllerHelpers を使っていることを確認してください、という内容が書いてあります）。この結果からわかることは、コントローラの skip_before_action の行はちゃんと仕事をしていたということです！このあと、コントローラをテストする際にコントローラが認証 済み になるようにテストを修正します。ですが、いったんはコメントアウトした行を元に戻し、Home コントローラの機能を元に戻してください。

認証が必要なコントローラスペック

今度は Project コントローラ用に新しいコントローラスペックを作りましょう。再度ジェネレータを使ってください（bin/rails g rspec:controller projects --controller-specs --no-request-specs ）。それから先ほどの home_controller_spec.rb と同じスペックを追加します。

spec/controllers/projects_controller_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe ProjectsController, type: :controller do
4    describe "#index" do
5      # 正常にレスポンスを返すこと
6      it "responds successfully" do
7        get :index
8        expect(response).to be_successful
9      end
10
11     # 200レスポンスを返すこと
12     it "returns a 200 response" do
13       get :index
14       expect(response).to have_http_status "200"
15     end
16   end
17 end
```

スペックを実行すると、先ほども出てきた Devise のヘルパーが見つからないというメッセージが表示されます。それでは今からこの問題に対処していきましょう。Devise は認証が必要なコントローラのアクションに対して、ユーザーのログイン状態をシミュレートするヘルパーを提供しています。ですが、そのヘルパーはまだ追加されていません。失敗メッセージにはこの問題に対処する方法が少し詳しく載っています。というわけで、テストスイートにこのヘルパーモジュールを組み込みましょう。*spec/rails_helper.rb* を開き、次のような設定を追加してください。

spec/rails_helper.rb

```
RSpec.configure do |config|
  # 設定ブロックの他の処理は省略 ...

  # コントローラスペックで Devise のテストヘルパーを使用する
  config.include Devise::Test::ControllerHelpers, type: :controller
end
```

この状態でテストを実行してもまだ失敗します。ですが、失敗メッセージには新しい情報が載っています。つまり、ちょっとは前に進んだということです。どちらのテストも基本的に同じ理由、すなわち、成功を表す `200` レスポンスではなく、リダイレクトを表す `302` レスポンスが返ってきているために失敗しているのです。失敗するのは `index` アクションがユーザーのログインを要求しているにもかかわらず、私たちはまだそれをテスト内でシミュレートしていないからです。

みなさんがもし Devise を使っていないのであれば、使用している認証ライブラリのドキュメントを読み、コントローラスペックでログイン状態をシミュレートするにはどういう方法が良いのかを確認してください。もし Rails が提供している `has_secure_password` メソッドなどを使って認証機能を自分で作っている場合は、次のようにして自分でヘルパーメソッドを定義してみてください。

```
# 自分で対処する ....
def sign_in(user)
  cookies[:auth_token] = user.auth_token
end
```

すでに Devise のヘルパーはテストスイートに組み込んであるので、ログイン状態をシミュレートすることができます。具体的にはテストユーザーを作成し、それからそのユーザーでログインするようにテストに伝えます。テストユーザーは両方のテストで有効になるよう `before`

ブロックで作成し、それからログイン状態をシミュレートするために sign_in ヘルパーを使います。

spec/controllers/projects_controller_spec.rb

```
require 'rails_helper'

RSpec.describe ProjectsController, type: :controller do
  describe "#index" do
    before do
      @user = FactoryBot.create(:user)
    end

    # 正常にレスポンスを返すこと
    it "responds successfully" do
      sign_in @user
      get :index
      expect(response).to be_successful
    end

    # 200レスポンスを返すこと
    it "returns a 200 response" do
      sign_in @user
      get :index
      expect(response).to have_http_status "200"
    end
  end
end
```

さあ、これでスペックはパスするはずです。なぜなら、index アクションには認証済みのユーザーでアクセスしていることになるからです。

ここでちょっとテストをパスさせるためにどうしたのかを考えてみましょう。テストがパスしたのは必要な変更を加えたあとです。最初はログインしていなかったので、テストは失敗していました。アプリケーションセキュリティの観点からすると、認証されていないユーザー（ゲスト と呼んでもいいでしょう）がアクセスしたら強制的にリダイレクトされることもテストすべきではないでしょうか。ここからテストを拡張して、このシナリオを追加することは可能です。こういうケースは describe と context のブロックを使うと、テストを整理しやすくなります。というわけで、次のように変更してみましょう。

spec/controllers/projects_controller_spec.rb

```
require 'rails_helper'

RSpec.describe ProjectsController, type: :controller do
  describe "#index" do
    # 認証済みのユーザーとして
    context "as an authenticated user" do
      before do
        @user = FactoryBot.create(:user)
      end

      # 正常にレスポンスを返すこと
      it "responds successfully" do
        sign_in @user
        get :index
        expect(response).to be_successful
      end

      # 200レスポンスを返すこと
      it "returns a 200 response" do
        sign_in @user
        get :index
        expect(response).to have_http_status "200"
      end
    end

    # ゲストとして
    context "as a guest" do
      # テストをここに書く
    end
  end
end
```

ここでは index アクションの describe ブロック内に、二つの context を追加しました。一つ目は認証済みのユーザーを扱う context です。テストユーザーを作成する before ブロックがこの context ブロックの内側で入れ子になっている点に注意してください。それから、スペックを実行して正しく変更できていることを確認してください。

続いて認証されていないユーザーの場合をテストしましょう。"as a guest" の context を変更し、次のようなテストを追加してください。

spec/controllers/projects_controller_spec.rb

```
1  # ゲストとして
2  context "as a guest" do
3    # 302レスポンスを返すこと
4    it "returns a 302 response" do
5      get :index
6      expect(response).to have_http_status "302"
7    end
8
9    # サインイン画面にリダイレクトすること
10   it "redirects to the sign-in page" do
11     get :index
12     expect(response).to redirect_to "/users/sign_in"
13   end
14 end
```

最初のスペックは難しくないはずです。have_http_status マッチャはすでに使っていますし、302 というレスポンスコードもちょっと前の失敗メッセージに出てきました。二つ目のスペックでは redirect_to という新しいマッチャを使っています。ここではコントローラが認証されていないリクエストの処理を中断し、Devise が提供しているログイン画面に移動させていることを検証しています。

同じテクニックはアプリケーションの認可機能（つまり、ログイン済みのユーザーが、やりたいことをできるかどうかの判断）にも適用できます。これはコントローラの3行目で処理されています。

```
before_action :project_owner?, except: %i[ index new create ]
```

このアプリケーションではユーザーがプロジェクトのオーナーであることを要求します。では、新しいテストを追加しましょう。今回は show アクションのテストです。一つの describe ブロックと二つの context ブロックを追加してください。

spec/controllers/projects_controller_spec.rb

```
require 'rails_helper'

RSpec.describe ProjectsController, type: :controller do

  # インデックスのテストが並ぶ ...

  describe "#show" do
    # 認可されたユーザーとして
    context "as an authorized user" do
      before do
        @user = FactoryBot.create(:user)
        @project = FactoryBot.create(:project, owner: @user)
      end

      # 正常にレスポンスを返すこと
      it "responds successfully" do
        sign_in @user
        get :show, params: { id: @project.id }
        expect(response).to be_successful
      end
    end

    # 認可されていないユーザーとして
    context "as an unauthorized user" do
      before do
        @user = FactoryBot.create(:user)
        other_user = FactoryBot.create(:user)
        @project = FactoryBot.create(:project, owner: other_user)
      end

      # ダッシュボードにリダイレクトすること
      it "redirects to the dashboard" do
        sign_in @user
        get :show, params: { id: @project.id }
        expect(response).to redirect_to root_path
      end
    end
```

```
38    end
39  end
```

今回はテストごとに @project を作成しました。最初の context ではログインしたユーザーがプロジェクトのオーナーになっています。二つ目の context では別のユーザーがオーナーになっています。このテストにはもう一つ新しい部分があります。それはプロジェクトの id をコントローラアクションの param 値として渡さなければいけない点です。

テストを実行してパスすることを確認してください。

ユーザー入力をテストする

ここまでは HTTP の GET リクエストしか使ってきませんでした。ですがもちろん、ユーザーは POST や PATCH、DESTROY といったリクエストでコントローラにアクセスしてくることもあります。そこで、それぞれについて example を追加していきましょう。最初は POST から始めます。ログイン済みのユーザーであれば新しいプロジェクトが作成でき、ゲストであればアクションへのアクセスを拒否されることを検証します。では、それぞれについて context を追加しましょう。

spec/controllers/projects_controller_spec.rb

```
describe "#create" do
  # 認証済みのユーザーとして
  context "as an authenticated user" do
    before do
      @user = FactoryBot.create(:user)
    end

    # プロジェクトを追加できること
    it "adds a project" do
      project_params = FactoryBot.attributes_for(:project)
      sign_in @user
      expect {
        post :create, params: { project: project_params }
      }.to change(@user.projects, :count).by(1)
    end
  end

  # ゲストとして
```

```
19    context "as a guest" do
20      # 302レスポンスを返すこと
21      it "returns a 302 response" do
22        project_params = FactoryBot.attributes_for(:project)
23        post :create, params: { project: project_params }
24        expect(response).to have_http_status "302"
25      end
26
27      # サインイン画面にリダイレクトすること
28      it "redirects to the sign-in page" do
29        project_params = FactoryBot.attributes_for(:project)
30        post :create, params: { project: project_params }
31        expect(response).to redirect_to "/users/sign_in"
32      end
33    end
34  end
```

"as a guest" の context は index アクションで書いたテストとよく似ています。ただし、ここでは POST 経由で params を渡しています。とはいえ、各テストで期待される結果は index アクションのときと同じです。

次に update アクションのテストを見てみましょう。ここでは次のようなテストシナリオが書いてあります。ユーザーは自分のプロジェクトは編集できますが、他人のプロジェクトは編集できません。ゲストはどのプロジェクトも編集できません。テストはちょっと複雑になってきていますが、これまでに説明した内容だけで構成されています。まず、認可されたユーザーのスペックから始めましょう。

spec/controllers/projects_controller_spec.rb

```
1   describe "#update" do
2     # 認可されたユーザーとして
3     context "as an authorized user" do
4       before do
5         @user = FactoryBot.create(:user)
6         @project = FactoryBot.create(:project, owner: @user)
7       end
8
9       # プロジェクトを更新できること
10      it "updates a project" do
11        project_params = FactoryBot.attributes_for(:project,
```

```
12          name: "New Project Name")
13        sign_in @user
14        patch :update, params: { id: @project.id, project: project_params }
15        expect(@project.reload.name).to eq "New Project Name"
16      end
17    end
18
19    # 認可されていないユーザーとゲストユーザーのテストはいったんスキップ ...
20  end
```

ここに出てくるテストは一つだけです。既存のプロジェクトの更新が成功したかどうかを検証しています。最初にユーザーを作成し、それからそのユーザーをプロジェクトにアサインしています。それからテストの内部でアクションに渡すプロジェクトの属性値を作成しています。この属性値はユーザーがプロジェクトの編集画面で入力する値を想定したものです。`FactoryBot.attributes_for(:project)` はプロジェクトファクトリからテスト用の属性値をハッシュとして作成します。ここではテストの結果をわかりやすくするために、ファクトリに定義された `name` のデフォルト値を独自の値で上書きしています。それからユーザーのログインをシミュレートし、元のプロジェクトの `id` と一緒に新しいプロジェクトの属性値を params として PATCH リクエストで送信しています。最後にテストで使った `@project` の新しい値を検証します。`reload` メソッドを使うのはデータベース上の値を読み込むためです。こうしないと、メモリに保存された値が再利用されてしまい、値の変更が反映されません。ここではアクションで渡した値がプロジェクトに設定されていることを検証します。

続いて認可されていないユーザーがプロジェクトを更新しようとしたときのテストを見てみましょう。

spec/controllers/projects_controller_spec.rb

```
1   describe "#update" do
2     # 認可されたユーザーのテストは省略 ...
3
4     # 認可されていないユーザーとして
5     context "as an unauthorized user" do
6       before do
7         @user = FactoryBot.create(:user)
8         other_user = FactoryBot.create(:user)
9         @project = FactoryBot.create(:project,
10          owner: other_user,
11          name: "Same Old Name")
```

```
12      end
13
14      # プロジェクトを更新できないこと
15      it "does not update the project" do
16        project_params = FactoryBot.attributes_for(:project,
17          name: "New Name")
18        sign_in @user
19        patch :update, params: { id: @project.id, project: project_params }
20        expect(@project.reload.name).to eq "Same Old Name"
21      end
22
23      # ダッシュボードへリダイレクトすること
24      it "redirects to the dashboard" do
25        project_params = FactoryBot.attributes_for(:project)
26        sign_in @user
27        patch :update, params: { id: @project.id, project: project_params }
28        expect(response).to redirect_to root_path
29      end
30    end
31
32    # ゲストユーザーのテストは省略 ...
33  end
```

こちらは先ほどの example よりもちょっと複雑です。今回は認可されていないユーザーが他のユーザーのプロジェクトにアクセスしようとしたときのテストと同じ方法でテストデータをセットアップしています。それから、テストの内部ではプロジェクトの名前が変わっていないことを最初に検証し、それから認可されていないユーザーがダッシュボード画面にリダイレクトされることを検証しています。

最後はゲストユーザーの context です。これはとてもシンプルです。

spec/controllers/projects_controller_spec.rb

```
describe "#update" do
  # 認可されている場合と認可されていない場合の context は省略 ...

  # ゲストとして
  context "as a guest" do
    before do
      @project = FactoryBot.create(:project)
    end

    # 302レスポンスを返すこと
    it "returns a 302 response" do
      project_params = FactoryBot.attributes_for(:project)
      patch :update, params: { id: @project.id, project: project_params }
      expect(response).to have_http_status "302"
    end

    # サインイン画面にリダイレクトすること
    it "redirects to the sign-in page" do
      project_params = FactoryBot.attributes_for(:project)
      patch :update, params: { id: @project.id, project: project_params }
      expect(response).to redirect_to "/users/sign_in"
    end
  end
end
```

この書き方はもうお馴染みでしょう。これはここまでに説明したやり方と同じです。すなわち、オブジェクトを作成し、それからコントローラを使った変更を試みます。試みは失敗し、かわりにユーザーはログイン画面にリダイレクトさせられます。

ユーザーの入力を受け付けるアクションがもう一つあります。次は destroy アクションを見てみましょう。これは update によく似ています。認可されたユーザーであれば自分のプロジェクトは削除できますが、他のユーザーのプロジェクトは削除できません。ゲストの場合は一切アクセスできません。

spec/controllers/projects_controller_spec.rb

```ruby
describe "#destroy" do
  # 認可されたユーザーとして
  context "as an authorized user" do
    before do
      @user = FactoryBot.create(:user)
      @project = FactoryBot.create(:project, owner: @user)
    end

    # プロジェクトを削除できること
    it "deletes a project" do
      sign_in @user
      expect {
        delete :destroy, params: { id: @project.id }
      }.to change(@user.projects, :count).by(-1)
    end
  end

  # 認可されていないユーザーとして
  context "as an unauthorized user" do
    before do
      @user = FactoryBot.create(:user)
      other_user = FactoryBot.create(:user)
      @project = FactoryBot.create(:project, owner: other_user)
    end

    # プロジェクトを削除できないこと
    it "does not delete the project" do
      sign_in @user
      expect {
        delete :destroy, params: { id: @project.id }
      }.to_not change(Project, :count)
    end

    # ダッシュボードにリダイレクトすること
    it "redirects to the dashboard" do
      sign_in @user
      delete :destroy, params: { id: @project.id }
```

```
38        expect(response).to redirect_to root_path
39      end
40    end
41
42    # ゲストとして
43    context "as a guest" do
44      before do
45        @project = FactoryBot.create(:project)
46      end
47
48      # 302レスポンスを返すこと
49      it "returns a 302 response" do
50        delete :destroy, params: { id: @project.id }
51        expect(response).to have_http_status "302"
52      end
53
54      # サインイン画面にリダイレクトすること
55      it "redirects to the sign-in page" do
56        delete :destroy, params: { id: @project.id }
57        expect(response).to redirect_to "/users/sign_in"
58      end
59
60      # プロジェクトを削除できないこと
61      it "does not delete the project" do
62        expect {
63          delete :destroy, params: { id: @project.id }
64        }.to_not change(Project, :count)
65      end
66    end
67  end
```

このテストに出てきた新しい点は、destroy メソッドには DELETE リクエストでアクセスしているところぐらいです。テストコードを順に読んでみてください。ここに出てくるのはこの章で何度も出てきてお馴染みのパターンばかりのはずです。

ですが、もしあなたがこのプロジェクト管理アプリケーションをブラウザ上でさわってみたことがあるなら、UI とコントローラに食い違いがあることに気づいたかもしれません。実は UI にはプロジェクトを削除するボタンがないのです！正直に白状すると、これはうっか

りミスです。とはいえ、これはある意味、コントローラレベルのテストにおける欠点を示している例かもしれません。もしユーザーがアプリケーションの機能の一部にアクセスできないとしたら、それはつまり何を意味しているのでしょうか？この欠点についてはまたのちほど説明します。

ユーザー入力のエラーをテストする

ここまでに追加した認可済みのユーザーに対するテストを思い出してください。ここまで私たちは正常系の入力しかテストしませんでした。ユーザーはプロジェクトを作成、または編集するために有効な属性値を送信したので、Rails は正常にレコードを作成、または更新できました。ですが、モデルスペックの場合と同じように、何か正しくないことがコントローラ内で起こったときも意図した通りの動きになるか検証するのは良い考えです。今回の場合だと、もしプロジェクトの作成、または編集時にバリデーションエラーが発生したら、何が起きるでしょうか？

こういうケースの一例として、認可済みのユーザーが create アクションにアクセスしたときのテストを少し変更してみましょう。まず、テストを二つの新しい context に分割することから始めます。一つは有効な属性値で、もう一つは無効な属性値です。既存のテストは最初の context に移動し、無効な属性については新しくテストを追加します。

spec/controllers/projects_controller_spec.rb

```
describe "#create" do
  # 認可済みのユーザーとして
  context "as an authenticated user" do
    before do
      @user = FactoryBot.create(:user)
    end

    # 有効な属性値の場合
    context "with valid attributes" do
      # プロジェクトを追加できること
      it "adds a project" do
        project_params = FactoryBot.attributes_for(:project)
        sign_in @user
        expect {
          post :create, params: { project: project_params }
        }.to change(@user.projects, :count).by(1)
```

```
      end
    end

    # 無効な属性値の場合
    context "with invalid attributes" do
      # プロジェクトを追加できないこと
      it "does not add a project" do
        project_params = FactoryBot.attributes_for(:project, :invalid)
        sign_in @user
        expect {
          post :create, params: { project: project_params }
        }.to_not change(@user.projects, :count)
      end
    end
  end

  # 他の context は省略 ...
end
```

新しいテストではプロジェクトファクトリの新しいトレイトも使っています。こちらも追加しておきましょう。

spec/factories/projects.rb

```
FactoryBot.define do
  factory :project do
    sequence(:name) { |n| "Test Project #{n}" }
    description "Sample project for testing purposes"
    due_on 1.week.from_now
    association :owner

    # 既存のトレイトが並ぶ ...

    # 無効になっている
    trait :invalid do
      name { nil }
    end
  end
end
```

これで create アクションを実行したときに、名前のないプロジェクトの属性値が送信されます。この場合、コントローラは新しいプロジェクトを保存しません。

HTML 以外の出力を扱う

コントローラの責務はできるだけ小さくすべきです。ただし、コントローラが担うべき責務の一つに、適切なフォーマットでデータを返す、という役割があります。ここまでにテストしたコントローラのアクションはすべて text/html フォーマットでデータを返していました。ですが、テストの中ではそのことを特に意識していませんでした。

簡単に説明するために、ここでは Task コントローラを見ていきます。Task コントローラは Rails の scaffold で作成され、デフォルトで定義された CRUD アクションにはほとんど変更を加えていません。ですので、HTML と JSON の両方のフォーマットでリクエストを受け付け、レスポンスを返すことができます。JSON に限定したテストを書くとどうなるか、今から見ていきましょう。

Task コントローラ用のスペックファイルはまだ作成していません。ですが、ジェネレータを使えば簡単に作成できます（bin/rails g rspec:controller tasks --controller-specs --no-request-specs ）。それから、ここまでに学んだ認証機能のテストと、データを送信するテストの知識を使えば、シンプルなテストを追加することができます。

今回のテストは JSON を扱うコントローラのテストを網羅的に説明するものではありません。ですが、コントローラのスペックファイルで何をどうすればいいか、という参考情報にはなると思います。まず、コントローラの *show* アクションを見てみましょう。

spec/controllers/tasks_controller_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe TasksController, type: :controller do
4    before do
5      @user = FactoryBot.create(:user)
6      @project = FactoryBot.create(:project, owner: @user)
7      @task = @project.tasks.create!(name: "Test task")
8    end
9
10   describe "#show" do
11     # JSON 形式でレスポンスを返すこと
12     it "responds with JSON formatted output" do
13       sign_in @user
```

```
14        get :show, format: :json,
15          params: { project_id: @project.id, id: @task.id }
16        expect(response.content_type).to include "application/json"
17      end
18    end
19
20    # 他のテストは省略 ...
21  end
```

セットアップはこの章ですでに説明した他のスペックとほとんど同じです。必要なデータはユーザーとプロジェクト（ユーザーがアサインされる）とタスク（プロジェクトがアサインされる）の3つです。それから、テストの中でユーザーをログインさせ、GET リクエストを送信してコントローラの show アクションを呼びだしています。このテストのちょっとだけ新しい点は、デフォルトの HTML 形式のかわりに `format: :json` というオプションで JSON 形式であることを指定しているところです。こうするとコントローラは言われたとおりにリクエストを処理します。つまり、*application/json* の Content-Type でレスポンスを返してくれるのです。ただし、厳密には *application/json; charset=utf-8* のように文字コード情報も一緒に付いてきます。そこで、このテストでは `include` マッチャを使ってレスポンスの中に *application/json* が含まれていればテストがパスするようにしました。

意図した通りにテストできていることを確認するため、*application/json* を *text/html* に変えてみましょう。案の定、テストは失敗するはずです。

次に、*create* アクションが JSON を処理できることを確認するテストをいくつか追加してみましょう。

spec/controllers/tasks_controller_spec.rb

```
1   require 'rails_helper'
2
3   RSpec.describe TasksController, type: :controller do
4     before do
5       @user = FactoryBot.create(:user)
6       @project = FactoryBot.create(:project, owner: @user)
7       @task = @project.tasks.create!(name: "Test task")
8     end
9
10    # show のテストは省略 ...
11
12    describe "#create" do
```

```
13      # JSON 形式でレスポンスを返すこと
14      it "responds with JSON formatted output" do
15        new_task = { name: "New test task" }
16        sign_in @user
17        post :create, format: :json,
18          params: { project_id: @project.id, task: new_task }
19        expect(response.content_type).to include "application/json"
20      end
21
22      # 新しいタスクをプロジェクトに追加すること
23      it "adds a new task to the project" do
24        new_task = { name: "New test task" }
25        sign_in @user
26        expect {
27          post :create, format: :json,
28            params: { project_id: @project.id, task: new_task }
29        }.to change(@project.tasks, :count).by(1)
30      end
31
32      # 認証を要求すること
33      it "requires authentication" do
34        new_task = { name: "New test task" }
35        # ここではあえてログインしない ...
36        expect {
37          post :create, format: :json,
38            params: { project_id: @project.id, task: new_task }
39        }.to_not change(@project.tasks, :count)
40        expect(response).to_not be_successful
41      end
42    end
43  end
```

セットアップは同じですが、今回は POST リクエストをコントローラの *create* アクションに送信しています。また、この章ですでに説明した方法で task のパラメータも一緒に送信しています。そして、ここでもやはり JSON 形式でリクエストを送信するようにオプションを指定する必要があります。それから、JSON 形式でも「リクエストを送信して本当にデータベースに保存されるか？」もしくは「ユーザーがログインしていない状態であればデータベ

ースへの保存が中断されるか?」というようなチェックができます。

まとめ

この章ではアプリケーションに数多くのテストを追加しました。テストしたコントローラはたった二つだけなんですけどね！コントローラのテストは簡単に追加していくことができます。ですが、すぐに大きくなって手に負えなくなることもよくあります。

今回は実際のアプリケーションでよくあるコントローラのテストシナリオをいくつか説明しました。ですが、私が開発しているアプリケーションでは、コントローラのテストはアクセス制御が正しく機能しているか確認するテストに限定するようにしています。この章で説明したテストでいうと、認可されていないユーザーとゲストに対するテストが該当します。認可されているユーザーに関しては、より上のレベルのテストで検証できます（この内容は次の章で説明します)。

また、コントローラのテストは Rails や RSpec から完全にはなくなっていないものの、最近では時代遅れのテストになってしまいました。Project コントローラの `destroy` アクションをテストしたときのことを思い出してください。あれはテストは作ったものの、結局 UI が用意されていなかった、というオチでした。この件はコントローラのテストに限界があることの一例です。

私からのアドバイスをまとめると、コントローラのテストは対象となる機能の単体テストとして最も有効活用できるときだけ使うのがよい、ということです。ただし、使いすぎないように注意してください。

Q&A

`successful` と `http status` は両方チェックしないといけませんか？必ずしも必須ではありません。どちらかひとつで十分な場合もありますが、それはあなたのコントローラが HTTP クライアントに返すレスポンスの複雑さによります。

演習問題

- 今回は Project コントローラの `new` や `edit` はテストしませんでした。このテストを追加してみてください。ヒント：このテストは `show` アクションによく似たものになります。
- あなたのアプリケーションのコントローラについて、どのメソッドがどのユーザーに対

してアクセスを許可するか、表にまとめてください。たとえば、私が有料コンテンツを扱うブログアプリケーションを作っていたとします。ブログ記事にアクセスするためには、ユーザーは会員にならなければいけません。ですが、その記事を読みたくなるようにタイトルの一覧だけは見せて良いことにします。実際のユーザーは自分に割り当てられたロールによってアクセスレベルが異なります。このような架空の Post コントローラは次のような権限制御機能になるかもしれません。

役割	Index	Show	Create	Update	Destroy
管理者	あり	あり	あり	あり	あり
編集者	あり	あり	あり	あり	あり
筆者	あり	あり	あり	あり	なし
会員	あり	あり	なし	なし	なし
ゲスト	あり	なし	なし	なし	なし

この一覧表を使って必要なテストシナリオを検討してください。今回は *new* と *create* は一つのカラムにまとめました（なぜなら、何も作成できないのに *new* で画面を表示しても意味がないからです）。*edit* と *update* も同様です。ただし、*index* と *show* は別々にしています。この表をあなたが開発しているアプリケーションの認証/認可の要件と比較するとどうでしょうか？どこを変える必要がありますか？

6. システムスペックで UI をテストする

現時点で私たちはプロジェクト管理ソフトウェアのテストをかなりたくさん作ってきました。RSpec のインストールと設定を終えたあと、モデルとコントローラの単体テストを作りました。テストデータを生成するためにファクトリも使いました。さて今度はこれら全部を一緒に使って統合テストを作ります。言い換えるなら、モデルとコントローラが他のモデルやコントローラとうまく一緒に動作することを確認します。このようなテストを RSpec ではシステムスペック（system specs）と呼んでいます。システムスペックは受入テスト、または統合テストと呼ばれることもあります。この種のテストでは開発したソフトウェア全体が一つのシステムとして期待どおりに動くことを検証します。システムスペックのコツを一度つかめば、Rails アプリケーション内の様々な機能をテストできるようになります。またシステムスペックはユーザーから上がってきたバグレポートを再現させる際にも利用できます。

嬉しいことに、あなたは堅牢なシステムスペックを書くために必要な知識をほとんど全部身につけています。システムスペックの構造はモデルやコントローラとよく似ているからです。Factory Bot を使ってテストデータを生成することもできます。この章では *Capybara* を紹介します。*Capybara* は大変便利な Ruby ライブラリで、システムスペックのステップを定義したり、アプリケーションの実際の使われ方をシミュレートしたりするのに役立ちます。

本章ではシステムスペックの基礎を説明します。

- まず最初に、システムスペックをいつ、そしてなぜ書くのかを他の選択肢と比較しながら考えてみます。
- 次に、統合テストで必要になる追加ライブラリについて説明します。
- それからシステムスペックの基礎を見ていきます。
- そのあと、もう少し高度なアプローチ、すなわち JavaScript が必要になる場合のテストに取り組みます。
- 最後に、システムスペックのベストプラクティスを少し考えて本章を締めくくります。

この章で発生する変更点全体は GitHub 上のこの diff[31] 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-06-system origin/05-controllers
```

[31] https://github.com/JunichiIto/everydayrails-rspec-jp-2024/compare/05-controllers...06-system

なぜシステムスペックなのか？

私たちは大変 長い 時間をかけてコントローラのテストに取りくんできました。にもかかわらず、なぜ別のレイヤーをテストしようとするのでしょうか？それはなぜなら、コントローラのテストは比較的シンプルな 単体テスト であり、結局アプリケーションのごく一部をテストしているに過ぎないからです。システムスペックはより広い部分をカバーし、実際のユーザーがあなたのコードとどのようにやりとりするのかを表現します。言い換えるなら、システムスペックではたくさんの異なる部品が統合されて、一つのアプリケーションになっていることをテストします。

Rails 5.1以降の Rails では システムテスト（system test） という名前で、このタイプのテストがセットアップ時にデフォルトでサポートされています。内容的にはこの章で説明するシステムスペックとほとんど同じです。この章では Rails 標準の Minitest ではなく RSpec を使うため、のちほど設定を変更します。

システムスペックが登場する前はフィーチャスペック（feature specs）と呼ばれる統合テスト機能が使われていました。システムスペックとフィーチャスペックの違いやシステムスペックへの移行手順については、この章の後半で説明します。

システムスペックで使用する gem

前述のとおり、ここではブラウザの操作をシミュレートするために Capybara を使います。Capybara を使うとリンクをクリックしたり、Web フォームを入力したり、画面の表示を検証したりすることができます。Rails 5.1以降の Rails であれば Capybara はすでにインストールされています。なぜなら Capybara はシステムテストでも利用されるからです。念のため *Gemfile* を開き、テスト環境に Capybara が追加されていることを確認してください。

Gemfile

```
group :test do
  gem 'capybara'

  # その他の gem は省略 ...
end
```

これまでに見てきた gem とは異なり、Capybara には Rails の開発環境で実行可能なジェネレータは用意されていないため、Capybara が追加されているのはテスト環境 だけ です。これにより、開発環境のメモリ消費を少し軽くすることができます。

システムスペックの基本

Capybara を使うと高レベルなテストを書くことができます。Capybara では `click_link` や `fill_in` 、`visit` といった理解しやすいメソッドが提供されていて、アプリケーションで必要な機能の シナリオ を書くことができるのです。今から実際にやってみましょう。ここではジェネレータを使って新しいテストファイルを作成します。最初に `rails generate rspec:system projects` とコマンドラインに入力してください。作成されたファイルは次のようになっています。

```
require 'rails_helper'

RSpec.describe "Projects", type: :system do
  before do
    driven_by(:rack_test)
  end

  pending "add some scenarios (or delete) #{__FILE__}"
end
```

この新しいスペックファイルの `before` ブロックには `driven_by(:rack_test)` というコードが書かれています。このコードはあとで削除しますが、いったんこのままにしておきます。次にテストを書いてみましょう。このシステムスペックが何をしてどう動くか、あなたは予想できますか？

spec/system/projects_spec.rb

```ruby
require 'rails_helper'

RSpec.describe "Projects", type: :system do
  before do
    driven_by(:rack_test)
  end

  # ユーザーは新しいプロジェクトを作成する
  scenario "user creates a new project" do
    user = FactoryBot.create(:user)

    visit root_path
    click_link "Sign in"
    fill_in "Email", with: user.email
    fill_in "Password", with: user.password
    click_button "Log in"

    expect {
      click_link "New Project"
      fill_in "Name", with: "Test Project"
      fill_in "Description", with: "Trying out Capybara"
      click_button "Create Project"

      expect(page).to have_content "Project was successfully created"
      expect(page).to have_content "Test Project"
      expect(page).to have_content "Owner: #{user.name}"
    }.to change(user.projects, :count).by(1)
  end
end
```

このスペックのステップを順番に見ていくと、最初に新しいテストユーザーを作成し、次にログイン画面からそのユーザーでログインしています。それから アプリケーションの利用者が使うものとまったく同じ Web フォームを使って 新しいプロジェクトを作成しています。これはシステムスペックとコントローラスペックの重要な違いです。コントローラスペックではユーザーインターフェースを無視して、パラメータを直接コントローラのメソッドに送信します。この場合のメソッドは 複数の コントローラと 複数の アクションにな

ります。具体的には home#index 、sessions#new 、projects#index 、projects#new 、それに projects#create です。しかし、結果は同じになります。新しいプロジェクトが作成され、アプリケーションはそのプロジェクト画面へリダイレクトし、処理の成功を伝えるフラッシュメッセージが表示され、ユーザーはプロジェクトのオーナーとして表示されます。ひとつのスペックで全部できています！

ここで expect{} の部分に少し着目してください。この中ではブラウザ上でテストしたいステップを明示的に記述し、それから、結果の表示が期待どおりになっていることを検証しています。ここで使われているのは Capybara の DSL です。自然な英文になっているかというとそうでもありませんが、それでも理解はしやすいはずです。

expect{} ブロックの最後では change マッチャを使って最後の重要なテスト、つまり「ユーザーがオーナーになっているプロジェクトが本当に増えたかどうか」を検証しています。

click_button を使うと、起動されたアクションが完了する前に次の処理へ移ってしまうことがあります。そこで、click_button を実行した expect{} の内部で最低でも1個以上のエクスペクテーションを実行し、処理の完了を待つようにするのが良いでしょう。このサンプルコードでもそのようにしています。

scenario は it と同様に example の起点を表しています。scenario の代わりに、標準的な it 構文に置き換えることもできます。RSpec のドキュメントにあるシステムスペックの説明では it が使われています。以下は変更後のコード例です。

spec/system/projects_spec.rb

```
require 'rails_helper'

RSpec.describe "Projects", type: :system do
  # before ブロックの記述は省略 ...

  it "user creates a new project" do
    # example の中身 ...
```

個人的には、このまま（訳注: it "user creates ... のままにしておくこと）だと英文として不完全で読みづらいので、説明の文言を it "creates a new project as a user" のように変更することも検討すべきだと思います。また、もうひとつの代替案として、describe や context ブロックを使い、ブロック内のテストが as a user （ユーザーとして）であることを明示するのも良いかもしれません。このようにいくつかの選択肢がありますが、本書では scenario を使うことにします。

さて、最後のポイントを今から話します。システムスペックでは一つの example、もしくは一つのシナリオで複数のエクスペクテーションを書くのは全く問題ありません。一般的にシステムスペックの実行には時間がかかります。これまでに書いてきたモデルやコントローラの小さな example に比べると、セットアップや実行にずっと時間がかかります。また、テストの途中でエクスペクテーションを追加するのも問題ありません。たとえば、一つ前のスペックの中で、ログインの成功がフラッシュメッセージで通知されることを検証しても良いわけです。しかし本来、こういうエクスペクテーションを書くのであれば、ログイン機能の細かい動きを検証するために専用のシステムスペックを用意する方が望ましいでしょう。

Capybara の DSL

先ほど作ったテストでは読者のみなさんはすでにお馴染みであろう RSpec の構文（`expect`）と、ブラウザ上の操作をシミュレートする Capybara のメソッドを組み合わせて使いました。このテストではページを訪問し（`visit`）、ハイパーリンクにアクセスするためにリンクをクリックし（`click_link`）、フォームの入力項目に値を入力し（`fill_in` と `with`）、ボタンをクリックして入力値を処理しました（`click_button`）。

ですが、Capybara でできることはもっとたくさんあります。以下のサンプルコードは Capybara の DSL が提供しているその他のメソッドの使用例です。

```
# 全種類の HTML 要素を扱う
scenario "works with all kinds of HTML elements" do
  # ページを開く
  visit "/fake/page"
  # リンクまたはボタンのラベルをクリックする
  click_on "A link or button label"
  # チェックボックスのラベルをチェックする
  check "A checkbox label"
  # チェックボックスのラベルのチェックを外す
  uncheck "A checkbox label"
  # ラジオボタンのラベルを選択する
  choose "A radio button label"
  # セレクトメニューからオプションを選択する
  select "An option", from: "A select menu"
  # ファイルアップロードのラベルでファイルを添付する
  attach_file "A file upload label", "/some/file/in/my/test/suite.gif"

```

```
18    # 指定した CSS に一致する要素が存在することを検証する
19    expect(page).to have_css "h2#subheading"
20    # 指定したセレクタに一致する要素が存在することを検証する
21    expect(page).to have_selector "ul li"
22    # 現在のパスが指定されたパスであることを検証する
23    expect(page).to have_current_path "/projects/new"
24  end
```

セレクタの スコープ を制限することもできます。その場合は Capybara の within を使ってページの一部分に含まれる要素を操作します。

```
1  <div id="node">
2    <a href="http://nodejs.org">click here!</a>
3  </div>
4  <div id="rails">
5    <a href="http://rubyonrails.org">click here!</a>
6  </div>
```

上のような HTML では次のようにしてアクセスしたい *click here!* のリンクを選択できます。

```
1  within "#rails" do
2    click_link "click here!"
3  end
```

もしテスト内で指定したセレクタに合致する要素が複数見つかり、Capybara にあいまいだ（ambiguous）と怒られたら、within ブロックで要素を内包し、あいまいさをなくしてみてください。

また、Capybara にはさまざまな find メソッドもあります。これを使うと値を指定して特定の要素を取り出すこともできます。たとえば次のような感じです。

```
1  language = find_field("Programming language").value
2  expect(language).to eq "Ruby"
3
4  find("#fine_print").find("#disclaimer").click
5  find_button("Publish").click
```

ここで紹介した Capybara のメソッドは、私が普段よく使うメソッドです。ですが、テスト内で使用できる Capybara の全機能を紹介したわけではありません。全容を知りたい場合はCapybara DSL のドキュメント[32]を参照してください。また、このドキュメントを便利なリ

[32] https://github.com/teamcapybara/capybara#the-dsl

ファレンスとして手元に置いておくのもいいでしょう。これ以降の章でも、まだ紹介していない機能をもうちょっと使っていきます。

システムスペックをデバッグする

Capybara のコンソール出力を読めば、どこでテストが失敗したのか調査することができます。ですが、それだけでは原因の一部分しかわからないことがときどきあります。たとえば次のシステムスペックを見てください。この場合、ユーザーはログインしていないのでテストは失敗します。

```
# ゲストがプロジェクトを追加する
scenario "guest adds a project" do
  visit projects_path
  click_link "New Project"
end
```

ですが、出力結果を見ると本当の原因に関する手がかりが載っていません。出力結果からわかることは、ページに要求されたリンクがない（Unable to find link “New Project”）ということだけです。

```
Failures:

  1) Projects guest adds a project
     Failure/Error: click_link "New Project"

     Capybara::ElementNotFound:
       Unable to find link "New Project"
     # 残りのスタックトレースは省略 ...
```

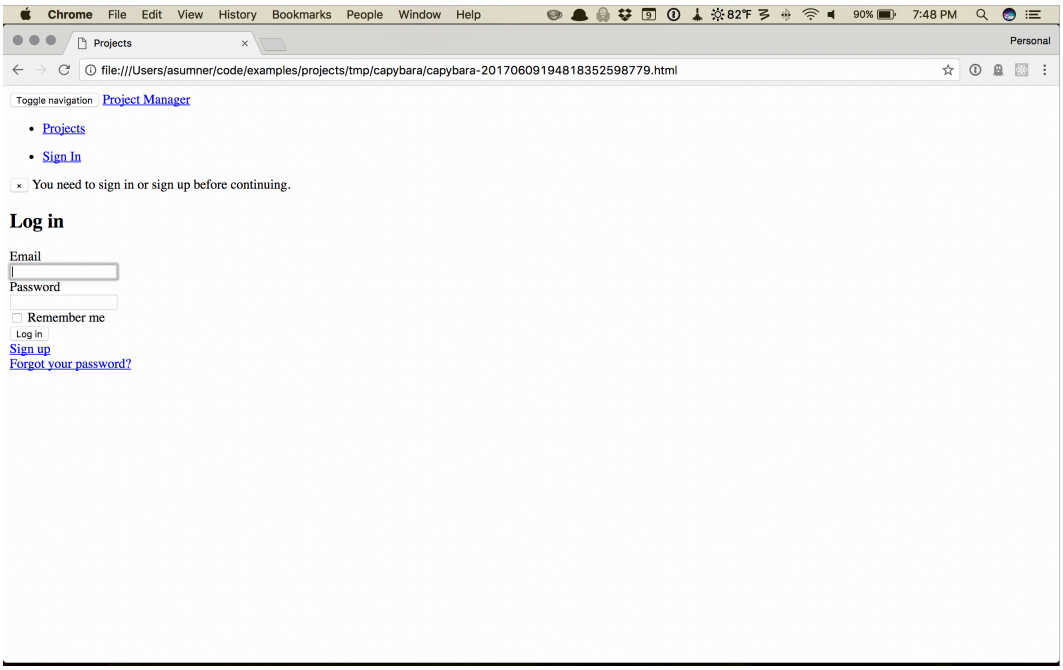
driven_by メソッドで :rack_test を指定した場合、Capybara は ヘッドレス ブラウザ（訳注: UI を持たないブラウザ）を使ってテストを実行するため、処理ステップを一つずつ目で確認することはできません。ですが、Rails がブラウザに返した HTML を見ることはできます。次のように save_and_open_page をテストが失敗する場所の直前に挟み込んでみてください。

```
scenario "guest adds a project" do
  visit projects_path
  save_and_open_page
  click_link "New Project"
end
```

この状態でテストを実行すると、同じ理由でテストは失敗するものの、新しい情報が手に入ります。

```
File saved to
  /Users/asumner/code/examples/projects/tmp/capybara/
  capybara-20170214213449303268565 2.html.
Please install the launchy gem to open the file automatically.
  guest adds a project (FAILED - 1)
```

コマンドライン、または使用しているマシンの GUI から保存されたファイルをブラウザ上で開いてください。



> 日本語版のサンプルアプリケーションでは後述する *Launchy* gem を予めインストールしてあるので、自動的にブラウザが立ち上がります。

なるほど！ボタンにアクセスできないのは、ユーザーがログインしていなかったからですね。プロジェクト一覧画面ではなく、ログイン画面にリダイレクトされていたわけです。

この機能はとても便利ですが、毎回手作業でファイルを開く必要はありません。コンソール出力にも書いてあるとおり、*Launchy* gem をインストールすれば自動的に開くようになります。*Gemfile* にこの gem を追加し、`bundle install` を実行してください。

Gemfile

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

  gem 'launchy'
end
```

> 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 *launchy* を追加してあるので、*Gemfile* を編集したり `bundle install` を実行したりする必要はありません。

こうすれば save_and_open_page をスペック内で呼びだしたときに、Launchy が保存された HTML を自動的に開いてくれます。

ブラウザを起動する必要がない場合や、ブラウザを起動できないコンテナ環境などでは代わりに save_page メソッドを使ってください。このメソッドを使うと HTML ファイルが *tmp/capybara* に保存されます。ブラウザは起動しません。

save_and_open_page や save_page はデバッグ用のメソッドです。システムスペックがパスするようになったら、それ以上のチェックは不要です。なので、不要になったタイミングでこのメソッド呼び出しは全部削除してください。削除しないままバージョン管理ツールにコミットしてしまわないよう注意しましょう。

JavaScript を使った操作をテストする

というわけで、私たちはシステムスペックを使ってプロジェクトを追加する UI が期待どおりに動作することを検証しました。ここで紹介した方法を使えば、Web 画面上の操作の大半をテストすることができます。ここまで Capybara はシンプルなブラウザシミュレータ（つまりドライバ）を使って、テストに書かれたタスクを実行してきました。このドライバは Rack::Test というドライバで、速くて信頼性が高いのですが、JavaScript の実行はサポートしていません。

本書のサンプルアプリケーションでは1箇所だけ JavaScript に依存する機能があります。それはタスクの隣にあるチェックボックスをクリックするとそのタスクが完了状態になる、と

いう機能です。新しいスペックを書いてこの機能をテストしてみましょう。システムスペックのジェネレータを使うか、もしくは自分の手で次のような *spec/system/tasks_spec.rb* という新しいファイルを追加してください。

spec/system/tasks_spec.rb

```
require 'rails_helper'

RSpec.describe "Tasks", type: :system do
  # ユーザーがタスクの状態を切り替える
  scenario "user toggles a task", js: true do
    user = FactoryBot.create(:user)
    project = FactoryBot.create(:project,
      name: "RSpec tutorial",
      owner: user)
    task = project.tasks.create!(name: "Finish RSpec tutorial")

    visit root_path
    click_link "Sign in"
    fill_in "Email", with: user.email
    fill_in "Password", with: user.password
    click_button "Log in"

    click_link "RSpec tutorial"
    check "Finish RSpec tutorial"

    expect(page).to have_css "label#task_#{task.id}.completed"
    expect(task.reload).to be_completed

    uncheck "Finish RSpec tutorial"

    expect(page).to_not have_css "label#task_#{task.id}.completed"
    expect(task.reload).to_not be_completed
  end
end
```

最初に説明した `projects_spec.rb` では `before` ブロックで `:rack_test` というドライバを指定していましたが、今回はこのあと別の方法でドライバを指定するため `driven_by` メソッドの記述はなくしています。加えて、ここでは `js: true` というオプション（タグ）を渡して

います。このようにして、指定したテストに対して JavaScript が使えるドライバを使うようにタグを付けておきます。このサンプルアプリケーションでは *selenium-webdriver* gem を使います。この gem は Rails 5.1以降の Rails にはデフォルトでインストールされていて、Capybara でもデフォルトの JavaScript ドライバになっています。

使用するドライバは `driven_by` メソッドを使ってテストごとに変更することができます。ですが、私は可能な限りシステム全体の共通設定とします。今からその共通設定を追加していきましょう。

rails_helper.rb ファイルはきれいな状態を保っておきたいので、今回は独立したファイルに新しい設定を書くことにします。RSpec はこのようなニーズをサポートしてくれているので、簡単な方法で有効化することができます。*spec/rails_helper.rb* 内にある以下の行のコメントを外してください。

spec/rails_helper.rb

```
Dir[Rails.root.join('spec', 'support', '**', '*.rb')].sort.each { |f| require f }
```

こうすると RSpec 関連の設定ファイルを *spec/support* ディレクトリに配置することができます。Devise 用の設定を追加したときのように、*spec/rails_helper.rb* 内に直接設定を書き込まなくても済むのです。それでは *spec/support/capybara.rb* という新しいファイルを作成し、次のような設定を追加しましょう。

spec/support/capybara.rb

```
RSpec.configure do |config|
  config.before(:each, type: :system) do
    driven_by :rack_test
  end

  config.before(:each, type: :system, js: true) do
    driven_by :selenium_chrome
  end
end
```

ここではブラウザを使った基本的なテストでは高速な `Rack::Test` ドライバを使い、より複雑なブラウザ操作が必要な場合は JavaScript が実行可能なドライバ（ここでは selenium-webdriver と Chrome）を設定するようにしています。どちらのドライバを使用するのかはタグで識別します。デフォルトでは `Rack::Test` ドライバを使いますが、`js: true` のタグが付いているテストに限り、selenium-webdriver と Chrome を使う設定になっています。

このほかに Chrome とやりとりするインターフェースになる *ChromeDriver* が必要になります。最新版の selenium-webdriver を使用すると自動的に適切なバージョンの ChromeDriver をダウンロードしてくれるため、特別な設定は不要です。

古い selenium-webdriver は ChromeDriver を自動的にダウンロードしてくれなかったため、Rails 6.0以降ではデフォルトで *Webdrivers* gem がインストールされていました。Chrome のバージョン114までは Webdrivers が適切に ChromeDriver をダウンロードしてくれていたのですが、Chrome 115以降ではテスト実行時に以下のようなエラーが発生します。

```
Webdrivers::VersionError:
  Unable to find latest point release version for 115.0.5790.
```

このエラーの解決方法について、Webdrivers gem はバージョン4.11以上の selenium-webdriver に移行することを公式に推奨しています。もしこのエラーに遭遇したら Gemfile から `gem 'webdrivers'` の行を削除してください。それから最新の selenium-webdriver と capybara をインストールするために以下のコマンドを実行してください。

```
bundle update selenium-webdriver capybara
```

もし、Gemfile 内に selenium-webdriver が見当たらなかったら、selenium-webdriver を追加し、`bundle update capybara` を実行してください。

Gemfile

```
group :test do
  # 他の gem は省略 ...

  gem 'selenium-webdriver'
end
```

このほかにもChromeDriver の公式ドキュメント[a]に記載されている手順に従ってインストールすることもできます。どちらでもお好きな方法を選択してください。

[a] https://sites.google.com/chromium.org/driver/

さあ、これで準備が整いました。実際にやってみましょう。新しく作ったスペックを実行してみてください。

```
$ bundle exec rspec spec/system/tasks_spec.rb
```

設定がうまくいっていれば、Chrome のウィンドウが新しく立ち上がります（ただし、現在開いている他のウィンドウのうしろに隠れているかもしれません）。ウィンドウ内ではサンプルアプリケーションが開かれ、目に見えない指がリンクをクリックし、フォームの入力項目を入力し、タスクの完了状態と未完了状態を切り替えます。素晴らしい！

テストはパスしましたが、このテストの遅さに注目してください！これは JavaScript を実行するテストと、Selenium を使うテストのデメリットです。一方で、セットアップは比較的簡単ですし、私たち自身が自分の手で操作する時間に比べたら、こちらの方がまだ速いです。ですが、もし一つのテストを実行するのに（私のマシンで）8秒以上かかるのであれば、この先 JavaScript を使う機能とそれに対応するテストを追加していったら、どれくらいの時間がかかるでしょうか？JavaScript ドライバはだんだん速くなっているので、そのうちいつか Rack::Test と同等のスピードで実行できるようになるかもしれません。ですが、それまでは必要なときにだけ、テスト上で JavaScript を有効にする方が良い、というのが私からのアドバイスです。

最後の仕上げとして、私たちのシステムスペックではデフォルトで `Rack::Test` ドライバを使うようになったため、`projects_spec.rb` の `before` ブロックは削除しても大丈夫です。`projects_spec.rb` から `before` ブロックを削除すると次のようになります。

spec/system/projects_spec.rb

```
require 'rails_helper'

RSpec.describe "Projects", type: :system do
  # ユーザーは新しいプロジェクトを作成する
  scenario "user creates a new project" do
    user = FactoryBot.create(:user)

    visit root_path
    click_link "Sign in"
    fill_in "Email", with: user.email
    fill_in "Password", with: user.password
    click_button "Log in"

    expect {
      click_link "New Project"
      fill_in "Name", with: "Test Project"
      fill_in "Description", with: "Trying out Capybara"
```

```
18        click_button "Create Project"
19
20        expect(page).to have_content "Project was successfully created"
21        expect(page).to have_content "Test Project"
22        expect(page).to have_content "Owner: #{user.name}"
23      }.to change(user.projects, :count).by(1)
24    end
25  end
```

修正が終わったら、このスペックを実行してみてください。

```
$ bundle exec rspec spec/system/projects_spec.rb
```

before ブロックを削除したあともこれまでと同様に Chrome が起動することなくテストが完了すれば OK です。

ヘッドレスドライバを使う

テストの実行中にブラウザのウィンドウが開くのはあまり望ましくないケースがよくあります。たとえば、GitHub Actions や Travis CI、Jenkins のような 継続的インテグレーション（CI）環境で実行する場合、先ほど作ったテストは CLI（コマンドラインインターフェース）上で実行する必要があります。ですが、CLI 上では新しいウィンドウを開くことはできません。こういった要件に対応するため、Capybara は ヘッドレス ドライバを使えるようになっています。そこで Chrome のヘッドレスモードを使ってテストを実行するよう、*spec/support/capybara.rb* を編集して次のようにドライバを変更してください。

spec/support/capybara.rb

```
config.before(:each, type: :system, js: true) do
  driven_by :selenium_chrome_headless
end
```

さあこれでブラウザのウィンドウを開くことなく、JavaScript を使うテストを実行できるようになりました。実際にスペックを実行してみてください。

```
$ bundle exec rspec spec/system/tasks_spec.rb
```

設定がうまくいっていれば、Chrome のウィンドウが表示されることなくテストが完了するはずです。

JavaScript の完了を待つ

デフォルトでは Capybara はボタンが現れるまで2秒待ちます。2秒待っても表示されなければ諦めます。次のようにするとこの秒数を好きな長さに変更できます。

```
Capybara.default_max_wait_time = 15
```

上の設定では待ち時間を15秒に設定しています。

この設定は *spec/support/capybara.rb* ファイルに書いてテストスイート全体に適用することができます（ただし、みなさんのアプリケーションが本書のサンプルアプリケーションと同じやり方で Capybara を設定していることが前提になります。つまり、このファイルは *spec/rails_helper.rb* によって読み込まれる場所に配置される必要があります）。しかし、この変更はテストスイートの実行がさらに遅くなる原因になるかもしれないので注意してください。もしこの設定を変えたいと思ったら、必要に応じてその都度 `using_wait_time` を使うようにした方がまだ良いかもしれません。たとえば次のようなコードになります。

```
# 本当に遅い処理を実行する
scenario "runs a really slow process" do
  using_wait_time(15) do
    # テストを実行する
  end
end
```

いずれにしても基本的なルールとして、処理の完了を待つために Ruby の `sleep` メソッドを使うのは 避けてください 。

スクリーンショットを使ってデバッグする

システムスペックでは `take_screenshot` メソッドを使って、テスト内のあらゆる場所でシミュレート中のブラウザの画像を作成することができます。ただし、このメソッドは JavaScript ドライバ（本書でいうところの `selenium-webdriver` です）を使うテストでしか使用できない点に注意してください。画像ファイルはデフォルトで *tmp/capybara* に保存されます。また、テストが失敗したら、自動的にスクリーンショットが保存されます！この機能はヘッドレスブラウザで実行している統合テストをデバッグするのに大変便利です。

JavaScript ドライバではなく Rack::Test を使ってテストしている場合は、本章の「システムスペックをデバッグする」で説明したように save_page メソッドを使って HTML ファイルを *tmp/capybara* に保存したり、save_and_open_page メソッドを使ってファイルを自動的にブラウザで開いたりすることができます。

システムスペックとフィーチャスペック

システムスペックが導入されたのは RSpec Rails 3.7からです。そして、システムスペックはその背後で Rails 5.1から導入されたシステムテストを利用しています。それ以前はフィーチャスペック（feature specs）と呼ばれる RSpec Rails 独自の機能を使って統合テストを書いていました。フィーチャスペックは見た目も機能面もシステムスペックに非常によく似ています。では、どちらのスペックを使うのが良いのでしょうか？もし、みなさんがまだ統合テストを一度も書いたことがないのなら、フィーチャスペックではなく、システムスペックを使ってください。ですが、フィーチャスペックも廃止されたわけではありません。昔から保守されている Rails アプリケーションではフィーチャスペックを使い続けている可能性もあります。そこで、このセクションでは簡単にシステムスペックとフィーチャスペックの違いを説明しておきます。

たとえば、この章の最初に紹介したシステムスペックのコード例をフィーチャスペックを使って書いた場合は次のようなコードになります。

spec/features/projects_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.feature "Projects", type: :feature do
4    # ユーザーは新しいプロジェクトを作成する
5    scenario "user creates a new project" do
6      user = FactoryBot.create(:user)
7
8      visit root_path
9      click_link "Sign in"
10     fill_in "Email", with: user.email
11     fill_in "Password", with: user.password
12     click_button "Log in"
13
14     expect {
15       click_link "New Project"
```

```
16        fill_in "Name", with: "Test Project"
17        fill_in "Description", with: "Trying out Capybara"
18        click_button "Create Project"
19
20        expect(page).to have_content "Project was successfully created"
21        expect(page).to have_content "Test Project"
22        expect(page).to have_content "Owner: #{user.name}"
23      }.to change(user.projects, :count).by(1)
24    end
25  end
```

一見、システムスペックとほとんど違いがありませんが、次のような点が異なります。

- *spec/system* ではなく *spec/features* にファイルを保存する
- `describe` メソッドではなく `feature` メソッドを使う
- `type:` オプションに `:system` ではなく `:feature` を指定する

ですが、`scenario` メソッドの内部はシステムスペックと全く同じです。
このほかにもフィーチャスペックには以下のような違いがあります。

- `let` や `let!` のエイリアスとして `given` や `given!` が使える（`let` や `let!` については第8章で説明します）
- `before` のエイリアスとして `background` が使える
- スクリーンショットを撮る場合、`save_screenshot` は使えるが、`take_screenshot` は使えない
- テストが失敗してもスクリーンショットは自動的に保存されない（明示的に `save_screenshot` メソッドを呼び出す必要があります）

フィーチャスペックはまだ使えますが、レガシーな機能になりつつあるため、早めにシステムスペックに移行する方が良いと思います。フィーチャスペックをシステムスペックに移行する場合は次のような手順に従ってください。

1. Rails 5.1以上かつ、RSpec Rails 3.7以上になっていることを確認する
2. システムスペックで使用する Capybara、Selenium Webdriver といった gem はなるべく最新のものを使うようにアップデートする
3. `js: true` のタグが指定された場合にドライバが切り替わるように設定を変更する（この章で紹介した `spec/support/capybara.rb` を参照してください）

4. *spec/features* ディレクトリを *spec/system* にリネームする
5. 各スペックのタイプを `type: :feature` から `type: :system` に変更する
6. 各スペックで使われている `feature` を `describe` に変更する
7. 各スペックで使われている `background` を `before` に変更する
8. 各スペックで使われている `given` / `given!` を `let` / `let!` に変更する
9. 各スペックで使われている `scenario` を `it` に変更する（この変更は任意です）
10. *spec/rails_helper.rb* の `config.include` などで、`type: :feature` になっている設定があれば `type: :system` に変更する

移行作業が終わったらテストスイートを実行してみてください。移行後のシステムスペックはきっとパスするはずです。

まとめ

システムスペックの書き方は一つ前の章までに習得したスキルの上に成り立っています。また、システムスペックは習得するのも理解するのも比較的簡単です。なぜなら Web ブラウザを起動すれば操作をシミュレートするために必要なステップが簡単にわかりますし、その次に Capybara を使ってそのステップを再現すれば済むからです。これは別にズルをしているわけではありません！このアプローチはみなさんがテストの書き方を練習し、コードのカバレッジを増やすための完璧な方法です。何もしなければコードはテストされないまま放置されてしまいます。

さて、次の章では人間以外のユーザーとアプリケーションのやりとりをテストし、外部向け API のカバレッジを増やしていきます。

演習問題

- システムスペックをいくつか書いてパスさせてください！最初はシンプルなユーザーの操作から始め、テストを書くプロセスに慣れてきたら、より複雑な操作へと進みましょう。
- システムの example に必要なステップを書くときは、ユーザーのことを考えてください。ユーザーは何らかの必要があってそのステップをブラウザ上で操作する人々です。もっとシンプルにできそうな、もしくは削除しても大丈夫そうなステップはありませんか？そうすることでユーザー体験全般をもっと快適にすることはできませんか？

7. リクエストスペックで API をテストする

最近では Rails アプリケーションが外部向け API を持つことも増えてきました。たとえば、Rails アプリケーションは JavaScript で作られたフロントエンドやネイティブモバイルアプリケーション、サードパーティ製アドオンのバックエンドとして API を提供することがあります。こうした API はこれまでにテストしてきたサーバーサイド出力による UI に追加される形で提供されることもありますし、UI のかわりに提供されることもあります。そして、みなさんのような開発者が API を利用し、顧客は API が高い信頼性を持っていることを望みます。ですのでやはり、みなさんは API もテストしたくなるはずです！

堅牢で開発者に優しい API の作り方は本書の範疇を超えてしまいますが、API のテストについてはそうではありません。嬉しいことに、もしみなさんがここまでにコントローラスペックやシステムスペックの章を読んできたのであれば、API をテストするために必要な基礎知識はすでに習得しています。この章で説明する内容は以下のとおりです。

- リクエストスペックとシステムスペックの違い
- 様々な種類の RESTful な API リクエストをテストする方法
- コントローラスペックをリクエストスペックで置き換える方法

この章で発生する変更点全体は GitHub 上のこの diff[33] 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-07-requests origin/06-system
```

リクエストスペックとシステムスペックの比較

最初に、こうしたテストはどのように使い分けるべきなのでしょうか？第5章 で説明したとおり、JSON（または XML）の出力はコントローラスペックで直接テストすることができます。みなさん自身のアプリケーションでしか使われない専用のシンプルなメソッドであれば、この方法で十分かもしれません。一方、より堅牢な API を構築するのであれば、第6章 で説明したシステムスペックによく似た統合テストが必要になってきます。ですが、違うと

[33] https://github.com/JunichiIto/everydayrails-rspec-jp-2024/compare/06-system...07-requests

ころもいくつかあります。RSpec の場合、今回の新しい API 関連のテストは *spec/requests* ディレクトリに配置するのがベストです。これまでに書いたシステムスペックとは区別しましょう。リクエストスペックでは Capybara も使いません。Capybara はブラウザの操作をシミュレートするだけであり、プログラム上のやりとりは特にシミュレートしないからです。かわりに、コントローラのレスポンスをテストする際に使った HTTP 動詞に対応するメソッド（`get`、`post`、`delete`、`patch`）を使います。

本書のサンプルアプリケーションにはユーザーのプロジェクト一覧にアクセスしたり、新しいプロジェクトを作成したりするための簡単な API が含まれています。どちらのエンドポイントもトークンによる認証を使います。サンプルコードは *app/controllers/api/projects_controller.rb* で確認できます。あまり難しいことはやっていませんが、先ほども述べたとおり、本書はテストの本であって、堅牢な API を設計するための本ではありません。

GET リクエストをテストする

最初の例では、最初に紹介したエンドポイントにフォーカスします。このエンドポイントは認証完了後、クライアントにユーザーのプロジェクト一覧を含む JSON データを返します。RSpec にはリクエストスペック用のジェネレータがあるので、これを使ってどういったコードが作成されるのか見てみましょう。コマンドラインから次のコマンドを実行してください。

```
$ bin/rails g rspec:request projects_api
```

新しく作られた *spec/requests/projects_apis_spec.rb* を開き、中を見てください。

spec/requests/projects_apis_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe "ProjectsApis", type: :request do
4    describe "GET /projects_apis" do
5      it "works! (now write some real specs)" do
6        get projects_apis_path
7        expect(response).to have_http_status(200)
8      end
9    end
10 end
```

一見すると、コントローラスペックにそっくりですね。しかし、すぐに思いますが、リクエストスペックではコントローラスペック以上にできることがたくさんあります。

RSpec はファイル名を複数形にしていますが、”API” が複数形になるのはちょっと不自然なように思います（つまり、*project APIs* にするか、*project API* にするか）。なので、私はいつも最初にファイル名をリネームします。というわけで、ファイル名を *spec/requests/projects_api_spec.rb* に変更し、新しいテストを追加してください。

spec/requests/projects_api_spec.rb

```
require 'rails_helper'

RSpec.describe 'Projects API', type: :request do
  # 1件のプロジェクトを読み出すこと
  it 'loads a project' do
    user = FactoryBot.create(:user)
    FactoryBot.create(:project,
      name: "Sample Project")
    FactoryBot.create(:project,
      name: "Second Sample Project",
      owner: user)

    get api_projects_path, params: {
      user_email: user.email,
      user_token: user.authentication_token
    }

    expect(response).to have_http_status(:success)
    json = JSON.parse(response.body)
    expect(json.length).to eq 1
    project_id = json[0]["id"]

    get api_project_path(project_id), params: {
      user_email: user.email,
      user_token: user.authentication_token
    }

    expect(response).to have_http_status(:success)
    json = JSON.parse(response.body)
    expect(json["name"]).to eq "Second Sample Project"
    # などなど
  end
```

```
33 end
```

上のサンプルコードはコントローラスペックっぽさが薄れ、システムスペックっぽいパターンになっています。この新しいスペックが リクエスト スペックです。最初はサンプルデータを作成しています。ここでは1人のユーザーと2件のプロジェクトを作成しています。一方のプロジェクトは先ほどのユーザーがオーナーで、もう一つのプロジェクトは別のユーザーがオーナーになっています。

次に、HTTP GET を使ったリクエストを実行しています。コントローラスペックと同様、ルーティング名に続いて パラメータ（params） を渡しています。この API ではユーザーのメールアドレスとサインインするためのトークンが必要になります。パラメータにはこの二つの値を含めています。ですが、コントローラスペックとは異なり、今回は好きなルーティング名を 何でも 使うことができます。リクエストスペックはコントローラに結びつくことはありません。これはコントローラスペックとは異なる点です。なので、テストしたいルーティング名をちゃんと指定しているか確認する必要も出てきます。

それから、テストは返ってきたデータを分解し、取得結果を検証します。データベースには2件のプロジェクトが格納されていますが、このユーザーがオーナーになっているのは1件だけです。そのプロジェクトの ID を取得し、2番目の API コールでそれを利用します。この API は1件のプロジェクトに対して、より多くの情報を返すエンドポイントです。この API はコールするたびに再認証が必要になる点に注意してください。ですので、メールアドレスとトークンは毎回パラメータとして渡す必要があります。

最後に、この API コールで返ってきた JSON データをチェックし、そのプロジェクト名とテストデータのプロジェクト名が一致するか検証しています。そしてここではちゃんと一致します。

POST リクエストをテストする

次のサンプルコードでは API にデータを送信しています。

spec/requests/projects_api_spec.rb

```ruby
require 'rails_helper'

RSpec.describe 'Projects API', type: :request do

  # 最初のサンプルコードは省略 ...

  # プロジェクトを作成できること
  it 'creates a project' do
    user = FactoryBot.create(:user)

    project_attributes = FactoryBot.attributes_for(:project)

    expect {
      post api_projects_path, params: {
        user_email: user.email,
        user_token: user.authentication_token,
        project: project_attributes
      }
    }.to change(user.projects, :count).by(1)

    expect(response).to have_http_status(:success)
  end
end
```

やはりここでもサンプルデータの作成から始まっています。今回は1人のユーザーと有効なプロジェクトの属性を集めたハッシュが必要です。それからアクションを実行して期待どおりの変化が発生するかどうか確認しています。この場合はユーザーが持つ全プロジェクトの件数が1件増えることを確認します。

今回のアクションはプロジェクト API に POST リクエストを送信することです。認証用のパラメータを送信する点は GET リクエストの場合と同じですが、今回はさらにプロジェクトの属性も含んでいます。それから最後にレスポンスのステータスをチェックしています。

コントローラスペックをリクエストスペックで置き換える

ここまで見てきたサンプルコードでは API をテストすることにフォーカスしていました。しかし API に限らず、第5章で作成したコントローラスペックをリクエストスペックで置き換

えることも可能です。既存の Home コントローラのスペックを思い出してください。このスペックは簡単にリクエストスペックに置き換えることができます。*spec/requests/home_spec.rb* にリクエストスペックを作成し、次のようなコードを書いてください。

spec/requests/home_spec.rb

```
require 'rails_helper'

RSpec.describe "Home page", type: :request do
  # 正常なレスポンスを返すこと
  it "responds successfully" do
    get root_path
    expect(response).to be_successful
    expect(response).to have_http_status "200"
  end
end
```

もう少し複雑な例も見てみましょう。たとえば Project コントローラの *create* アクションのテストは次のようなリクエストスペックに書き換えることができます。*spec/requests/projects_spec.rb* を作成してテストコードを書いてみましょう（前述の *projects_api_spec.rb* とはファイル名が異なる点に注意してください）。

spec/requests/projects_spec.rb

```
require 'rails_helper'

RSpec.describe "Projects", type: :request do
  # 認証済みのユーザーとして
  context "as an authenticated user" do
    before do
      @user = FactoryBot.create(:user)
    end

    # 有効な属性値の場合
    context "with valid attributes" do
      # プロジェクトを追加できること
      it "adds a project" do
        project_params = FactoryBot.attributes_for(:project)
        sign_in @user
        expect {
          post projects_path, params: { project: project_params }
```

```
18          }.to change(@user.projects, :count).by(1)
19        end
20      end
21
22      # 無効な属性値の場合
23      context "with invalid attributes" do
24        # プロジェクトを追加できないこと
25        it "does not add a project" do
26          project_params = FactoryBot.attributes_for(:project, :invalid)
27          sign_in @user
28          expect {
29            post projects_path, params: { project: project_params }
30          }.to_not change(@user.projects, :count)
31        end
32      end
33    end
34  end
```

コントローラスペックとの違いはごくわずかです。リクエストスペックでは Project コントローラの `create` アクションに直接依存するのではなく、具体的なルーティング名を指定して POST リクエストを送信します。それ以外はコントローラスペックとまったく同じコードです。

API 用のコントローラとは異なり、このコントローラでは標準的なメールアドレスとパスワードの認証システムを使っています。なので、この仕組みがちゃんと機能するように、ちょっとした追加の設定がここでも必要になります。今回は Devise の `sign_in` ヘルパーをリクエストスペックに追加します。Devise の wiki ページにあるサンプルコードを参考にして[34]この設定を有効にしてみましょう。

まず、*spec/support/request_spec_helper.rb* という新しいファイルを作成します。

[34] https://github.com/plataformatec/devise/wiki/How-To:-sign-in-and-out-a-user-in-Request-type-specs-(specs-tagged-with-type:-:request)

spec/support/request_spec_helper.rb

```
module RequestSpecHelper
  include Warden::Test::Helpers

  def self.included(base)
    base.before(:each) { Warden.test_mode! }
    base.after(:each) { Warden.test_reset! }
  end

  def sign_in(resource)
    login_as(resource, scope: warden_scope(resource))
  end

  def sign_out(resource)
    logout(warden_scope(resource))
  end

  private

  def warden_scope(resource)
    resource.class.name.underscore.to_sym
  end
end
```

それから *spec/rails_helper.rb* を開き、先ほど作成したヘルパーメソッドをリクエストスペックで使えるようにします。

spec/rails_helper.rb

```
# 最初のセットアップコードは省略 ...

RSpec.configure do |config|
  # 他の設定は省略 ...

  # Devise のヘルパーメソッドをテスト内で使用する
  config.include Devise::Test::ControllerHelpers, type: :controller
  config.include RequestSpecHelper, type: :request
end
```

bundle exec rspec spec/requests コマンドを実行して新しく追加したテストを動かしてみてく

ださい。いつものようにエクスペクテーションをいろいろ変えて遊んでみましょう。テストを失敗させ、それからまた元に戻してみましょう。

さあこれでみなさんはコントローラスペックとリクエストスペックの両方を書けるようになりました。では、どちらのテストを書くべきでしょうか？第5章でもお話ししたとおり、私はコントローラスペックよりも統合スペック（システムスペックとリクエストスペック）を強くお勧めします。なぜなら Rails におけるコントローラスペックは重要性が低下し、かわりにより高いレベルのテストの重要性が上がってきているためです。こうしたテストの方がアプリケーションのより広い範囲をテストすることができます。

とはいえ、コントローラレベルのテストを書く方法は人によってさまざまです。なので、とりあえずどちらのテストも書けるように練習しておいた方が良いと思います。実際、みなさんはこれでどちらのテストも書けるようになりました。

まとめ

Rails で作成した API をテストすることの重要性は徐々に上がってきています。なぜなら、最近ではアプリケーション同士がやりとりする機会が増えてきているからです。テストスイートを自分の API のクライアントだと考えるようにしてください。すでにお伝えしたとおり、この章は API の作り方を教える章ではありません。ですが、テストを利用して他のクライアントに公開しているインターフェースを改善すること も できます。

さて、これで典型的な Rails アプリケーションの全レイヤーをテストしました。しかし、これまで書いたテストコードではコードが重複している部分もあります。次の章ではこうしたコードの重複を無くしていきます。それだけでなく、テストを書くときに、あえて 重複を残したままにするケースについても見ていきます。

演習問題

- サンプルアプリケーションに別の API エンドポイントを追加してください。既存のプロジェクト API に追加してもいいですし、タスクやその他の機能にアクセスする API を追加しても構いません。それから、その API のテストを書いてください（可能ならテストから先に書き始めてみましょう !)。
- 自分のアプリケーションですでにコントローラスペックを書いている場合、そのテストをリクエストスペックに移行する方法を考えてみてください。リクエストスペックではどんな違いが出てくるでしょうか？

8. スペックを DRY に保つ

みなさんがここまでに学んだ知識を使って自分のアプリケーションにテストを書いていけば、しっかりしたテストスイートがきっとできあがるはずです。しかし、コードはたくさん重複しています。いわば、*Don't Repeat Yourself*[35] (DRY) 原則を破っている状態です。

アプリケーションコードと同様、テストスイートをきれいにすることも検討しましょう。この章では RSpec が提供しているツールを使い、複数のテストをまたがってコードを共有する方法を説明します。また、どのくらい DRY になると DRY すぎる のかについても説明します。

この章で説明する内容は以下のとおりです。

- ワークフローをサポートモジュールに切り出す
- テスト内でインスタンス変数を再利用するかわりに `let` を使う
- shared_context に共通のセットアップを移動する
- RSpec と rspec-rails で提供されているマッチャに加えて、カスタムマッチャを作成する
- エクスペクテーションを集約して、複数のスペックをひとつにする
- テストの何を抽象化し、何をそのまま残すか判断する

この章で発生する変更点全体は GitHub 上のこの diff[36] 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-08-dry-specs origin/07-requests
```

サポートモジュール

ここまでに作ったシステムスペックをあらためて見てみましょう。今のところ、たった二つのスペックしか書いていませんが、どちらのテストにもユーザーがアプリケーションにログインするステップが含まれています。

[35] http://wiki.c2.com/?DontRepeatYourself
[36] https://github.com/JunichiIto/everydayrails-rspec-jp-2024/compare/07-requests...08-dry-specs

```
visit root_path
click_link "Sign in"
fill_in "Email", with: user.email
fill_in "Password", with: user.password
click_button "Log in"
```

もしログイン処理が変わったらどうなるでしょうか？たとえば、ボタンのラベルが変わるような場合です。こんな単純な変更であっても、いちいち全部のテストコードを変更しなければいけません。この重複をなくすシンプルな方法は サポートモジュール を使うことです。

ではコードを新しいモジュールに切り出してみましょう。*spec/support* ディレクトリに *login_support.rb* という名前のファイルを追加し、次のようなコードを書いてください。

spec/support/login_support.rb

```
1  module LoginSupport
2    def sign_in_as(user)
3      visit root_path
4      click_link "Sign in"
5      fill_in "Email", with: user.email
6      fill_in "Password", with: user.password
7      click_button "Log in"
8    end
9  end
10
11 RSpec.configure do |config|
12   config.include LoginSupport
13 end
```

このモジュールにはメソッドがひとつ含まれます。コードの内容は元のテストで重複していたログインのステップです。

モジュールの定義のあとには、RSpec の設定が続きます。ここでは `RSpec.configure` を使って新しく作ったモジュールを include しています。これは必ずしも必要ではありません。テスト毎に明示的にサポートモジュールを include する方法もあります。たとえば次のような感じです。

spec/system/projects_spec.rb

```
require 'rails_helper'

RSpec.describe "Projects", type: :system do
  include LoginSupport

  # ユーザーは新しいプロジェクトを作成する
  scenario "user creates a new project" do
    # ...
  end
end
```

さあ、これでログインのステップが重複している二つのスペックをシンプルにすることができます。また、この先で同じステップが必要になるスペックでも、このヘルパーメソッドを使うことができます。たとえば、プロジェクトのシステムスペックは次のように書き換えられます。

spec/system/projects_spec.rb

```
require 'rails_helper'

RSpec.describe "Projects", type: :system do
  # ユーザーは新しいプロジェクトを作成する
  scenario "user creates a new project" do
    user = FactoryBot.create(:user)
    sign_in_as user

    expect {
      click_link "New Project"
      fill_in "Name", with: "Test Project"
      fill_in "Description", with: "Trying out Capybara"
      click_button "Create Project"
    }.to change(user.projects, :count).by(1)

    expect(page).to have_content "Project was successfully created"
    expect(page).to have_content "Test Project"
    expect(page).to have_content "Owner: #{user.name}"
  end
end
```

共通のワークフローをサポートモジュールに切り出す方法は、コードの重複を減らすお気に入りの方法の一つで、とくに、システムスペックでよく使います。モジュール内のメソッド名は、コードを読んだときに目的がぱっとわかるような名前にしてください。もしメソッドの処理を理解するために、いちいちファイルを切り替える必要があるのなら、それはかえってテストを不便にしてしまっています。

ここで適用したような変更は過去にもやっています。それが何だかわかりますか？Devise はログインのステップを完全に省略できるヘルパーメソッドを提供しています。これを使えば、特定のユーザーに対して即座にセッションを作成できます。これを使えば UI の操作をシミュレートするよりずっと速いですし、ユーザーがログイン済みになっていることがテストを実行する上での重要な要件になっている場合は大変便利です。別の見方をすれば、ここでテストしたいのはプロジェクトの機能であって、ユーザーの機能やログインの機能ではない、ということもできます。

これを有効化するために *rails_helper.rb* を開き、他の Devise の設定に続けて次のようなコードを追加してください（訳注: `Devise::Test::IntegrationHelpers` の行を追加します）。

spec/rails_helper.rb

```
RSpec.configure do |config|
  # 他の設定は省略 ...

  # Devise のヘルパーメソッドをテスト内で使用する
  config.include Devise::Test::ControllerHelpers, type: :controller
  config.include RequestSpecHelper, type: :request
  config.include Devise::Test::IntegrationHelpers, type: :system
end
```

さあ、これで今回独自に作った *sign_in_as* メソッドを呼び出す部分は、Devise の *sign_in* ヘルパーで置き換えることができます。

spec/system/projects_spec.rb

```ruby
require 'rails_helper'

RSpec.describe "Projects", type: :system do
  # ユーザーは新しいプロジェクトを作成する
  scenario "user creates a new project" do
    user = FactoryBot.create(:user)
    sign_in user

    # 残りのシナリオが続く ...
  end
end
```

ではシステムスペックを実行して、何が起きるか見てみましょう。

```
Projects
  user creates a new project (FAILED - 1)

Failures:

  1) Projects user creates a new project
     Failure/Error: click_link "New Project"

     Capybara::ElementNotFound:
       Unable to find link "New Project"
     # スタックトレースは省略 ...
```

いったい何が起こったかわかりますか？もしわからなければ、save_and_open_page メソッドを各スペックで失敗している click_link メソッドの直前で呼び出してください（訳注：真っ白の画面が立ち上がりますが、後述する理由により、これは想定通りの挙動です）。これはどうやら独自に作ったログインヘルパーと、ヘルパーに切り出す前の元のステップでは、ログイン後にユーザーのホームページに遷移する副作用があったようです。しかし、Devise のヘルパーメソッドではセッションを作成するだけなので、どこからワークフローを開始するのかテスト内で明示的に記述しなければなりません（訳注:sign_in user に続けて、visit root_path を追加します）。

spec/system/projects_spec.rb

```
require 'rails_helper'

RSpec.describe "Projects", type: :system do
  # ユーザーは新しいプロジェクトを作成する
  scenario "user creates a new project" do
    user = FactoryBot.create(:user)
    sign_in user

    visit root_path

    # 残りのシナリオが続く ...
  end
end
```

このあとも同じようなコードを書いていく点に注意してください。独自のサポートメソッドのような仕組みを利用する場合、テスト内で明示的に次のステップを記述するのは基本的によい考えです。そうすることで、ワークフローが文書化されます。繰り返しになりますが、今回のサンプルコードではユーザーがログイン済みになっていることは、あくまでセットアップ上の要件にすぎません。ログインのステップ自体はテスト上の重要な機能になっているわけではない、という点を押さえておきましょう。

今回使った Devise のヘルパーメソッドはこのあとのテストでも使っていきます。

`let` で遅延読み込みする

私たちはここまでに `before` ブロックを使ってテストを DRY にしてきました。`before` ブロックを使うと `describe` や `context` ブロックの内部で、各テストの実行前に共通のインスタンス変数をセットアップできます。この方法も悪くはないのですが、まだ解決できていない問題が二つあります。第一に、`before` の中に書いたコードは `describe` や `context` の内部に書いたテストを実行するたびに 毎回 実行されます。これはテストに予期しない影響を及ぼす恐れがあります。また、そうした問題が起きない場合でも、使う必要のないデータを作成してテストを遅くする原因になることもあります。第二に、要件が増えるにつれてテストの可読性を悪くします。

こうした問題に対処するため、RSpec は `let` というメソッドを提供しています。`let` は呼ばれたときに初めてデータを読み込む、遅延読み込み を実現するメソッドです。`let` は before

ブロックの 外部で 呼ばれるため、セットアップに必要なテストの構造を減らすこともできます。

`let` の使い方を説明するために、タスクモデルのモデルスペックを作成してみましょう。このスペックはまだ作成していませんでした。`rspec:model` ジェネレータを使うか、*spec/models* に自分でファイルを作るかして、スペックファイルを作成してください。（ジェネレータを使う場合は、タスク用の新しいファクトリも作られるはずです。）それから次のようなコードを書いてください。

spec/models/task_spec.rb

```
require 'rails_helper'

RSpec.describe Task, type: :model do
  let(:project) { FactoryBot.create(:project) }

  # プロジェクトと名前があれば有効な状態であること
  it "is valid with a project and name" do
    task = Task.new(
      project: project,
      name: "Test task",
    )
    expect(task).to be_valid
  end

  # プロジェクトがなければ無効な状態であること
  it "is invalid without a project" do
    task = Task.new(project: nil)
    task.valid?
    expect(task.errors[:project]).to include("must exist")
  end

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

今回は4行目にある let を使って必要となるプロジェクトを作成しています。しかし、プロジェクトが作成されるのはプロジェクトが必要になるテストだけです。最初のテストはプロジェクトを作成します。なぜなら9行目で project が呼ばれるからです。project は4行目の let で作られた値を呼び出します。let は新しいプロジェクトを作成します。テストの実行が終わると、12行目以降のテストではプロジェクトが取り除かれます。他の2つのテストではプロジェクトを使いません。実際、テストは「プロジェクトがなければ無効な状態であること」というテストなので、本当にプロジェクトが いらない のです。なので、この二つのテストではプロジェクトはまったく作成されません。

let を使う場合はちょっとした違いがあります。before ブロックでテストデータをセットアップする際は、インスタンス変数に格納していたことを覚えていますか？let を使ったデータに関してはこれが当てはまりません。なので、9行目を見てみると、@project ではなく、project でデータを呼び出しています。

let は必要に応じてデータを作成するので、注意しないとトラブルの原因になることもあります。たとえば、メモ（Note）のモデルスペックをファクトリと let を使ってリファクタリングしてみましょう。

spec/models/note_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe Note, type: :model do
4    let(:user) { FactoryBot.create(:user) }
5    let(:project) { FactoryBot.create(:project, owner: user) }
6
7    # 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
8    it "is valid with a user, project, and message" do
9      note = Note.new(
10       message: "This is a sample note.",
11       user: user,
12       project: project,
13     )
14     expect(note).to be_valid
15   end
16
17   # メッセージがなければ無効な状態であること
18   it "is invalid without a message" do
19     note = Note.new(message: nil)
20     note.valid?
```

```
21      expect(note.errors[:message]).to include("can't be blank")
22    end
23
24    # 文字列に一致するメッセージを検索する
25    describe "search message for a term" do
26      let(:note1) {
27        FactoryBot.create(:note,
28          project: project,
29          user: user,
30          message: "This is the first note.",
31        )
32      }
33
34      let(:note2) {
35        FactoryBot.create(:note,
36          project: project,
37          user: user,
38          message: "This is the second note.",
39        )
40      }
41
42      let(:note3) {
43        FactoryBot.create(:note,
44          project: project,
45          user: user,
46          message: "First, preheat the oven.",
47        )
48      }
49
50      # 一致するデータが見つかるとき
51      context "when a match is found" do
52        # 検索文字列に一致するメモを返すこと
53        it "returns notes that match the search term" do
54          expect(Note.search("first")).to include(note1, note3)
55        end
56      end
57
58      # 一致するデータが1件も見つからないとき
```

```
59      context "when no match is found" do
60        # 空のコレクションを返すこと
61        it "returns an empty collection" do
62          expect(Note.search("message")).to be_empty
63        end
64      end
65    end
66  end
```

コードがちょっときれいになりましたね。なぜなら before ブロックでインスタンス変数をセットアップする必要がなくなったからです。実行してみると一発でテストがパスします！しかし一つ問題があります。ためしに *returns an empty collection* のテストで次の一行（訳注: `expect(Note.count).to eq 3`）を追加してみましょう。

spec/models/note_spec.rb

```
# 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
  # 空のコレクションを返すこと
  it "returns an empty collection" do
    expect(Note.search("message")).to be_empty
    expect(Note.count).to eq 3
  end
end
```

続けてスペックを実行します。

```
Failures:

  1) Note search message for a term when no match is found returns
  an empty collection
     Failure/Error: expect(Note.count).to eq 3

       expected: 3
            got: 0

       (compared using ==)
     # ./spec/models/note_spec.rb:56:in `block (4 levels) in
     <top (required)>'
```

いったい何が起きてるんでしょうか？このテストでは note1 と note2 と note3 をどれも明示的に呼び出していません。なので、データが作られず、search メソッドは何もデータがないデータベースに対して検索をかけます。当然、検索しても何も見つかりません！

この問題は search メソッドを実行する前に let で作ったメモを強制的に読み込むようにハックすれば解決できます。

spec/models/note_spec.rb

```
# 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
  # 空のコレクションを返すこと
  it "returns an empty collection" do
    note1
    note2
    note3
    expect(Note.search("message")).to be_empty
    expect(Note.count).to eq 3
  end
end
```

ですが、私に言わせれば、これは まさに ハックです。私たちは読みやすいスペックを書こうと努力しています。新しく追加した行は読みやすくありません。そこで、このようなハックをするかわりに、let! を使うことにします。let とは異なり、let! は遅延読み込みされません。let! はブロックを即座に実行します。なので、内部のデータも即座に作成されます。それでは、let を let! に書き換えましょう。

spec/models/note_spec.rb

```
let!(:note1) {
  FactoryBot.create(:note,
    project: project,
    user: user,
    message: "This is the first note.",
  )
}

let!(:note2) {
  FactoryBot.create(:note,
    project: project,
    user: user,
    message: "This is the second note.",
```

```
  )
}

let!(:note3) {
  FactoryBot.create(:note,
    project: project,
    user: user,
    message: "First, preheat the oven.",
  )
}
```

これで先ほどの実験はパスしました。しかし let! にまったく問題がないわけでもありません。まず、この変更でテストデータが遅延読み込みされない元の状態に戻ってきてしまいました。この点は今回大した問題にはなっていません。テストデータを使う example は、どちらも正しく実行するために3件全部のメモが必要です。とはいえ、多少注意する必要はあります。なぜなら すべての テストが余計なデータを持つことになり、予期しない副作用を引き起こすかもしれないからです。

次に、コードを読む際は let と let! の見分けが付きにくく、うっかり読み間違えてしまう可能性があります。繰り返しになりますが、私たちは読みやすいテストスイートを作ろうと努力しています。もし、みなさんがこのわずかな違いを確認するためにコードを読み返すようであれば、before とインスタンス変数に戻すことも検討してください。別にテストで必要なデータを直接テスト内でセットアップしてしまっても、なんら問題はない[37]のです。

こうした選択肢をいろいろ試し、あなたとあなたのチームにとって最適な方法を見つけてください。

shared_context （context の共有）

let を使うと複数のテストで必要な共通のテストデータを簡単にセットアップすることができます。一方、shared_context を使うと複数のテスト ファイル で必要なセットアップを行うことができます。

タスクコントローラのスペックを見てください。ここで各テストの前に実行されている before ブロックが shared_context に抜き出す候補のひとつになります。ですがその前に、インスタンス変数のかわりに let を使うようにリファクタリングしておいた方が良さそうです。というわけで、スペックを次のように変更してください。

[37] https://robots.thoughtbot.com/my-issues-with-let

spec/controllers/tasks_controller_spec.rb

```ruby
require 'rails_helper'

RSpec.describe TasksController, type: :controller do
  let(:user) { FactoryBot.create(:user) }
  let(:project) { FactoryBot.create(:project, owner: user) }
  let(:task) { project.tasks.create!(name: "Test task") }

  describe "#show" do
    # JSON 形式でレスポンスを返すこと
    it "responds with JSON formatted output" do
      sign_in user
      get :show, format: :json,
        params: { project_id: project.id, id: task.id }
      expect(response.content_type).to include "application/json"
    end
  end

  describe "#create" do
    # JSON 形式でレスポンスを返すこと
    it "responds with JSON formatted output" do
      new_task = { name: "New test task" }
      sign_in user
      post :create, format: :json,
        params: { project_id: project.id, task: new_task }
      expect(response.content_type).to include "application/json"
    end

    # 新しいタスクをプロジェクトに追加すること
    it "adds a new task to the project" do
      new_task = { name: "New test task" }
      sign_in user
      expect {
        post :create, format: :json,
          params: { project_id: project.id, task: new_task }
      }.to change(project.tasks, :count).by(1)
    end

```

```
38      # 認証を要求すること
39      it "requires authentication" do
40        new_task = { name: "New test task" }
41        # ここではあえてログインしない ...
42        expect {
43          post :create, format: :json,
44            params: { project_id: project.id, task: new_task }
45        }.to_not change(project.tasks, :count)
46        expect(response).to_not be_successful
47      end
48    end
49  end
```

この最初のリファクタリングで重要な手順は次の通りです。まず before ブロックの中にあった3行をブロックの外に移動します。それからインスタンス変数を作成するかわりに let を使うように変更します。そして、ファイル内のインスタンス変数を順番に書き換えます。これはファイル内のインスタンス変数を「検索と置換」すれば OK ですね（たとえば、@project は project に置換します）。

スペックを実行してテストが引き続きパスすることを確認してください。次に、*spec/support/contexts/project_setup.rb* を新たに作成し、次のような context を書いてください。

spec/support/contexts/project_setup.rb

```
1  RSpec.shared_context "project setup" do
2    let(:user) { FactoryBot.create(:user) }
3    let(:project) { FactoryBot.create(:project, owner: user) }
4    let(:task) { project.tasks.create!(name: "Test task") }
5  end
```

最後にコントローラスペックに戻り、最初に出てくる3行の let を次のような1行に置き換えてください。

spec/controllers/tasks_controller_spec.rb

```
require 'rails_helper'

RSpec.describe TasksController, type: :controller do
  include_context "project setup"

  # show と create のテストが続く ...
```

もう一度スペックを実行してください。テストは引き続きパスするはずです。さあ、これでユーザーやプロジェクトやタスクにアクセスする必要があるスペックは、`include_context "project setup"` という新しい context を include するだけで済みます。

カスタムマッチャ

ここまでは RSpec と rspec-rails が提供しているマッチャで全部の要件を満たすことができました。そしてこの先もこのまま使い続けられそうな気がします。ですが、もし自分のアプリケーションで何度も同じテストコードを書いているような場合は、自分で独自のカスタムマッチャを作成した方が良いかもしれません。

今まで書いてきた既存のテストでは、カスタムマッチャの候補になりそうな部分が少なくとも一つあります。タスクコントローラのテストではコントローラのレスポンスが *application/json* の Content-Type で返ってきていることを何度も検証しています。RSpec の重要な信条のひとつは、人間にとっての読みやすさです。ですので、カスタムマッチャを使って読みやすさを改善してみましょう。

ではこれから新しいマッチャを作っていきます。まず、*spec/support/matchers/content_type.rb* というファイルを追加してください。最初はシンプルに始めて、それから徐々にマッチャの機能を充実させていきます。

spec/support/matchers/content_type.rb

```
RSpec::Matchers.define :have_content_type do |expected|
  match do |actual|
    content_types = {
      html: "text/html",
      json: "application/json",
    }
    actual.content_type.include? content_types[expected.to_sym]
  end
end
```

マッチャは必ず名前付きで定義され、その名前をスペック内で呼び出すときに使います。今回は have_content_type という名前にしました。マッチャには match メソッドが必要です。典型的なマッチャは二つの値を利用します。一つは 期待される値 （expected value 、マッチャをパスさせるのに必要な結果）で、もう一つは 実際の値 （actual value 、テストを実行するステップで渡される値）です。ここでは、短く省略した Content-Type（:html または :json ）で content_types ハッシュ内の Content-Type 値を取り出し、それが実際の Content-Type に含まれることを 期待 します。含まれていればテストはパスし、含まれていなければテストは失敗します。

ではこのカスタムマッチャを使ってみましょう。タスクコントローラのスペックを開き、新しいマッチャを使うように変更します。

spec/controllers/tasks_controller_spec.rb

```
require 'rails_helper'

RSpec.describe TasksController, type: :controller do
  include_context "project setup"

  describe "#show" do
    # JSON 形式でレスポンスを返すこと
    it "responds with JSON formatted output" do
      sign_in user
      get :show, format: :json,
        params: { project_id: project.id, id: task.id }
      expect(response).to have_content_type :json
    end
  end

```

```
16    describe "#create" do
17      # JSON 形式でレスポンスを返すこと
18      it "responds with JSON formatted output" do
19        new_task = { name: "New test task" }
20        sign_in user
21        post :create, format: :json,
22          params: { project_id: project.id, task: new_task }
23        expect(response).to have_content_type :json
24      end
25
26      # 残りのスペックが続く ...
27    end
28  end
```

スペックを実行するとテストはパスするはずです。ではあえてテストを失敗させてみましょう。スペックの一つで to を to_not に変えてみてください。予想どおり失敗すると思います。

```
Failures:

  1) TasksController#show responds with JSON formatted output
     Failure/Error: expect(response).to_not have_content_type :json
       expected #<ActionDispatch::TestResponse:0x007fc6d2951730
       @mon_owner=nil, @mon_count=0,
       @mon_mutex=#<Thread::Mu...:Headers:0x007fc6d291f4b0
       @req=#<ActionController::TestRequest:0x007fc6d2951960 ...>>,
       @variant=[]>> not to have content type :json
```

これはあまり読みやすいとは言えませんね。この点については後ほど対処します。とりあえず、次は別の方法でテストを失敗させてみましょう。to_not を to に戻し、別の Content-Type を与えてみます。たとえば、:html や :csv などです。こちらもやはり失敗します。そしてエラーメッセージもあまり読みやすくありません。

```
Failures:

  1) TasksController#show responds with JSON formatted output
     Failure/Error: expect(response).to have_content_type :csv
       expected #<ActionDispatch::TestResponse:0x007fc6d1d353c0
       @mon_owner=nil, @mon_count=0,
       @mon_mutex=#<Thread::Mu...:Headers:0x007fc6d1d0f170
       @req=#<ActionController::TestRequest:0x007fc6d1d356b8 ...>>,
       @variant=[]>> to have content type :csv
```

では今から読みやすさを改善していきましょう。まず最初に Content-Type のハッシュを match メソッドの外に切り出します。RSpec ではマッチャー内でヘルパーメソッドを定義し、コードをきれいにすることができます。

spec/support/matchers/content_type.rb

```
1  RSpec::Matchers.define :have_content_type do |expected|
2    match do |actual|
3      begin
4        actual.content_type.include? content_type(expected)
5      rescue ArgumentError
6        false
7      end
8    end
9
10   def content_type(type)
11     types = {
12       html: "text/html",
13       json: "application/json",
14     }
15     types[type.to_sym] || "unknown content type"
16   end
17 end
```

もう一度テストを実行し、先ほどと同じ方法でテストを失敗させてみてください。マッチャのコードはちょっと読みやすくなりましたが、出力はまだ読みやすくありません（訳注: 同じ出力結果になります）。この点も改善可能です。RSpec のカスタムマッチャの DSL では match メソッドに加えて、失敗メッセージ（failure message ）と、否定の失敗メッセージ（negated failure message ）を定義するメソッドが用意されています。つまり、to や to_not で

失敗したときの報告方法を定義できるのです。

spec/support/matchers/content_type.rb

```
RSpec::Matchers.define :have_content_type do |expected|
  match do |actual|
    begin
      actual.content_type.include? content_type(expected)
    rescue ArgumentError
      false
    end
  end

  failure_message do |actual|
    "Expected \"#{content_type(actual.content_type)} " +
    "(#{actual.content_type})\" to be Content Type " +
    "\"#{content_type(expected)}\" (#{expected})"
  end

  failure_message_when_negated do |actual|
    "Expected \"#{content_type(actual.content_type)} " +
    "(#{actual.content_type})\" to not be Content Type " +
    "\"#{content_type(expected)}\" (#{expected})"
  end

  def content_type(type)
    types = {
      html: "text/html",
      json: "application/json",
    }
    types[type.to_sym] || "unknown content type"
  end
end
```

テストは引き続きパスするはずです。ですが、わざと失敗させてみると、出力内容が改善されています。

```
Failures:

  1) TasksController#show responds with JSON formatted output
     Failure/Error: expect(response).to_not have_content_type :json
       Expected "unknown content type (application/json; charset=utf-8)"
       to not be Content Type "application/json" (json)
```

ちょっと良くなりましたね。結果として受け取ったレスポンス（*application/json* の Content-Type ）はマッチャに渡した Content-Type を含んでいます。ですが、（わざと失敗させているため）それは期待していないレスポンスです。スペックに戻って to_not を to に戻し、それから :json のかわりに :html を渡してください。スペックを実行してみましょう。

```
Failures:

  1) TasksController#show responds with JSON formatted output
     Failure/Error: expect(response).to have_content_type :html
       Expected "unknown content type (application/json; charset=utf-8)"
       to be Content Type "text/html" (html)
```

すばらしい！読みやすさが改善されました。最後にもうひとつだけ改善しておきましょう。have_content_type はちゃんと動作していますが、be_content_type でも動くようにすると良いかもしれません。マッチャはエイリアスを作ることができます。カスタムマッチャの最終バージョンはこのようになります。

spec/support/matchers/content_type.rb

```
1  RSpec::Matchers.define :have_content_type do |expected|
2    match do |actual|
3      begin
4        actual.content_type.include? content_type(expected)
5      rescue ArgumentError
6        false
7      end
8    end
9
10   failure_message do |actual|
11     "Expected \"#{content_type(actual.content_type)} " +
12     "(#{actual.content_type})\" to be Content Type " +
13     "\"#{content_type(expected)}\" (#{expected})"
14   end
```

```
15
16    failure_message_when_negated do |actual|
17      "Expected \"#{content_type(actual.content_type)} " +
18      "(#{actual.content_type})\" to not be Content Type " +
19      "\"#{content_type(expected)}\" (#{expected})"
20    end
21
22    def content_type(type)
23      types = {
24        html: "text/html",
25        json: "application/json",
26      }
27      types[type.to_sym] || "unknown content type"
28    end
29  end
30
31  RSpec::Matchers.alias_matcher :be_content_type , :have_content_type
```

以上でカスタムマッチャは完成です。これはいいアイデアでしょうか？たしかにテストは確実に読みやすくなりました。「レスポンスが Content-Type JSON になっていることを期待する（*Expect response to be content type JSON* ）」は、「レスポンスの Content-Type が application/json を含んでいる（*Expect response content type to include application/json* ）」よりも改善されています。ですが、新しいマッチャがあると、メンテナンスしなければならないコードが増えます。カスタムマッチャにはその価値があるでしょうか？その結論はあなたとあなたのチームで決める必要があります。しかし何にせよ、これでみなさんは必要になったときにカスタムマッチャを作ることができるようになりました。

カスタムマッチャ作りにハマっていく前に、*shoulda-matchers* gem も一度見ておいてください。この gem はテストをきれいにする便利なマッチャをたくさん提供してくれます。特に役立つのがモデルとコントローラのスペックです。みなさんが必要とするマッチャは、すでにこの gem で提供されているかもしれません。たとえば、第3章で書いたスペックのいくつかは、`it { is_expected.to validate_presence_of :name }` のように、短くシンプルに書くことができます。

aggregate_failures （失敗の集約）

この章よりも前の章で、私はモデルとコントローラのスペックは各 example につきエクスペクテーションを一つに制限した方が良いと書きました。一方、システムスペックとリクエストスペックでは、機能の統合がうまくできていることを確認するために、必要に応じてエクスペクテーションをたくさん書いても良い、と書きました。ですが、単体テストでもいったんコーディングが完了してしまえば、この制限が必ずしも必要ないことがあります。また、統合テストでも Launchy（第6章 参照）に頼ることなく、失敗したテストのコンテキストを収集すると役に立つことが多いです。

ここで問題となるのは、RSpec はテスト内で失敗するエクスペクテーションに遭遇すると、そこで即座に停止して失敗を報告することです。残りのステップは実行されません。しかし、RSpec 3.3では **aggregate_failures** （失敗の集約）という機能が導入され、他のエクスペクテーションも続けて実行することができます。これにより、そのエクスペクテーションが失敗した原因がさらによくわかるかもしれません。

まず、aggregate_failures によって、低レベルのテストがきれいになるケースを見てみましょう。第5章では Project コントローラを検証するためにこのようなテストを書きました。

spec/controllers/projects_controller_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe ProjectsController, type: :controller do
4    describe "#index" do
5      # 認証済みのユーザーとして
6      context "as an authenticated user" do
7        before do
8          @user = FactoryBot.create(:user)
9        end
10
11       # 正常にレスポンスを返すこと
12       it "responds successfully" do
13         sign_in @user
14         get :index
15         expect(response).to be_successful
16       end
17
18       # 200レスポンスを返すこと
```

```
19        it "returns a 200 response" do
20          sign_in @user
21          get :index
22          expect(response).to have_http_status "200"
23        end
24      end
25
26      # 残りのスペックは省略 ...
27    end
28  end
```

このふたつのテストでやっていることはほとんど同じです（第5章でもそのように説明しています）。なので、どちらか一方を選択して、二つのテストを一つに集約することができます。まずは説明のために、二つのテストを次のようにまとめてください。その際、sign_in のステップをコメントアウトすることをお忘れなく（これは一時的な変更です）。

spec/controllers/projects_controller_spec.rb

```
# 正常にレスポンスを返すこと
it "responds successfully" do
  # sign_in @user
  get :index
  expect(response).to be_successful
  expect(response).to have_http_status "200"
end
```

このスペックを実行すると予想通り最初のエクスペクテーションで失敗します。二番目のエクスペクテーションも失敗するはずですが、このままでは絶対に実行されません。そこで、この二つのエクスペクテーションを集約してみましょう。

spec/controllers/projects_controller_spec.rb

```
# 正常にレスポンスを返すこと
it "responds successfully" do
  # sign_in @user
  get :index
  aggregate_failures do
    expect(response).to be_successful
    expect(response).to have_http_status "200"
  end
end
```

それからもう一度スペックを実行します。

```
Failures:

  1) ProjectsController#index as an authenticated user responds
  successfully
     Got 2 failures from failure aggregation block.
     # ./spec/controllers/projects_controller_spec.rb:14:in `block (4
     levels) in <main>'

     1.1) Failure/Error: expect(response).to be_successful
            expected `#<ActionDispatch::TestResponse:0x0000000119cf28
            ...省略...>.successful?` to be truthy, got false
          # ./spec/controllers/projects_controller_spec.rb:15:in `block
          (5 levels) in <main>'

     1.2) Failure/Error: expect(response).to have_http_status "200"
            expected the response to have status code 200 but it was 302
          # ./spec/controllers/projects_controller_spec.rb:16:in `block
          (5 levels) in <main>'
```

すばらしい、これでどちらのエクスペクテーションも実行されました。そして、どうして レスポンスが成功として返ってこなかったのか、追加の情報を得ることもできました。sign_in の行のコメントを外し、テストをグリーンに戻してください。

私は統合テストで aggregate_failures をよく使います。aggregate_failures を使えば、同じコードを何度も実行して遅くなったり、複雑なセットアップを複数のテストで共有したりせずに、

テストが失敗した複数のポイントを把握することができます。たとえば第6章 で説明した、プロジェクトを作成するシナリオでエクスペクテーションの一部を集約してみましょう。

spec/system/projects_spec.rb

```
require 'rails_helper'

RSpec.describe "Projects", type: :system do
  # ユーザーは新しいプロジェクトを作成する
  scenario "user creates a new project" do
    user = FactoryBot.create(:user)
    # この章で独自に定義したログインヘルパーを使う場合
    # sign_in_as user
    # もしくは Devise が提供しているヘルパーを使う場合
    sign_in user
    visit root_path

    expect {
      click_link "New Project"
      fill_in "Name", with: "Test Project"
      fill_in "Description", with: "Trying out Capybara"
      click_button "Create Project"
    }.to change(user.projects, :count).by(1)

    aggregate_failures do
      expect(page).to have_content "Project was successfully created"
      expect(page).to have_content "Test Project"
      expect(page).to have_content "Owner: #{user.name}"
    end
  end
end
```

こうすると、何らかの原因でフラッシュメッセージが壊れても、残りの二つのエクスペクテーションは続けて実行されます。ただし、注意すべき点がひとつあります。それは、aggregate_failures は失敗する エクスペクテーション にだけ有効に働くのであって、テストを実行するために必要な一般的な実行条件には働かないと言うことです。上の例で言うと、もし何かがおかしくなって *New Project* のリンクが正しくレンダリングされなかった場合は、Capybara はエラーを報告します。

```
Failures:

  1) Projects user creates a new project
     Failure/Error: click_link "New Project"

     Capybara::ElementNotFound:
       Unable to find link "New Project"
```

言い換えるなら、ここでやっているのはまさに失敗するエクスペクテーションの集約であり、失敗全般を集約しているわけではありません。とはいえ、私は aggregate_failures を気に入っており、自分が書くテストではこの機能をよく使っています。

テストの可読性を改善する

統合テストはさまざまな構成要素を検証します。UI も（JavaScript との実行ですらも）、アプリケーションロジックも、データベース操作も、ときには外部システムとの連携も、全部ひとつのテストで検証できます。これはときに、だらだらしたテストコードになったり、テストコードが読みづらくなったり、何が起きているか理解するために手前や後ろのコードを行ったり来たりすることにつながります。

第6章で作成したテストを見直してください。このテストではタスクの完了を実行する UI を検証しました。その中でも特に、タスクの完了や未完了をマークするステップと、その処理が期待どおりに実行されたか検証するやり方を見直してみてください。

spec/system/tasks_spec.rb

```
1  # ユーザーがタスクの状態を切り替える
2  scenario "user toggles a task", js: true do
3    # セットアップとログインは省略 ...
4
5    check "Finish RSpec tutorial"
6    expect(page).to have_css "label#task_#{task.id}.completed"
7    expect(task.reload).to be_completed
8
9    uncheck "Finish RSpec tutorial"
10   expect(page).to_not have_css "label#task_#{task.id}.completed"
11   expect(task.reload).to_not be_completed
12 end
```

これぐらいであればそこまで可読性は悪くないかもしれません。ですが、この機能が今後もっと複雑になったらどうでしょうか？たとえば、タスクが完了したときにもっと詳細な情報を記録する必要が出てきた場合を考えてみてください。タスクを完了状態にしたユーザーの情報や、いつタスクが完了したのか、といった情報を記録する必要が出てきた場合などです。この場合、新しく追加された属性に対してそれぞれ、データベースの状態と UI の表示を確認するために新しい行を追加する必要が出てきます。また、チェックボックスのチェックが外されたときは逆の検証を同じようにする必要があります。こうなると、今はまだ小さいテストも、あっという間に長くなってしまいます。

このような場合は、スペックを読むときにコードの前後を行ったり来たりせずに済むよう、各ステップを独立したヘルパーメソッドに抽出することができます。新しくエクスペクテーションを追加する場合は、テストコード内に直接埋め込むのではなく、このヘルパーメソッドに追加します。これは「シングルレベルの抽象化を施したテスト（*testing at a single level of abstraction*[38]）」として知られるテクニックです。このテクニックの基本的な考えはテストコード全体を、内部で何が起きているのか抽象的に理解できる名前を持つメソッドに分割することです。あくまで抽象化が目的なので、内部の詳細を見せる必要はありません。

では、*spec/system/tasks_spec.rb* 内のだらだらしたコードを次のように整理してみましょう。

spec/system/tasks_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe "Tasks", type: :system do
4    let(:user) { FactoryBot.create(:user) }
5    let(:project) {
6      FactoryBot.create(:project,
7        name: "RSpec tutorial",
8        owner: user)
9    }
10   let!(:task) { project.tasks.create!(name: "Finish RSpec tutorial") }
11
12   # ユーザーがタスクの状態を切り替える
13   scenario "user toggles a task", js: true do
14     sign_in user
15     go_to_project "RSpec tutorial"
16
17     complete_task "Finish RSpec tutorial"
18     expect_complete_task "Finish RSpec tutorial"
```

[38] https://robots.thoughtbot.com/acceptance-tests-at-a-single-level-of-abstraction

```
19
20      undo_complete_task "Finish RSpec tutorial"
21      expect_incomplete_task "Finish RSpec tutorial"
22    end
23
24    def go_to_project(name)
25      visit root_path
26      click_link name
27    end
28
29    def complete_task(name)
30      check name
31    end
32
33    def undo_complete_task(name)
34      uncheck name
35    end
36
37    def expect_complete_task(name)
38      aggregate_failures do
39        expect(page).to have_css "label.completed", text: name
40        expect(task.reload).to be_completed
41      end
42    end
43
44    def expect_incomplete_task(name)
45      aggregate_failures do
46        expect(page).to_not have_css "label.completed", text: name
47        expect(task.reload).to_not be_completed
48      end
49    end
50  end
```

テストをもっと読みやすくするために、この新バージョンでは何カ所かリファクタリングをしています。まず、テストデータの作成はテスト内から上側の `let` と `let!` メソッドに移動させています。それからテストの各ステップを別々のメソッドに切り出しました。ユーザーとしてログインし、続いてテスト対象のプロジェクトに移動します。それから最後に目的

のステップを実行し、各ステップについて独自に作成したエクスペクテーションを実行します。具体的にはタスクを完了済みにし、本当に完了済みになっているか確認します。そして完了済みのタスクを元に戻し、今度は未完了になっているか確認します。テストはとてもシンプルに読めるようになりました。私たちが実際に どうやって 必要なアクションを実行しているのかという詳細は、別のメソッドに追い出されています。

どうでしょう、よくなりましたか？個人的には新しいテストはとても読みやすくなっている点が気に入っています。もしかするとプログラミングを知らない人がこのテストを読んでも、何をやっているのかある程度理解できるかもしれません。もしタスクを完了したり未完了にしたりする方法を変更する場合でも、変更を加えるのはヘルパーメソッドだけで済みます。これも新バージョンの良いところです。新しく作ったヘルパーメソッドは同じファイルの他のテストで再利用できる点もいいですね。たとえば、もしタスクを追加したり削除したりするシナリオを追加する場合でも、このヘルパーメソッドを再利用できます（この新しいテストは演習問題としてみなさんにやってもらいます⁉）。

ですが、このアプローチについてはちょっと不満に感じている点もあります。私はこの変更でテストデータを準備するために `let!` を使ったところがあまり好きではありません。これは必ずしも「シングルレベルの抽象化を施したテスト」の良い効果とは言えないでしょう。この点については、ファイルにさらにシナリオを追加していく際に、テストデータの作成方法を引き続き改善していけると思います。また、私は過去にこのアプローチが過剰に使われているケースを見てきました。過剰に使われてしまうと、ヘルパーメソッドが何をやって、何を検証しているのか理解するのにテストスイートの中身を詳細に確認しなければなりません。私に言わせれば、こうなってしまうと可読性を上げるための努力が、かえって可読性を下げることになっています。

私は「シングルレベルの抽象化を施したテスト」の考え方は好きです。ですが、みなさんは RSpec や Capybara に慣れるまで、必ずしも使う必要はないと考えています。ここで覚えておいてほしいことは、アプリケーション側のコードと同様、テストスイートのコードも自分のスキルが向上するにつれて改善されていく、ということです。

まとめ

この章ではテスト内、もしくは複数のテストファイルにまたがるコードの重複を減らすアプローチをいくつか見てきました。もうお気づきかもしれませんが、どのアプローチも効果的なテストスイートを構築するために 必須 というわけではありません。ですが、こうしたアプローチを理解し、賢く適用していけば、長い目で見たときに保守しやすいテストスイー

トになります。これは Rails のコードベースに Rails のベストプラクティスを適用しておけば、ずっと快適に開発できるのと同じです。

演習問題

- 今度は現状のプロジェクト用のシステムスペックを見てください。もしプロジェクトを編集する場合のテストを追加するとしたら、既存の「ユーザーは新しいプロジェクトを作成する（*user creates a new project* ）」シナリオからどのステップを再利用しますか？みなさんはこのシナリオ用のテストを追加できますか？ただし、その際はこの章で紹介した、テストを DRY にするテクニックも一緒に使ってください。もちろん、可読性と保守性は維持する必要があります。

9. 速くテストを書き、速いテストを書く

プログラミングに関する格言の中で私が気に入っているものの一つは、「まず動かし、次に正しくし、それから速くする（Make it work, make it right, make it fast[39]）」です。最初の方の章では、とりあえずテストが動くようにしました。それから第8章ではテストをリファクタリングして、コードの重複をなくすテクニックを説明しました。この章では先ほどの格言で言うところの 速く するパートを説明します。

私は 速い という用語を二つの意味で使っています。一つはもちろん、スペックの実行時間です。本書のサンプルアプリケーションとテストスイートは徐々に大きくなってきています。このまま何もせずに開発を進めれば、テストはうんざりするぐらい遅くなっていくことでしょう。よって目標としたいのは、RSpec の可読性や堅牢なテストスイートによってもたらされる安心感を損なうことなく、RSpec の実行速度を十分満足できる速さに保つことです。そして、私が意図する二つ目の スピード は、意味がわかりやすくてきれいなスペックを開発者としていかに素早く作るか、ということです。

本章ではこの両面を説明します。具体的な内容は以下の通りです。

- 構文を簡潔かつ、きれいにすることでスペックをより短くする RSpec のオプション
- みなさんのお気に入りのエディタを活用して、キー入力を減らす方法
- モックとスタブを使って、潜在的なパフォーマンス上のボトルネックをテストから切り離す方法
- 遅いスペックを除外するためのタグの使用
- テスト全体をスピードアップするテクニック

この章で発生する変更点全体は GitHub 上のこの diff[40] 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-09-test-faster origin/08-dry-specs
```

[39] http://wiki.c2.com/?MakeItWorkMakeItRightMakeItFast
[40] https://github.com/JunichiIto/everydayrails-rspec-jp-2024/compare/08-dry-specs...09-test-faster

RSpec の簡潔な構文

ここまでに書いてきたスペックのいくつかをもう一度見直してみてください。特に、モデルスペックに注目してみましょう。私たちはこれまでいくつかのベストプラクティスに従い、テストにはわかりやすいラベルを付け、一つの example に一つのエクスペクテーションを書いてきました。これらは全部目的があってやってきたことです。ここまで見てきたような明示的なアプローチは、私がテストの書き方を学習したときのアプローチを反映しています。こうしたアプローチを使えば、自分のやっていることが理解しやすいはずです。しかし、RSpec にはキーストロークを減らしながらこうしたベストプラクティスを実践し続けられるテクニックがあります。

一つ前の章では、テストデータを宣言するオプションとして `let` があることを説明しました。もう一つのメソッドである `subject` も同じように呼ばれますが、ユースケースがちょっと異なります。`subject` はテストの対象物（subject ）を宣言します。そして、そのあとに続く example 内で暗黙的に何度でも再利用することができます。

私たちは第3章からここまでずっと、`it` を長い記法で使い続けてきました。長い記法とはすなわち、文字列としてテストの説明を自然言語で記述するアプローチのことです。ですが、この `it` はただの Ruby のメソッドです。みなさんもすでに気づいているかもしれませんが、このメソッドはブロックを受け取り、そのブロックの中にはいくつかのテストのステップを含めることができます。これはつまり、シンプルなテストケースであればテストコードを1行にまとめることができるかもしれないということです！

これを実現するために、ここでは RSpec の `is_expected` メソッドを使用します。`is_expected` は `expect(something)` によく似ていますが、こちらはワンライナーのテスト（1行で書くテスト）を書くために使われます。

`specify` は `it` のエイリアスです。開発者の中には RSpec の簡潔な構文を使う際に `specify` を好む人もいます。スペックを書いたらそれを声に出して読んでみましょう。そして、一番意味がわかりやすいと思った方を使ってください（訳注：ここでいうわかりやすさとは英文としてのわかりやすさです）。この使い分けにはどういうときにどちらを使うべきかという厳密なルールは存在しません。

`is_expected` を使うと、次のようなテストが・・・

```
it "returns a user's full name as a string" do
  user = FactoryBot.build(:user)
  expect(user.name).to eq "Aaron Sumner"
end
```

以下のように少しだけ簡潔に書き換えることができます。

```
subject(:user) { FactoryBot.build(:user) }
it { is_expected.to satisfy { |user| user.name == "Aaron Sumner" } }
```

このようにしても結果は同じです。また、subject はあとに続くテストで暗黙的に再利用できるので、各 example で何度も打ち直す必要がありません。

とはいえ、私は自分のテストスイートでこの簡潔な構文を使うことはそれほど多くありません。また、subject が一つのテストでしか使われない場合も subject を使いません。ですが、**Shoulda Matchers** と一緒に使うのは 大好き です。Shoulda というのはそれ自体が単体のテスティングフレームワークで、RSpec と完全に置き換えて使うことができるものでした。Shoulda はすでに開発が止まっていますが、ActiveModel や ActiveRecord 、ActionController に使える豊富なマッチャのコレクションは独立した gem に切り出され、RSpec（もしくは Minitest）で利用することができます。この gem を一つ追加すれば、3行から5行あるスペックを1〜2行に減らすことができる場合があります。

Shoulda Matchers を使うには、まず gem を追加しなければなりません。*Gemfile* 内のテスト関係の gem のうしろに、この gem を追加してください。

Gemfile

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

  gem 'launchy'
  gem 'shoulda-matchers',
end
```

> 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 *shoulda-matchers* を追加してあるので、*Gemfile* を編集する必要はありません。

コマンドラインから bundle コマンドを実行したら、この新しい gem を使うようにテストスイートを設定する必要があります。*spec/rails_helper.rb* を開き、ファイルの一番最後に以下

のコードを追加してください。ここでは、RSpec と Rails で Shoulda Matchers を使うことを宣言しています。

spec/rails_helper.rb

```
Shoulda::Matchers.configure do |config|
  config.integrate do |with|
    with.test_framework :rspec
    with.library :rails
  end
end
```

さあこれでスペックを短くすることができます。User モデルのスペックにある、4つのバリデーションのテストから始めましょう。Shoulda Matchers を使えば、このテストがたった4行になります！

spec/models/user_spec.rb

```
it { is_expected.to validate_presence_of :first_name }
it { is_expected.to validate_presence_of :last_name }
it { is_expected.to validate_presence_of :email }
it { is_expected.to validate_uniqueness_of(:email).case_insensitive }
```

ここでは Shoulda Matchers が提供する二つのマッチャ（`validate_presence_of` と `validate_uniqueness_of`）を使ってテストを書いています。`email` のユニークバリデーションは Devise によって設定されているので、バリデーションは大文字と小文字を区別しないこと（not case sensitive）をスペックに伝える必要があります。`case_insensitive` が付いているのはそのためです。

次に Project モデルのスペックに注目してみましょう。具体的には以前書いた、ユーザーは同じ名前のプロジェクトを複数持つことができないが、ユーザーが異なれば同じ名前のプロジェクトがあっても構わない、というテストです。Shoulda Matchers を使えば、このテストを1個かつ1行のテストにまとめることができます。

spec/models/project_spec.rb

```
it { is_expected.to validate_uniqueness_of(:name).scoped_to(:user_id) }
```

私はモデル層でこのようなスタイルのテストを書くのが大好きです。特に、モデルをテストファーストで開発するときによく使います。たとえば、ウィジェット（widget）という新しいモデルが出てきたとします。そんなとき、私はスペックを開き、ウィジェットがどんな振る舞いを持つのか考えます。それから次のようなテストを書きます。

```
it { is_expected.to validate_presence_of :name }
it { is_expected.to have_many :dials }
it { is_expected.to belong_to :compartment }
it { is_expected.to validate_uniqueness_of :serial_number }
```

テストを書いたら、コードを書いてテストをパスさせます。このアプローチは、どんなコードを書くべきかを考え、それからアプリケーションコードが要件を満たしていることを確認するために非常に役立ちます。

エディタのショートカット

プログラミングを習得する上で非常に重要なことは、自分が使っているエディタを隅から隅まで理解することです。私はコーディングするときは（そして執筆するときも）たいていAtom[41]を使っています。さらに、スニペットを作る構文[42]も必要最小限は理解したので、テストを書くときによく使うコードもほとんどタイプせずに済ませることができます。たとえば、desc と入力してタブキーを押すと、私のエディタは describe...end のブロックを作成します。さらに、カーソルはブロック内に置かれ、適切な場所にコードを追加できるようになっています。

もしあなたも Atom を使っているのであれば、Everyday Rails RSpec Atom パッケージ[43]をインストールして私が使っているショートカットを試してみることができます。全スニペットの内容を知りたい場合はパッケージのドキュメントを参照してください。Atom 以外のエディタを使っている場合は、多少時間をかけてでも用意されているショートカットを使えるようになってください。また、最初から用意されているショートカットだけでなく、自分自身でショートカットを定義する方法も学習しておきましょう。お決まりのコードをタイプする時間が少なくなればなるほど、時間あたりのビジネスバリューが増えていきます！

> 日本語版の補足：Atom は2022年12月に開発が終了[a]しました。そのため、この節のリンクはすべて削除されています。なお、Atom の代わりとなるエディタとしては VS Code[b]等が挙げられます。
>
> [a] https://github.blog/2022-06-08-sunsetting-atom/
> [b] https://code.visualstudio.com/

[41] https://atom.io
[42] http://flight-manual.atom.io/using-atom/sections/snippets/
[43] https://atom.io/packages/atom-everydayrails-rspec

モックとスタブ

モックとスタブの使用、そしてその背後にある概念は、それだけで大量の章が必要になりそうなテーマです（一冊の本になってもおかしくありません）。インターネットで検索すると、正しい使い方や間違った使い方について人々が激しく議論する場面に出くわすはずです。また、多くの人々が二つの用語を定義しようとしているのもわかると思います。ただし、うまく定義できているかどうかは場合によりけりです。私が一番気に入っている定義は次の通りです。

- **モック（mock）**は本物のオブジェクトのふりをするオブジェクトで、テストのために使われます。モックはテストダブル（test doubles）と呼ばれる場合もあります。モックはこれまでファクトリやPOROを使って作成したオブジェクトの代役を務めます（訳注：英語の“double”には「代役」や「影武者」の意味があります）。しかし、モックはデータベースにアクセスしない点が異なります。よって、テストにかかる時間は短くなります。
- **スタブ（stub）**はオブジェクトのメソッドをオーバーライドし、事前に決められた値を返します。つまりスタブは、呼び出されるとテスト用に本物の結果を返す、ダミーメソッドです。スタブをよく使うのはメソッドのデフォルト機能をオーバーライドするケースです。特にデータベースやネットワークを使う処理が対象になります。

RSpecには多機能なモックライブラリが最初から用意されています。みなさんはもしかすると、Mochaのようなその他のモックライブラリをプロジェクトで使ったことがあるかもしれません。第4章以降でテストデータを作成するのに使ってきたFactory Botにもスタブオブジェクトを作るメソッドが用意されています。この項ではRSpec標準のモックライブラリに焦点を当てます。

例をいくつか見てみましょう。メモ（Note）モデルでは`delegate`を使ってメモに`user_name`という属性を追加しました。ここまでに学んだ知識を使うと、この属性がちゃんと機能していることをテストするために、次のようなコードを書くことができます。

spec/models/note_spec.rb

```
# 名前の取得をメモを作成したユーザーに委譲すること
it "delegates name to the user who created it" do
  user = FactoryBot.create(:user, first_name: "Fake", last_name: "User")
  note = Note.new(user: user)
  expect(note.user_name).to eq "Fake User"
end
```

このコードでは User オブジェクトを永続化する必要があります。これはテストで使う first_name と last_name というユーザーの属性にアクセスするためです。この処理に必要な時間はほんのわずかです。ですが、セットアップのシナリオが複雑になったりすると、わずかな時間もどんどん積み重なって無視できなくなるかもしれません。モックはこのように、データベースにアクセスする処理を減らすためによく使われます。

さらに、このテストは Note モデルのテストですが、User モデルの実装を知りすぎています。はたしてこのテストの中で User モデルの name が first_name と last_name から導出されていることを意識する必要があるのでしょうか？本来であれば関連を持つ User モデルが name という文字列を返すことを知っていればいいだけのはずです。

以下はこのテストの修正バージョンです。このコードではモックのユーザーオブジェクトと、テスト対象のメモに設定したスタブメソッドを使っています。

spec/models/note_spec.rb

```
# 名前の取得をメモを作成したユーザーに委譲すること
it "delegates name to the user who created it" do
  user = double("user", name: "Fake User")
  note = Note.new
  allow(note).to receive(:user).and_return(user)
  expect(note.user_name).to eq "Fake User"
end
```

ここでは永続化したユーザーオブジェクトをテストダブルに置き換えています。テストダブルは本物のユーザーではありません。実際、このオブジェクトを調べてみると、Double という名前のクラスになっていることに気づくはずです。テストダブルは name というリクエストに反応する方法しか知りません。説明のためにエクスペクテーションをテストに一つ追加してみてください。

spec/models/note_spec.rb

```
# 名前の取得をメモを作成したユーザーに委譲すること
it "delegates name to the user who created it" do
  user = double("user", name: "Fake User")
  note = Note.new
  allow(note).to receive(:user).and_return(user)
  expect(note.user_name).to eq "Fake User"
  expect(note.user.first_name).to eq "Fake"
end
```

このテストコードは元のコード（つまりモックに置き換える前のコード）であればパスしますが、このコードでは失敗します。

```
1) Note delegates name to the user who created it
   Failure/Error: expect(note.user.first_name).to eq "Fake"
     #<Double "user"> received unexpected message :first_name with
      (no args)
```

テストダブルは name に反応する方法 しか 知りません。なぜなら Note モデルが動作するために必要なコードはそれだけだからです。というわけで、先ほど追加した expect は削除してください。

さて、次はスタブについて見てみましょう。スタブは allow を使って作成しました。この行はテストランナーに対して、このテスト内のどこかで note.user を呼び出すことを伝えています。実際に user.name が呼ばれると、note.user_id の値を使ってデータベース上の該当するユーザーを検索し、見つかったユーザーを返却する代わりに、user という名前のテストダブルを返すだけになります。

結果として私たちは、テスト対象のモデルの外部に存在する実装の詳細から独立し、なおかつ2つのデータベース呼び出しを取り除いたテストを手に入れることができました。このテストはユーザーを永続化することもありませんし、データベース上のユーザーを検索しにいくこともありません。

このアプローチに対する非常に一般的で正しい批判は、もし User#name というメソッドの名前を変えたり、このメソッドを削除したりしても、このテストはパスし続けてしまう、という点です。みなさんも実際に試してみてください。User#name メソッドをコメントアウトし、テストを実行してみるとどうなるでしょうか？

ベーシックな RSpec のテストダブルは、代役になろうとするオブジェクトにスタブ化しようとするメソッドが存在するかどうかを検証しません。この問題を防止するには、かわりに

検証機能付きのテストダブル（verified double）を使用します。このテストダブルがUserのインスタンスと同じように振る舞うことを検証してみましょう（ここではUserの最初の1文字が大文字になっている点に注目してください。これは検証機能付きのテストダブルが動作するために必要な変更点です）。

spec/models/note_spec.rb

```
# 名前の取得をメモを作成したユーザーに委譲すること
it "delegates name to the user who created it" do
  user = instance_double("User", name: "Fake User")
  note = Note.new
  allow(note).to receive(:user).and_return(user)
  expect(note.user_name).to eq "Fake User"
end
```

この状態でnameメソッドに何か変更を加えると、テストは失敗します。

```
1) Note delegates name to the user who created it
     Failure/Error: user = instance_double("User", name: "Fake User")
       the User class does not implement the instance method: name.
       Perhaps you meant to use `class_double` instead?
```

今回の実験では、別にclass_doubleを使おうとしていたわけではありません（訳注：エラーメッセージの最後に「もしかするとclass_doubleを使おうとしましたか？」という文言が表示されています）。ですので、コメントアウトしたnameメソッドを元に戻して、テストを元通りにすればOKです。

失敗メッセージに書かれているとおり、class_doubleを使うとテストダブルのクラスメソッドも検証することができます。

テスト内でオブジェクトをモック化するためにテストダブルを使う場合、できるかぎり検証機能付きのテストダブルを使うようにしてください。これを使えば、誤ってテストがパスしてしまう問題を回避することができます。

RSpecで構築された既存のテストスイートが存在するコードベースで開発したことがある人や、他のテストチュートリアルをやったことがある人は、コントローラのテストでモックやスタブが頻繁に使われていることに気づいたかもしれません。実際、私はコントローラのテストではデータベースにアクセスすることを過剰に避けようとする開発者を過去に何人か見てきました（正直に白状すると、私もやったことがあります）。

以下はデータベースにまったくアクセスせずにコントローラのメソッドをテストする例です。ここで使われているNoteコントローラのindexアクションはジェネレータが作った元

の index アクションから変更されています。具体的には、アクションの内部で Note モデルの search スコープを呼びだして結果を集め、それをブラウザに返却しています。この機能をまったくデータベースにアクセスしない形でテストしてみましょう。それがこちらです。

spec/controllers/notes_controller_spec.rb

```
require 'rails_helper'

RSpec.describe NotesController, type: :controller do
  let(:user) { double("user") }
  let(:project) { instance_double("Project", owner: user, id: "123") }

  before do
    allow(request.env["warden"]).to receive(:authenticate!).and_return(user)
    allow(controller).to receive(:current_user).and_return(user)
    allow(Project).to receive(:find).with("123").and_return(project)
  end

  describe "#index" do
    # 入力されたキーワードでメモを検索すること
    it "searches notes by the provided keyword" do
      expect(project).to receive_message_chain(:notes, :search).
        with("rotate tires")
      get :index,
        params: { project_id: project.id, term: "rotate tires" }
    end
  end
end
```

順を追ってコードを見ていきましょう。まず、let を利用してテストで使う user と project を遅延定義しています（let については第8章 で説明しました）。モック化された user に対してはメソッドをまったく呼び出さないので、ここでは問題なく通常のテストダブルを使うことができます。一方、project に関しては owner と id の属性を使うので、検証機能付きのテストダブルを使った方が安全です。

次に、before ブロックの中では最初に Devise が用意してくれる authenticate! と current_user メソッドをスタブ化しています。なぜなら、これはパスワードによって保護されたページだからです。さらに、Active Record が提供している Project.find メソッドもスタブ化しています。モック化するのは、データベースを検索するかわりにモックの project を返すため

です。これにより、Project.find(123) がテスト対象のコード内のどこかで呼ばれても、本物のプロジェクトオブジェクトではなく、テストダブルの project が代わりに返されるようになります。

最後に、コントローラのコードが期待どおりに動作することを検証しなければなりません。このケースでは、project に関連する notes が持つ search スコープが呼ばれることと、その際の検索キーワード（term ）が同名のパラメータの値に一致することを検証しています。この検証は以下のコードで実現しています。

spec/controllers/notes_controller_spec.rb

```
expect(project).to receive_message_chain(:notes, :search).
  with("rotate tires")
```

ここでは receive_message_chain を使って project.notes.search を参照しています。ここで覚えておいてほしいことが一つあります。それは、このエクスペクテーションはアプリケーションコードを実際に動かす 前に 追加しないとテストがパスしないということです。ここでは allow ではなく expect を使っているので、もしこのメッセージチェーン（連続したメソッド呼び出し）が呼び出され なかった 場合はテストが失敗します。

このテストコードや、テスト対象のコントローラのコードでちょっと遊んでみてください。テストを失敗させる方法を見つけて、RSpec がどんなメッセージを出力するのか調べてみましょう。

さて、ここでこの新しいテストコードの実用性について考えてみましょう。このテストは速いでしょうか？ええ、ずっと速いはずです。ユーザーとプロジェクトをデータベース内に作成し、ユーザーをログインさせ、必要なパラメータを使ってコントローラのアクションを呼び出すことに比べれば、このテストは間違いなく速いでしょう。テストスイートにこのようなテストがたくさんあるなら、データベース呼び出しを必要最小限にすることで、テストの実行がだんだん遅くなる問題を回避することができます。

一方、このコードはトリッキーでちょっと読みづらいです。Devise の認証処理をスタブ化する部分など、セットアップ処理のいくつかは定型的なコードなので、メソッドとして抽出できるかもしれません。ですが、このセットアップコードはアプリケーションコード内でオブジェクトを作成して保存する方法とはかなり異なるものです。こうなると、テストコードを読み解くのが大変です。特に初心者は間違いなく苦労することでしょう。

また、ここではモック化に関する有名な原則も無視しています。それは「自分で管理していないコードをモック化するな（Don’t mock what you don’t own[44]）」という原則です。つまり、ここでは私たちが自分で管理していないサードパーティライブラリの機能を二つモック化し

[44] https://8thlight.com/blog/eric-smith/2011/10/27/thats-not-yours.html

ています。具体的には Devise が提供している認証レイヤーと、Active Record の `find` メソッドです。この二つの機能をスピードの向上とコードの分離を目的としてモック化しています。私たちはこのサードパーティライブラリを自分たちで管理していないにも関わらずモック化しました。そのため、アプリケーションを壊してしまうような変更がライブラリに入っても、テストコードはその問題を報告してくれないかもしれません。

とはいえ、自分のアプリケーションで管理していないコードをモック化することに意味があるときもあります。たとえば、時間のかかるネットワーク呼び出しを実行する必要があったり、レートリミットを持つ外部 API とやりとりする必要があったりする場合、モック化はそうしたコストを最小化してくれます。テストと Rails の経験が増えてくると、そういったインターフェースと直接やりとりするコードを作成し、アプリケーション内でそのコードが使われている部分をスタブ化するテクニックを有益だと考えるようになるかもしれません。ですがその場合でも、自分が書いたコードを高コストなインターフェースと直接やりとりさせるテストもある程度残すようにしてください。この話題は本書の範疇（もしくは本書の小さなサンプルアプリケーションのスコープ）を超えてしまいますが…。

いろいろ話してきましたが、もしあなたがモックやスタブにあまり手を出したくないのであれば、それはそれで心配しないでください。本書でやっているように、基本的なテストには Ruby オブジェクトを使い、複雑なセットアップにはファクトリを使う、というやり方でも大丈夫です。スタブはトラブルの原因になることもあります。重要な機能を気軽にスタブ化してしまうと、結果として実際には何もテストしていないことになってしまいます。

テストがとても遅くなったり、再現の難しいデータ（たとえば次の章である程度実践的な例を使って説明する外部 API や web サービスなど）をテストしたりするのでなければ、オブジェクトやファクトリを普通に使うだけで十分かもしれません。

タグ

たとえば、これからこのサンプルアプリケーションに新しい機能を追加することになったとします。このときに書くテストコードにはいくつかのモデルとコントローラに対する単体テストと、ひとつの統合テストが含まれます。新しい機能を開発している最中はテストスイート 全体 を実行するのは避けたいと考えるでしょう。ですが、各テストを一つずつ実行していくのも面倒です。このような場合は、RSpec のタグ機能[45]が使えます。タグを使うと、特定のテストだけを実行し、それ以外はスキップするようにフラグを立てることができます。

こうした用途では `focus` という名前のタグがよく使われます。実行したいテストに対して

[45] https://rspec.info/features/3-12/rspec-core/command-line/tag/

次のようにしてタグを付けてください。

```
# クレジットカードを処理すること
it "processes a credit card", focus: true do
  # example の詳細
end
```

こうすると、コマンドラインから focus タグを持つスペックだけを実行することができます。

```
$ bundle exec rspec --tag focus
```

特定のタグを スキップ することもできます。たとえば、特別遅い統合テストを今だけ実行したくない場合は、それがわかるようなタグを遅いテストに付けて、それからチルダを付けてタグの指定をひっくり返します（訳注：シェルによっては `--tag "~slow"` のようにチルダ付きのタグをダブルクオートで囲む必要があります）。

```
$ bundle exec rspec --tag ~slow
```

こうすると slow というタグが付いたテスト 以外の 全テストを実行します。

タグは describe や context ブロックに付けることもできます。その場合はブロック内の全テストにそのタグが適用されます。

focus タグはコミットする前に忘れずに削除してください。また、新しい機能が完成したと判断する前に、必ずタグなしでテストスイート全体を実行するようにしてください。一方で、slow のようなタグはしばらくコードベースに残しておいても便利かもしれません。

もし、focus タグを頻繁に利用するようになったら、一つでも focus タグの付いたテストがあるときに、そのタグを利用するよう RSpec を設定することもできます。もし focus タグの付いた example が一つも見つからなければ、テストスイート全体を実行します。

spec/spec_helper.rb

```
RSpec.configure.do |config|
  config.filter_run_when_matching :focus
  # 他の設定が続く ...
end
```

特定のタグが付いた example は常にスキップするよう、RSpec を設定することもできます。たとえば次のように設定します。

spec/spec_helper.rb

```
RSpec.configure do |config|
  config.filter_run_excluding slow: true
  # 他の設定が続く ...
end
```

この場合でもコマンドラインから slow タグの付いたテストだけを明示的に実行することは可能です。

```
$ bundle exec rspec --tag slow
```

不要なテストを削除する

もし、あるテストが目的を果たし、今後の回帰テストでも使う必要がないと確信できるなら、そのテストを削除してください！もし、何かしらの理由でそれを 本当に 残したいと考えているが、日常的には実行する必要はないなら、そのスペックに skip のマークを付けてください。

```
# 大量のデータを読み込むこと
it "loads a lot of data" do
  # 今後不要（なのでスキップする）
  skip "no longer necessary"

  # スペックのコードが続く。ただし実行はされない
end
```

私はテストをコメントアウトするよりも、スキップする方を推奨します。なぜなら、スキップされたスペックはテストスイートを実行したときにそのことが表示されるため、スキッ

プしていることを忘れにくくなるからです。とはいえ、不必要なコードは単純に削除するのが一番良いのは間違いありません（ただし、その確信がある場合に限ります）。

RSpec 3.0より前の RSpec では、この機能は `pending` メソッドとして提供されていました。`pending` は今でも使えますが、機能は全く異なっています。現在の仕様では pending された spec はそのまま実行されます。そのテストが途中で失敗すると、pending（保留中）として表示されます。しかし、テストがパスすると、そのテストは失敗とみなされます。

テストを並列に実行する

私は実行に30分かかるテストスイートを見たことがあります。また、もっと遅いテストスイートがある話もたくさん耳にしました。すでに遅くなってしまったテストスイートを実行するよい方法は、ParallelTests gem[46] を使ってテストを並列に実行することです。ParallelTests を使うと、テストスイートが複数のコアに分割され、格段にスピードアップする可能性があります。30分かかるテストスイートを6コアに分けて実行すれば、7分ぐらいまで実行時間が短くなります。これは劇的な効果があります。特に、遅いテストを個別に改善していく時間がない場合に最適です。

私はこのアプローチが好きですが、注意も必要です。なぜならこのテクニックを使うと、怪しいテストの習慣を隠してしまうことがあるからです。もしかすると、遅くて高コストな統合テストを使いすぎたりしているのかもしれません。テストスイートを高速化するときは ParallelTests だけを頼るのではなく、本書で紹介したその他のテクニックも併用するようにしましょう。

Rails を取り外す

モックやタグの活用はどちらもテストスイートの実行時間を減らすためのものでした。しかし究極的には、処理を遅くしている大きな要因の一つは Rails 自身です。テストを実行するときは毎回 Rails の一部、もしくは全部を起動させる必要があります。Spring gem をインストールし（ただし、Rails 6.1以前であればデフォルトでインストールされています）、binstub を使って Spring 経由で RSpec を実行すれば、起動時間を短くすることができます。しかし、もし 本当に テストスイートを高速化したいなら、Rails を完全に取り外すこともできます。Spring を使うとフレームワークを読み込んでしまいますが（ただし一回だけ）、フレームワークを読み込まないこちらのソリューションなら、フレームワークをまったく読み込まないの

[46] https://github.com/grosser/parallel_tests

で、さらにもう一歩高速化させることができます。

この話題は本書の範疇を大きく超えてしまいます。なぜならあなたのアプリケーションアーキテクチャ全体を厳しく見直す必要があるからです。また、これは Rails 初心者に説明する際の個人的なルールも破ることになります。つまり、Rails の規約を破ることは可能な限り避けなければならない、という私自身のルールも破ることになるわけです。それでもこの内容について詳しく知りたいのであれば、Corey Haines の講演[47]とDestroy All Software[48]を視聴することをお勧めします。

まとめ

この章にやってくる前は、テストのやり方に複数の選択肢があることをほとんど説明してきませんでした。しかし、これであなたは複数の選択肢を選べるようになりました。あなたはあなた自身とあなたのチームにとって一番良い方法を選び、スペックを明快なドキュメントにすることができます。第3章 で説明したような冗長な書き方を使うこともできますし、ここで説明したようなもっと簡潔な方法を使うこともできます。あなたはモックとスタブ、ファクトリ、ベーシックな Ruby オブジェクト、それらの組み合わせ等々、様々な方法を選べるはずです。最後にあなたはテストスイートをロードし、それを実行するオプションも覚えました。

さあ、ゴールは目前です！あといくつかのカバーすべき話題を見たら、テストプロセスの全体像と落とし穴の避け方（これには初めて登場するテストファースト形式の開発プロセスも含まれます）を説明して本書を締めくくります。ですが、その前に典型的な web アプリケーションでよくある、私たちが まだテストしていない マイナーな機能について見ていきましょう。

演習問題

- あなたのテストスイートの中で、この章で説明した簡潔な構文を使ってきれいにできるスペックを探してください。この演習問題をやってみることで、あなたのスペックはどれくらい短くなるでしょうか？読みやすさも同じでしょうか？あなたは簡潔な書き方と冗長な書き方のどちらが好みでしょうか？（ヒント: 最後の問いには正解はありません。）

[47] https://www.youtube.com/watch?v=bNn6M2vqxHE
[48] https://www.destroyallsoftware.com/screencasts

- shoulda-matchers をインストールし、それを使ってスペックをきれいにできそうな場所を探してください（またはテストしていない部分をテストしてください）。GitHub にある gem のソースコード[49]をチェックし、これらのマッチャがどのように実装されているのか詳しく学習してください。
- モックを使って実験できそうな機会を探してみてください。必ずしも自分が管理しているコードでなくても構いません。最初はモデルのテストにだけ注目して小さく始めてください。それから、より複雑なシナリオに進んでいきましょう。モック化したテストコードは速くなりましたか？新しいテストコードは理解しやすいままですか？
- RSpec のタグを使って、遅いスペックに目印を付けてください。そのテストを含めたり除いたりしてテストスイートを実行してください。どれくらいパフォーマンスが向上するでしょうか？

[49] https://github.com/thoughtbot/shoulda-matchers

10. その他のテスト

私たちはこれまで、プロジェクト管理アプリケーション全体に対して堅牢なテストスイートを構築してきました。モデルとコントローラをテストし、システムスペックを通じて view と組み合わせた場合もテストしました。リクエストスペックを使った、外部向け API のテストもあります。しかし、まだカバーしていないテストが少し残っています。メモ（Note）モデルにあるファイルアップロード機能や、メール送信機能、ジオコーディング（緯度経度）連携のテストはどうすればいいでしょうか？こうした機能もテストできます！

この章では次のような内容を説明します。

- ファイルアップロードのテスト
- Active Job を使ったバックグラウンドジョブのテスト
- メール送信のテスト方法
- 外部 Web サービスに対するテスト

この章で発生する変更点全体は GitHub 上のこの diff[50] 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-10-testing-the-rest origin/09-test-faster
```

ファイルアップロードのテスト

添付ファイルのアップロードは Web アプリケーションでよくある機能要件の一つです。Rails 5.2からは Active Storage と呼ばれるファイルアップロード機能が標準で提供されるようになりました。本書のサンプルアプリケーションでも Active Storage を使用しています。このほかにも CarrierWave や Shrine といった gem を使ってファイルアップロード機能を実装する方法もあります（もしくは自分で独自に作るという手もあります）。しかし、どうやってテストすれば良いでしょうか？本書では Active Storage を使ったテスト方法を紹介しますが、基本的なアプローチは他のライブラリを使う場合でも同じです。では、メモ機能をテストするシステムスペックから始めましょう。

[50] https://github.com/JunichiIto/everydayrails-rspec-jp-2024/compare/09-test-faster...10-testing-the-rest

spec/system/notes_spec.rb

```
require 'rails_helper'

RSpec.describe "Notes", type: :system do
  let(:user) { FactoryBot.create(:user) }
  let(:project) {
    FactoryBot.create(:project,
      name: "RSpec tutorial",
      owner: user)
  }

  # ユーザーが添付ファイルをアップロードする
  scenario "user uploads an attachment" do
    sign_in user
    visit project_path(project)
    click_link "Add Note"
    fill_in "Message", with: "My book cover"
    attach_file "Attachment", "#{Rails.root}/spec/files/attachment.jpg"
    click_button "Create Note"
    expect(page).to have_content "Note was successfully created"
    expect(page).to have_content "My book cover"
    expect(page).to have_content "attachment.jpg (image/jpeg"
  end
end
```

これは今まで書いてきた他のシステムスペックととてもよく似ています。ですが、今回はCapybara の `attach_file` メソッドを使って、ファイルを添付する処理をシミュレートしています。最初の引数は入力項目のラベルで、二つ目の引数がテストファイルのパスです。ここでは *spec/files* という新しいディレクトリが登場しています。この中にテストで使う小さなJPEG ファイルが格納されています。このディレクトリ名は何でも自由に付けられます。他にはたとえば、*spec/fixtures* という名前が付けられることもあるようです。ファイルの名前も自由です。ただし、このファイルは忘れずにバージョン管理システムにコミットしておいてください。そうしないと他の開発者がテストを実行したときに、テストが失敗してしまうからです。テストファイルが指定された場所に存在すれば、この新しいスペックは問題なくパスするはずです。

ところで、Active Storage では *config/storage.yml* でアップロードしたファイルの保存先を指定します。特に変更していなければ、テスト環境では *tmp/storage* になっているはずです。

config/storage.yml

```
test:
  service: Disk
  root: <%= Rails.root.join("tmp/storage") %>

local:
  service: Disk
  root: <%= Rails.root.join("storage") %>
```

tmp/storage ディレクトリを開いて、なんらかのディレクトリやファイルが作成されていることを確認してみてください。ただし、ディレトリ名やファイルは推測されにくい名前になっているため、どのファイルがどのテストで作成されたのか探し当てるのは少し難しいかもしれません。今のままだと、テストを実行するたびにファイルが増えていってしまうため、テストの実行が終わったら、アップロードされた古いファイルは自動的に削除されるようにしておくと良いと思います。*spec/rails_helper.rb* に以下の設定を追加してください。

spec/rails_helper.rb

```
RSpec.configure do |config|
  # このブロック内の他の設定は省略 ...

  # テストスイートの実行が終わったらアップロードされたファイルを削除する
  config.after(:suite) do
    Pathname(ActiveStorage::Blob.service.root).each_child do |path|
      path.rmtree if path.directory?
    end
  end
end
```

さあ、これでテストが終わった時点で RSpec がこのディレクトリとその中身を削除してくれます。さらに、偶然この中のファイルがバージョン管理システムにコミットされてしまわないよう、プロジェクトの *.gitignore* に設定を追加しておくことも重要です。Active Storage の場合は Rails が最初から設定を追加してくれているので、念のため確認するだけで OK ですが、他のライブラリを使ってアップロード機能を実現するときはこうした設定追加を忘れないようにしてください。

.gitignore

```
# Ignore uploaded files in development.
/storage/*
!/storage/.keep
/tmp/storage/*
!/tmp/storage/
!/tmp/storage/.keep
```

さて、Active Storage のことはいったん忘れて、ここでやった内容を振り返ってみましょう。最初はファイルアップロードのステップを含む、スペックファイルを作成しました。さらに、スペック内ではテストで使うファイルを添付しました。次に、テスト専用のアップロードパスを設定（確認）しました。最後に、テストスイートが終わったらファイルを削除するように RSpec を設定しました。この3つのステップはシステムレベルのテストでファイルアップロードを実行するための基本的なステップです。もしみなさんが Active Storage を使っていない場合は、自分が選んだアップロード用ライブラリのドキュメントを読み、この3つのステップを自分のアプリケーションに適用する方法を確認してください。もし自分でファイルアップロードを処理するコードを書いている場合は、アップロード先のディレクトリを変更するために、`allow` メソッド（第9章 を参照）でスタブ化できるかもしれません。もしくはアップロード先のパスを実行環境ごとの設定値として抜き出す方法もあるでしょう。

最後に、もしテスト内で何度もファイル添付を繰り返しているのであれば、そのモデルをテストで使う際に、添付ファイルを属性値に含めてしまうことを検討した方がいいかもしれません。たとえば、Note ファクトリにこのアプローチを適用する場合は、第4章 で説明したトレイトを使って実現することができます。

spec/factories/notes.rb

```
1  FactoryBot.define do
2    factory :note do
3      message { "My important note." }
4      association :project
5      user { project.owner }
6
7      trait :with_attachment do
8        attachment { Rack::Test::UploadedFile.new( \
9          "#{Rails.root}/spec/files/attachment.jpg", 'image/jpeg') }
10     end
11   end
12 end
```

Active Storage を使っている場合は上のように `Rack::Test::UploadedFile.new` の引数に添付ファイルが存在するパスと Content-Type を指定します。こうすればテスト内で `FactoryBot.create(:note, :with_attachment)` のように書くことで、ファイルが最初から添付された新しい Note オブジェクトを作成することができます。モデルスペックを開き、次のテストを追加してください。

spec/models/note_spec.rb

```
require 'rails_helper'

RSpec.describe Note, type: :model do
  # 他のテストは省略 ...

  # 添付ファイルを1件添付できる
  it 'has one attached attachment' do
    note = FactoryBot.create :note, :with_attachment
    expect(note.attachment).to be_attached
  end
```

このアプローチはよく考えて使ってください！このファクトリは使うたびにファイルシステムへの書き込みが発生するので、遅いテストの原因になります。

バックグラウンドワーカーのテスト

私たちのプロジェクト管理アプリケーションを利用している、架空の会社の架空のマーケティング部が、CM を流したり、広告を打ったりする参考情報を得るために、私たちに対してユーザーに関する位置情報を集めてくるように依頼してきたとします。個人情報の扱いや利用規約に関する詳細は法務部に任せるとして、とりあえず私たちは `location` 属性をユーザーモデルに追加してこの機能を完成させました。この機能はユーザーがログインしたときに外部のジオコーディングサービスにアクセスし、町や州、国といった情報を取得します。今回新たに加わったこの処理は Active Job を使ってバックグラウンドで実行されます。そのため、ユーザーはこの処理が終わるまで待たされることはありません。

この機能はサンプルアプリケーションに実装済みですが、Rails の開発環境では少しトリッキーな実装になっています。なぜなら、開発環境ではログインしたときに localhost や 127.0.0.1 になってしまうからです。私は偽物のランダムな IP アドレスを持ったサンプルデータ（シードデータ）を追加しています。これは Rails コンソールで試すことが可能です。

まず、bin/rails db:seed を実行し、シードデータを開発環境に追加してください。

それから bin/rails console で Rails コンソールを開き、ユーザーを一人選んでください。ユーザーを選んだら、geocode メソッドを呼んでください。

この

```
> u = User.find(10)
> u.location
 => nil
> u.geocode
 => true
> u.location
 => "Johannesburg, Gauteng, South Africa"
```

ような遅い処理をバックグラウンドで実行するのは、比較的シンプルかつ効果的な方法で、Web アプリケーションのパフォーマンスを高く保つことができます。ですが、メインフローからプロセスを切り分けるということは、テストの方法を少し変える必要があることを意味します。幸いなことに、Rails と rspec-rails はさまざまなレベルのテストにおいて、Active Job ワーカーをテストする、便利なサポート機能を提供してくれています。

まず、統合テストから始めましょう。私たちはまだユーザーのログイン処理を明示的にテストしていません。そこで、bin/rails g rspec:system sign_in を実行して、テストを作成します。それから、次のようなテストコードを書いてください。

spec/system/sign_ins_spec.rb

```
require 'rails_helper'

RSpec.describe "Sign in", type: :system do
  let(:user) { FactoryBot.create(:user) }

  before do
    ActiveJob::Base.queue_adapter = :test
  end

  # ユーザーのログイン
  scenario "user signs in" do
    visit root_path
    click_link "Sign In"
    fill_in "Email", with: user.email
    fill_in "Password", with: user.password
    click_button "Log in"

    expect {
      GeocodeUserJob.perform_later(user)
    }.to have_enqueued_job.with(user)
  end
end
```

このコードには Active Job とバックグラウンドのジオコーディング処理に関連する部分が2箇所あります。まず、rspec-rails ではバックグラウンドジョブをテストするために、`queue_adapter` に `:test` を指定する必要があります。これがないとテストは次のような例外を理由付きで raise します（訳注：下の例外メッセージには「ActiveJob マッチャを使うには、`ActiveJob::Base.queue_adapter = :test` を設定してください」と書いてあります）。

```
StandardError:
  To use ActiveJob matchers set `ActiveJob::Base.queue_adapter = :test`
```

ここではよく目立つように `before` ブロックでこのコードを実行しましたが、`scenario` の中で実行しても構いません。なぜなら、このファイルにはテストが一つしかないからです。また、複数のファイルでテストキューを使う場合は、第8章 で紹介したテストを DRY にするテクニック（訳注：`shared_context` のこと）を使って実験することもできます。

次に、ジョブが実際にキューに追加されたことを確認する必要があります。rspec-rails ではこの確認に使えるマッチャがいくつか用意されています。ここでは have_enqueued_job を使い、正しいジョブが正しい入力値で呼ばれていることをチェックしています。注意してほしいのは、このマッチャはブロックスタイルの expect と組み合わせなければいけないことです。こうしないと、テストは次のような別の例外を理由付きで raise します（訳注：下の例外メッセージには「have_enqueued_job と enqueue_job はブロック形式のエクスペクテーションだけをサポートします」と書いてあります）。

```
ArgumentError:
  have_enqueued_job and enqueue_job only support block expectations
```

have_enqueued_job マッチャはチェイニングする（他のマッチャと連結させる）ことにより、キューの優先度やスケジュール時間を指定することもできます。このマッチャのオンラインドキュメント[51] には数多くの実行例が載っています。

さて、これでバックグラウンドジョブがアプリケーションの他の部分と正しく連携できていることをチェックできました。今度はもっと低レベルのテストを書いて、ジョブがアプリケーション内のコードを適切に呼びだしていることを確認しましょう。このジョブをテストするために、新しいテストファイルを作成します。

```
bin/rails g rspec:job geocode_user
```

この新しいファイルは本書を通じて作成してきた他のテストファイルとよく似ています。

spec/jobs/geocode_user_job_spec.rb

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

では、このジョブが User モデルの geocode メソッドを呼んでいることをテストしましょう。pending になっているテストを次のように書き換えてください。

[51] http://www.rubydoc.info/gems/rspec-rails/RSpec%2FRails%2FMatchers%3Ahave_enqueued_job

spec/jobs/geocode_user_job_spec.rb

```
require 'rails_helper'

RSpec.describe GeocodeUserJob, type: :job do
  # user の geocode を呼ぶこと
  it "calls geocode on the user" do
    user = instance_double("User")
    expect(user).to receive(:geocode)
    GeocodeUserJob.perform_now(user)
  end
end
```

このテストでは第9章で説明した instance_double を使ってテスト用のモックユーザーを作っています。それからテスト実行中のどこかのタイミングでこのモックユーザーに対して geocode メソッドが呼び出されることを RSpec に伝えています。最後に、perform_now メソッドを使って、このバックグラウンドジョブ自身を呼び出します。こうすると、ジョブはキューに入らないため、テストの実行結果をすぐに検証できます。

この geocode というインスタンスメソッドは Geocoder gem によって定義されています。サンプルアプリケーションでは User モデル内で geocoded_by というメソッドを使ってこの gem を利用しています。このテストでは Geocoder のことを詳しく知らなくても問題ありませんが、練習のためにもっとテストを書いてみたいと思った場合は、この gem のドキュメント[52] を読むとよく理解できるはずです。

メール送信をテストする

フルスタック Web アプリケーションの多くは何らかのメールをユーザーに送信します。このサンプルアプリケーションでは短いウェルカムメッセージに、アプリケーションのちょっとした使い方を添えて送信します。メール送信は二つのレベルでテストできます。一つはメッセージがただしく構築されているかどうかで、もう一つは正しい宛先にちゃんと送信されるかどうかです。このテストでは先ほど新しく学んだ、バックグラウンドワーカーのテストの知識を使います。なぜならメール送信はバックグラウンドワーカーを利用することが一般的だからです。

最初は Mailer にフォーカスしたテストから始めましょう。今回対象となる Mailer は *app/mailers/user_mailer.rb* にある Mailer です。このレベルのテストでは、送信者と受信者のア

[52] http://www.rubygeocoder.com

ドレスが正しいことと、件名とメッセージ本文に大事な文言が含まれていることをテストします。（もっと複雑な Mailer になると、今回の基本的な Mailer よりも、もっとたくさんの項目をテストすることになるはずです。）

新しいテストファイルはジェネレータを使って作成します。

```
bin/rails g rspec:mailer user_mailer
```

この新しいファイルで注目したいのは、*spec/mailers/user_mailer.rb* にファイルが作成されている点と、次のような定型コードが書かれている点です。

spec/mailers/user_mailer_spec.rb

```
require "rails_helper"

RSpec.describe UserMailer, type: :mailer do
  pending "add some examples to (or delete) #{__FILE__}"
end
```

ここに以下のようなコードを書いていきましょう。

spec/mailers/user_mailer_spec.rb

```
require "rails_helper"

RSpec.describe UserMailer, type: :mailer do
  describe "welcome_email" do
    let(:user) { FactoryBot.create(:user) }
    let(:mail) { UserMailer.welcome_email(user) }

    # ウェルカムメールをユーザーのメールアドレスに送信すること
    it "sends a welcome email to the user's email address" do
      expect(mail.to).to eq [user.email]
    end

    # サポート用のメールアドレスから送信すること
    it "sends from the support email address" do
      expect(mail.from).to eq ["support@example.com"]
    end

    # 正しい件名で送信すること
    it "sends with the correct subject" do
      expect(mail.subject).to eq "Welcome to Projects!"
```

```
21      end
22
23      # ユーザーにはファーストネームであいさつすること
24      it "greets the user by first name" do
25        expect(mail.body).to match(/Hello #{user.first_name},/)
26      end
27
28      # 登録したユーザーのメールアドレスを残しておくこと
29      it "reminds the user of the registered email address" do
30        expect(mail.body).to match user.email
31      end
32    end
33  end
```

ここではテストデータ、すなわち、テスト対象のユーザーと Mailer のセットアップから始まっています。それから実装された仕様を確認するための小さな単体テストをいくつか書いています。最初に書いたのは `mail.to` のアドレスを確認するテストです。ここで注意してほしいのは、`mail.to` の値は文字列の配列になる点です。単体の文字列ではありません。`mail.from` も同様にテストします。このテストでは RSpec の `contain` マッチャを使うこともできますが、私は配列の等値性をチェックする方が好みです。こうすると余計な受信者や送信者が含まれていないことを確実に検証できます。

`mail.subject` のテストはとても単純だと思います。ここまでは `eq` マッチャを使って何度も文字列を比較してきました。ですが、`mail.body` を検証する最後の二つのテストでは `match` マッチャを使っている点が少し変わっています。このテストではメッセージ本文全体をテストする必要はなく、本文の一部をテストすればいいだけです。最初の example では正規表現を使って、フレンドリーなあいさつ（たとえば、*Hello, Maggie,* のようなあいさつ）が本文に含まれていることを確認しています。二つ目の example でも `match` を使っていますが、この場合は正規表現を使わずに、本文のどこかに `user.email` の文字列が含まれることを確認しているだけです。

もしメールに関連するスペックをもっと表現力豊かに書きたいのであれば、Email Spec[53] というライブラリをチェックしてみてください。このライブラリは `deliver_to` や `have_body_text` といったマッチャを提供してくれます。これを使えば、みなさんのテストがもっと読みやすくなるかもしれません。

繰り返しになりますが、みなさんが書く Mailer のスペックの複雑さは、みなさんが作った

[53] http://rubygems.org/gems/email_spec

Mailer の複雑さ次第です。今回はこれぐらいで十分かもしれませんが、もっとたくさんテストを書いた方が Mailer に対して自信が持てるのであれば、そうしても構いません。

さて、今度はアプリケーションの大きなコンテキストの中で Mailer をテストする方法を見ていきましょう。このアプリケーションでは新しいユーザーが作られると、そのたびにウェルカムメールが送信される仕様になっています。どうすれば、本当に送信されていることを検証できるでしょうか？高いレベルでテストするなら統合テストでテストできますし、もう少し低いレベルでテストするならモデルレベルでテストできます。練習として、両方のやり方を見ていきましょう。最初は統合テストから始めます。

このメッセージはユーザーのサインアップワークフローの一部として送信されます。それではこのテストを書くための新しいシステムスペックを作成しましょう。

```
bin/rails g rspec:system sign_up
```

次に、ユーザーがサインアップするためのステップと期待される結果をこのファイルに追加します。

spec/system/sign_ups_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe "Sign-ups", type: :system do
4    include ActiveJob::TestHelper
5
6    # ユーザーはサインアップに成功する
7    scenario "user successfully signs up" do
8      visit root_path
9      click_link "Sign up"
10
11     perform_enqueued_jobs do
12       expect {
13         fill_in "First name", with: "First"
14         fill_in "Last name", with: "Last"
15         fill_in "Email", with: "test@example.com"
16         fill_in "Password", with: "test123"
17         fill_in "Password confirmation", with: "test123"
18         click_button "Sign up"
19       }.to change(User, :count).by(1)
20
21       expect(page).to \
```

```
22          have_content "Welcome! You have signed up successfully."
23        expect(current_path).to eq root_path
24        expect(page).to have_content "First Last"
25      end
26
27      mail = ActionMailer::Base.deliveries.last
28
29      aggregate_failures do
30        expect(mail.to).to eq ["test@example.com"]
31        expect(mail.from).to eq ["support@example.com"]
32        expect(mail.subject).to eq "Welcome to Projects!"
33        expect(mail.body).to match "Hello First,"
34        expect(mail.body).to match "test@example.com"
35      end
36    end
37  end
```

このテストの約3分の2は第6章 以降でよく見慣れたものかもしれません。ここではCapybaraを使ってサインアップフォームへの入力をシミュレートしています。残りの部分はメール送信にフォーカスしています。メールはバックグラウンドプロセスで送信されるため、テストコードは `perform_enqueued_jobs` ブロックで囲む必要があります。このヘルパーメソッドは、このスペックファイルの最初で include している `ActiveJob::TestHelper` モジュールが提供しています。

このメソッドを使えば `ActionMailer::Base.deliveries` にアクセスし、最後の値を取ってくることができます。この場合、最後の値はユーザーが登録フォームに入力したあとに送信されるウェルカムメールになります。テストしたいメールオブジェクトを取得できれば、残りのエクスペクテーションは Mailer に追加した単体テストとほとんど同じです。

実際のところ、統合テストのレベルではここまで詳細なテストは必要ないかもしれません。ここでは `mail.to` をチェックして適切なユーザーにメールが送信されていることと、`mail.subject` をチェックして適切なメッセージが送信されていることを検証するだけで十分かもしれません。なぜならその他の詳細なテストは Mailer のスペックに書いてあるからです。これはただ私が RSpec にはこういうやり方もあるということを紹介したかっただけです。

> 日本語版のサンプルアプリケーションには letter_opener_web[a] を組み込んであります。ブラウザ上でサインアップした場合、http://localhost:3000/letter_opener にアクセスするとサ

インアップ時に送信されたウェルカムメールの内容を確認できます。

[4] https://github.com/fgrehm/letter_opener_web

このテストはUserモデルとUserMailerが連携するポイントを直接テストすることでも実現できます。このアプリケーションでは新しいユーザーが追加されたときにafter_createコールバックでこの連携処理が発生するようになっています。なので、ユーザーのモデルスペックにテストを追加することができます。

spec/models/user_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe User, type: :model do
4    # これより前にある example は省略 ...
5
6    # アカウントが作成されたときにウェルカムメールを送信すること
7    it "sends a welcome email on account creation" do
8      allow(UserMailer).to \
9        receive_message_chain(:welcome_email, :deliver_later)
10     user = FactoryBot.create(:user)
11     expect(UserMailer).to have_received(:welcome_email).with(user)
12   end
13 end
```

このテストではまず、receive_message_chainを使ってdeliver_laterメソッドをスタブ化しています。このメソッドはActive JobがUserMailer.welcome_emailに提供してくれるメソッドです。receive_message_chainメソッドについては、第9章でも同じような使い方を説明しています。

次に、テストするユーザーを作成する必要があります。Mailerはユーザーを作成する処理の一部として実行されるため、その直後にスパイ（spy）を使ってテストします。スパイは第9章で説明したテストダブルによく似ていますが、テストしたいコードが実行されたあとに発生した何かを確認できる点が異なります。具体的には、クラス（UserMailer）に対して、期待されたオブジェクト（user）とともに、目的のメッセージ（:welcome_email）が呼び出されたかどうかを、have_receivedマッチャを使って検証しています。

なぜスパイを使う必要があるのでしょうか？これは次のような問題を回避するためです。すなわち、ここではユーザーを作成してそれを変数に入れる必要がありますが、そうするとテストしようとしているMailerも実行されてしまいます。

つまり、次のようなコードを書いても動かないということです。

```
# アカウントが作成されたときにウェルカムメールを送信すること
it "sends a welcome email on account creation" do
  expect(UserMailer).to receive(:welcome_email).with(user)
  user = FactoryBot.create(:user)
end
```

これは以下のエラーを出して失敗します。

```
Failures:

  1) User sends a welcome email on account creation
     Failure/Error: expect(UserMailer).to
       receive(:welcome_email).with(user)

     NameError:
       undefined local variable or method `user' for
       #<RSpec::ExampleGroups::User:0x007fca733cb578>
```

user という変数が作成される前にこの変数を使うことはできません。ですが、テスト対象のコードを実行せずに user を作成することもできません。幸いなことに、スパイを使えばどちらでもない新たなワークフローを選択することができます。

このテストのいいところは、Mailer が正しい場所で、正しい情報とともに呼ばれることを確認する だけ で十分であることです。このテストではいちいち Web 画面を使ってユーザーを作成する必要がありません。そのため速く実行できます。ですが、短所もあります。このテストはレガシーコードをテストするために提供されている RSpec のメソッド[54]を使っている点です。また、スパイを使うとこのテストコードを初めて見た開発者をびっくりさせてしまうかもしれません。

また、このワークフローを一度見直してみるのもいいかもしれません。たとえば deliver_later を使ってウェルカムメールを送信するのではなく、別のバックグラウンドジョブを使って送信するようにしたり、after_create コールバックを削除したりすることも検討してみましょう。このようなケースでは高いレベルの統合テストを残した状態で低レベルのテストを作成し、アプリケーションコードが適切に連携できていることを確認できたら、統合テストの方は削除してもいいかもしれません。繰り返しになりますが、開発者には選択肢がいろいろあるのです。

[54] https://rspec.info/features/3-12/rspec-mocks/working-with-legacy-code/message-chains/

Web サービスをテストする

ではジオコーディングに戻りましょう。バックグラウンドジョブのテストは作成しましたが、実装の詳細はまだちゃんとテストできていません。ジオコーディングの処理は本当に実行されているのでしょうか？現在の実装コードでは、ユーザーが正常にログインしたあとにジオコーディングがバックグラウンドで実行されます。ですが、それはユーザーが実行中のアプリケーションサーバーと異なるホストからログインした場合だけです。位置情報は IP アドレスに基づいてリクエストされます。そして、繰り返しになりますが、ジオコーディングはアプリケーションの中で実行されているのではありません。この処理は HTTP コールを経由して外部のジオコーディングサービスに実行してもらっているのです。

そこでテストを追加して本当にジオコーディングサービスにアクセスしていることを検証しましょう。既存の User モデルのスペックに新しいテストを追加してください。

spec/models/user_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe User, type: :model do
4    # これより前のテストは省略 ...
5
6    # ジオコーディングを実行すること
7    it "performs geocoding" do
8      user = FactoryBot.create(:user, last_sign_in_ip: "161.185.207.20")
9      expect {
10       user.geocode
11     }.to change(user, :location).
12       from(nil).
13       to("New York City, New York, US")
14   end
15 end
```

このテストは第3章で書いた他のテストによく似ています。静的な IP アドレスを事前に設定したユーザーを作成し、ユーザーに対してジオコーディングを実行し、ユーザーの位置情報が取得できたことを検証しています（ちなみに私は上の実験でこの IP アドレスに対応する位置情報をたまたま知りました）。

スペックを実行すると、このテストはパスします（訳注：ジオコーディングサービスが返す位置情報が変更されるとテストが失敗することがあります。その場合はテストコードを修

正してテストがパスするようにしてください)。ですが、一つ問題があります。この小さなスペックは同じファイルに書かれた他のテストよりも明らかに遅いのです。なぜだかわかりますか？このテストではジオコーディングサービスに対して実際に HTTP リクエストを投げます。そのため、サービスが位置情報を返すまで待ってから新しい値を確認しなければならないのです。

実行速度が遅くなることに加えて、外部のサービスを直接使ってテストすることは別のコストも発生させます。もしこの外部 API にレートリミットが設定されていたら、レートリミットを超えたタイミングでテストが失敗し始めます。また、これが有料のサービスだった場合、テストを実行することで実際に利用料金が発生してしまうかもしれません！

VCR[55] gem はこういった問題を軽減してくれる素晴らしいツールです。VCR を使えばテストを高速に保ち、API の呼び出しを必要最小限に抑えることができます。VCR は Ruby コードから送られてくる外部への HTTP リクエストを監視します。そうした HTTP リクエストが必要なテストが実行されると、そのテストは失敗します。テストをパスさせるには、HTTP 通信を記録する “カセット” を作る必要があります。テストをもう一度実行すると、VCR はリクエストとレスポンスをファイルに記録します。こうすると今後、同じリクエストを投げるテストはファイルから読み取ったデータを使うようになります。外部の API に新たなリクエストを投げることはありません。

では VCR をアプリケーションに追加することから始めましょう。まず、この VCR と VCR によって利用される WebMock を *Gemfile* に追加してください。

Gemfile

```
group :test do
  # 他のテスト用 gem は省略 ...
  gem 'vcr'
  gem 'webmock'
end
```

> 日本語版のサンプルアプリケーションでは最初から Gemfile にこれらの gem を追加してあるので、*Gemfile* を編集する必要はありません。

WebMock[56]は HTTP をスタブ化するライブラリで、VCR は処理を実行するたびにこのライブラリを水面下で利用しています。WebMock はそれ自体がパワフルなツールですが、説明を

[55] https://github.com/vcr/vcr
[56] https://github.com/bblimke/webmock

シンプルにするため、ここでは詳しく説明しません。`bundle install` を実行して、新しい gem を追加してください。

次に、外部に HTTP リクエストを送信するテストで VCR を使うように RSpec を設定する必要があります。*spec/support/vcr.rb* というファイルを新たに作成し、次のようなコードを追加してください。

spec/support/vcr.rb

```
require "vcr"

VCR.configure do |config|
  config.cassette_library_dir = "#{::Rails.root}/spec/cassettes"
  config.hook_into :webmock
  config.ignore_localhost = true
  config.configure_rspec_metadata!
end
```

この新しい設定ファイルでは カセット（cassette） 、すなわち記録されたやりとりをアプリケーションの *spec/cassettes* ディレクトリに保存するように設定しています。すでに説明したとおり、スタブ化には WebMock を使います（ですが、VCR は他の数多くのスタブライブラリをサポートしています）。タスクを完了済みにする AJAX コールなどで使われる `localhost` へのリクエストは無視します。最後に、RSpec のタグによって VCR が有効になるようにします。JavaScript を実行するテストに `js: true` を付けたのと同じように、VCR を使うテストでは `vcr: true` を付けることで VCR が有効化されます。

この設定が済んだら、新しいテストを実行して何が起きるか確認してください。テストの実行はずっと速くなりますが、次のように失敗してしまいます。

```
Failures:

  1) User performs geocoding
     Failure/Error: user.geocode

     VCR::Errors::UnhandledHTTPRequestError:


       ================================================================================
       An HTTP request has been made that VCR does not know how to handle:
         GET http://ipinfo.io/161.185.207.20/geo
```

```
There is currently no cassette in use. There are a few ways
you can configure VCR to handle this request:
```

VCR が設定されていると、外部への HTTP リクエストは `UnhandledHTTPRequestError` という例外を起こして失敗するようになります。この問題を解消するために、`vcr: true` というオプションをテストに追加してください。

spec/models/user_spec.rb

```
1  # ジオコーディングを実行すること
2  it "performs geocoding", vcr: true do
3    user = FactoryBot.create(:user, last_sign_in_ip: "161.185.207.20")
4    expect {
5      user.geocode
6    }.to change(user, :location).
7      from(nil).
8      to("New York City, New York, US")
9  end
```

テストを再実行すると、二つの変化が起きるはずです。一点目はテストがパスするようになることです。二点目はリクエストとレスポンスの記録が、先ほど設定した *spec/cassettes* ディレクトリのファイルに保存されます。このファイルをちょっと覗いてみてください。このファイルは YAML ファイルになっていて、最初にリクエストのパラメータが記録され、そのあとにジオコーディングサービスから返ってきたパラメータが記録されています。テストをもう一度実行してください。依然としてテストはパスしますが、今回はカセットファイルの内容を使って実際の HTTP リクエストとレスポンスをスタブ化します。

この例ではモデルスペックで実行された HTTP トランザクションを記録するために VCR を使いましたが、テストの一部で HTTP トランザクションが発生するテストであれば、どのレイヤーのどのテストでも VCR を利用することができます。

私は VCR が大好きですが、VCR には短所もあります。特に、カセットが古びてしまう問題には注意が必要です。これはつまり、もしテストに使っている外部 API の仕様が変わってしまっても、あなたはカセットが古くなっていることを知る術がない、ということです。それを知ることができる唯一の方法は、カセットを削除して、もう一度テストを実行することです。Rails の開発者の多くはプロジェクトのバージョン管理システムからカセットファイルを除外することを選択します。これは新しい開発者が最初にテストスイートを実行した際に、必ず自分でカセットを記録するようにするためです。この方法を採用する前に、一定の頻度でカセットを自動的に再記録する方法[57]を検討しても良いかもしれません。この方法を使え

[57] https://benoittgt.github.io/vcr/#/cassettes/automatic_re_recording

ば、後方互換性のない API の変更を比較的早く検知することができます。

また、二つ目の注意点として、API のシークレットトークンやユーザーの個人情報といった機密情報をカセットに含めないようにしてください。VCR にはサニタイズ用のオプション[58]が用意されているので、これを使えば機密情報がファイルに保存される問題を回避することができます。もし、みなさんがカセットをバージョン管理システムに あえて 保存するのであれば、この点は非常に重要です。

まとめ

メールやファイルアップロード、web サービス、バックグラウンドプロセスといった機能は、あなたのアプリケーションの中では些細な機能かもしれません。しかし、必要に応じてその機能をテストする時間も作ってください。なぜなら、この先何が起こるかわからないからです。その web サービスがあるときからアプリケーションの重要な機能になるかもしれないですし、あなたが次に作るアプリケーションがメールを多用するものになるかもしれません。練習を繰り返す時間が無駄になることは決してないのです。

これであなたは 私が 普段テストするときのノウハウを身につけました。必ずしも全部がエレガントな方法だとは限りませんが、結果としては十分なカバレッジを出せています。そして、そのおかげで私は気軽に機能を追加できています。既存の機能を壊してしまう心配はいりません。万一 、何かを壊してしまったとしても、私はテストを使ってその場所を特定し、内容に応じて問題を修復することができます。

RSpec と Rails に関する説明はそろそろ終わりに近づいてきたので、次は今までに得た知識を使い、もっと テスト駆動 らしくソフトウェアを開発する方法についてお話ししたいと思います。これが次章で説明する内容です。

演習問題

- もしあなたのアプリケーションにメール送信機能があるなら、練習としてその機能をテストしてください。候補としてよく挙がりそうなのはパスワードリセットのメッセージと通知かもしれません。もしそうした機能がないのであれば、本書のサンプルアプリケーションの中にある、Devise のパスワードリセット機能をテストしてみてください。
- あなたのアプリケーションにはファイルアップロード機能や、バックグラウンドジョブはありますか？繰り返しますが、練習用にこうした機能をテストするのは非常に良い考

[58] https://benoittgt.github.io/vcr/#/configuration/filter_sensitive_data

えです。テストをするときは本章で紹介したユーティリティも使ってください。こうした機能はよく忘れ去られます。そして早朝や深夜に止まって初めて思い出されるのです。

- あなたは外部の認証サービスや支払い処理、その他の web サービス用のスペックを書いたことがありますか？VCR を使ってスピードアップするにはどうすればよいですか？

11. テスト駆動開発に向けて

ふう。私たちはプロジェクト管理アプリケーションでたくさんのことをやってきました。本書の冒頭から欲しかった機能はすでにできあがっていましたが、テストは全くありませんでした。今ではアプリケーションは十分にテストされていますし、もし穴が残っていたとしても、私たちはその穴を調べて塞ぐだけのスキルを身につけています。

しかし、私たちがやってきたことはテスト駆動開発（TDD）でしょうか？

厳密に言えば「いいえ」です。アプリケーションコードは私たちが最初のスペックを追加する前から存在していました。私たちが今までやって来たことは探索的テストに近いものです。つまり、アプリケーションをより深く理解するためにテストを使っていました。本当に TDD を練習するにはアプローチを変える必要があります。すなわち、テストを先に書き、それからテストをパスさせるコードを書き、そしてコードをリファクタリングして今後もずっとコードを堅牢にしていくのです。その過程で、テストを使ってどういうコーディングをすべきか検討します。私たちはバグのないソフトウェアを作り上げるために努力しています。そのソフトウェアは将来新しい要件が発生して変更が入るかもしれません。

さあ、このサンプルアプリケーションで TDD を実践してみましょう！

この章で発生する変更点全体は GitHub 上のこの diff[59] 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-11-tdd origin/10-testing-the-rest
```

フィーチャを定義する

現時点ではプロジェクトを完了済みにする方法は用意されていません。この機能があれば、ユーザーは完了したプロジェクトを見えない場所に片付けて、現在のプロジェクトだけにフォーカスできるようになるので便利そうです。次のような二つの機能を追加して、この要件を実装してみましょう。

- プロジェクトを完了済みにするボタンを追加する。

[59] https://github.com/JunichiIto/everydayrails-rspec-jp-2024/compare/10-testing-the-rest...11-tdd

- ログイン直後に表示されるダッシュボード画面では完了済みのプロジェクトを表示しないようにする。

まず最初のシナリオから始めましょう。コーディングする前に、テストスイート全体を実行し、機能を追加する前に全部グリーンになることを確認してください。もしパスしないスペックがあれば、本書で身につけたスキルを活かしてスペックを修正してください。大事なことは開発を進める前にまず、きれいな状態から始められるようにしておくことです。

次に新しいシステムスペックに作業のアウトラインを記述します。プロジェクトを管理するためのシステムテストファイルはすでに作ってあるので、そこに新しいシナリオを追加できます。（状況によっては新しくファイルを作った方がいい場合もあります。）既存のシステムテストファイルを開き、新しいシナリオのスタブを追加してください。

spec/system/projects_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe "Projects", type: :system do
4    # ユーザーは新しいプロジェクトを作成する
5    scenario "user creates a new project" do
6      # 元のシナリオは省略 ...
7    end
8
9    # ユーザーはプロジェクトを完了済みにする
10   scenario "user completes a project"
11 end
```

ファイルを保存し、`bundle exec rspec spec/system/projects_spec.rb` というコマンドでスペックを実行してください。たぶん予想が付いていると思いますが、RSpec は次のようなフィードバックを返してきます。

```
Projects
  user creates a new project
  user completes a project (PENDING: Not yet implemented)

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

  1) Projects user completes a project
     # Not yet implemented
     # ./spec/system/projects_spec.rb:26



Finished in 0.7601 seconds (files took 0.35072 seconds to load)
2 examples, 0 failures, 1 pending
```

新しいシナリオにいくつかステップを追加しましょう。ここではユーザーがプロジェクトを完了済みにする流れを記述します。最初に、何が必要で、ユーザーは何をして、それがどんな結果になるのかを考えましょう。

1. プロジェクトを持ったユーザーが必要で、そのユーザーはログインしていないといけない。
2. ユーザーはプロジェクト画面を開き 完了（complete） ボタンをクリックする。
3. プロジェクトは 完了済み（completed） としてマークされる。

私はときどきテストに必要な情報をコメントとして書き始めます。こうすれば簡単にコメントをテストコードで置き換えられるからです。

spec/system/projects_spec.rb

```
1  # ユーザーはプロジェクトを完了済みにする
2  scenario "user completes a project" do
3    # プロジェクトを持ったユーザーを準備する
4    # ユーザーはログインしている
5    # ユーザーがプロジェクト画面を開き、
6    # "complete" ボタンをクリックすると、
7    # プロジェクトは完了済みとしてマークされる
8  end
```

ここまでに学んだテクニックを使って、コメントをテストコードに置き換えていきましょう。

spec/system/projects_spec.rb

```
# ユーザーはプロジェクトを完了済みにする
scenario "user completes a project", focus: true do
  user = FactoryBot.create(:user)
  project = FactoryBot.create(:project, owner: user)
  sign_in user

  visit project_path(project)
  click_button "Complete"

  expect(project.reload.completed?).to be true
  expect(page).to \
    have_content "Congratulations, this project is complete!"
  expect(page).to have_content "Completed"
  expect(page).to_not have_button "Complete"
end
```

ここで私は focus: true タグを追加しています。これは bundle exec rspec でこのスペックだけを実行するためです。この新機能に取り組んでいる間は他にもこのタグを付けるかもしれません。こうすれば毎回テストスイート全体を実行せずに済みます。みなさんも同じようにする場合は、変更点をコミットする前にタグを削除して、テストスイート全体を実行するのを忘れないようにしてください。タグを使ってテストの実行スピードを上げる方法については、第9章を参照してください。focus: true の省略形として、:focus を使うこともできます。

この新機能を実現するためのアプリケーションコードはまだ実際に書いていませんが、どのような形で動くのかはすでに記述しています。まず、*Complete* と書かれたボタンがあります。このボタンをクリックするとプロジェクトの completed 属性が更新され、値が false から true に変わります。それからフラッシュメッセージでプロジェクトが完了済みになったことをユーザーに通知します。さらにボタンがなくなるかわりに、プロジェクトが完了済みになったことを示すラベルが表示されることを確認します。これが テスト駆動開発 の基本です。テストから先に書き始めることで、コードがどのように振る舞うのかを積極的に考えることができます。

レッドからグリーンへ

スペックをもう一度実行してください。テストは失敗します！ですが、テスト駆動開発の場合、これは良いこととされているので覚えておきましょう。なぜなら、テストが失敗することで次に作業すべきゴールがわかるからです。RSpec は失敗した内容をわかりやすく表示してくれます。

```
1) Projects user completes a project
   Failure/Error: click_button "Complete"

   Capybara::ElementNotFound:
     Unable to find button "Complete"

     1) Projects user completes a project
        Failure/Error: click_button "Complete"

        Capybara::ElementNotFound:
          Unable to find button "Complete"
```

さて、この状況を前進させる一番シンプルな方法は何でしょうか？view に新しいボタンを追加してみるとどうなるでしょう？

app/views/projects/_project.html.erb

```
1  <h1 class="heading">
2    <%= project.name %>
3    <%= link_to edit_project_path(project),
4        class: "btn btn-light btn-sm btn-inline" do %>
5      <span class="bi bi-pencil-fill" aria-hidden="true"></span>
6      Edit
7    <% end %>
8    <button class="btn btn-light btn-sm btn-inline">
9      <span class="bi bi-check-lg" aria-hidden="true"></span>
10     Complete
11   </button>
12 </h1>
13 <!-- 残りの view は省略 ... -->
```

テストをもう一度実行し、前に進んだことを確認しましょう。

```
1) Projects user completes a project
   Failure/Error: expect(project.reload.completed?).to be true

   NoMethodError:
     undefined method `completed?' for an instance of Project
```

RSpec は次に何をすべきか、ヒントを与えてくれています。Project モデルに completed? という名前のメソッドを追加してください。アプリケーションの内容とビジネスロジックによりますが、ここではいくつかの検討事項が浮かび上がってきます。プロジェクト内のタスクがすべて完了したら、プロジェクトは 完了済み になるのでしょうか？もしそうであれば、completed? メソッドを Project モデルに定義し、プロジェクト内の未完了タスクが空（つまり、プロジェクトは完了済み）か、そうでないか（プロジェクトは未完了の意味）を返すことができます。

ただし、今回は完了済みかどうかは、ボタンをクリックするというユーザーの操作によって決定され、完了済みかどうかのステータスは永続化されることになります。なので、この値を保存するために、新しい Active Record の属性をモデルに追加する必要があるようです。projects テーブルにこのカラムを追加するマイグレーションを作成し、それを実行してください。

```
bin/rails g migration add_completed_to_projects completed:boolean
bin/rails db:migrate
```

Rails は自動的に新しいマイグレーションをテストデータベースに適用しようとしますが、毎回成功するとは限りません。テストデータベースに対してマイグレーションをまず実行してくださいというエラーメッセージが出た場合は、表示されているメッセージに従い（bin/rails db:migrate RAILS_ENV=test を実行します）、再度テストを実行してください。

bin/rails db:migrate の実行時に “SQLite3::SQLException: duplicate column name: completed” というエラーが発生した場合は、bin/rails db:reset コマンドでデータベースをリセットしてから再度 bin/rails db:migrate を実行してください。

変更を適用したら、新しいスペックをもう一度実行してください。今度は別の理由で失敗しますが、新しい失敗は私たちが前進していることを意味しています。

```
1) Projects user completes a project
   Failure/Error: expect(project.reload.completed?).to be true

     expected true
          got false
```

ある意味残念なことですが、この高レベルなテストは他のテストと同じように重要な情報を示しています。みなさんは何が起きているのかもうわかったかもしれませんが、第6章で説明した Launchy を使ってチェックしてみましょう。一時的に Launchy をテストに組み込んでください。

spec/system/projects_spec.rb

```
1  scenario "user completes a project", focus: true do
2    user = FactoryBot.create(:user)
3    project = FactoryBot.create(:project, owner: user)
4    sign_in user
5
6    visit project_path(project)
7    click_button "Complete"
8    save_and_open_page
9    expect(project.reload.completed?).to be true
10   expect(page).to \
11     have_content "Congratulations, this project is complete!"
12   expect(page).to have_content "Completed"
13   expect(page).to_not have_button "Complete"
14 end
```

興味深いことに、画面がプロジェクト画面から変わっていません。ボタンをクリックしても何も起きないようですね。ああそうだ、先ほど view に <button> タグを追加しましたが、このボタンが機能するようにしなければならないのでした。view を変更してちゃんと動くようにしていきましょう。

ですが、ここで新たに設計上の判断が必要になります。データベースに変更を加えるこのボタンのルーティングはどのようにするのが一番良いでしょうか？コードをシンプルに保ち、Project コントローラの update アクションを再利用することもできますが、ここではフラッシュメッセージが異なるため、まったく同じ振る舞いにはなりません。そこでコントローラに新しいメンバーアクションを追加することにしましょう。この新機能のテストがいったんパスすれば、時間が許す限り別の実装を試すことも可能です。

この場合は高いレベルから始めて、だんだん下へ降りていくアプローチが良いと思います。view に戻ってボタンを修正しましょう。<button> タグは Rails の button_to ヘルパーの呼び出しに置き換えてください。

app/views/projects/_project.html.erb

```
<h1 class="heading">
  <%= project.name %>
  <%= link_to edit_project_path(project),
      class: "btn btn-light btn-sm btn-inline" do %>
    <span class="bi bi-pencil-fill" aria-hidden="true"></span>
    Edit
  <% end %>
  <%= button_to complete_project_path(project),
      method: :patch,
      form: { style: "display:inline-block;" },
      class: "btn btn-light btn-sm btn-inline" do %>
    <span class="bi bi-check-lg" aria-hidden="true"></span>
    Complete
  <% end %>
</h1>
<!-- 残りの view は省略 ... -->
```

スタイルの記述を少し追加したことに加えて、この新しいアクションで呼ばれるルートヘルパーの定義も考えてみました。それが complete_project_path です。save_and_open_page をスペックから削除し、テストをもう一度実行してください。おっと、新しい失敗メッセージが出ました。

```
1) Projects user completes a project
   Failure/Error: <%= button_to complete_project_path(@project),

   ActionView::Template::Error:
     undefined method `complete_project_path' for
     an instance of #<Class:0x0000000124707b30>
```

このエラーは まさに これから使おうとしている新しいルーティングをまだ定義していないというヒントになっています。では、routes ファイルにルーティングを追加しましょう。

config/routes.rb

```
resources :projects do
  resources :notes
  resources :tasks do
    member do
      post :toggle
    end
  end
  member do
    patch :complete
  end
end
```

ルーティングを追加してからスペックを実行すると、別の新しい失敗メッセージが表示されます。

```
1) Projects user completes a project
   Failure/Error: click_button "Complete"

   AbstractController::ActionNotFound:
     The action 'complete' could not be found for ProjectsController
```

RSpec は私たちに何を修正すればいいのか、良いヒントを与えてくれています（訳注：エラーメッセージの最後に「ProjectsController に 'complete' アクションが見つかりません」と出力されています）。ここに書かれているとおり、Project コントローラに空の `complete` アクションを作成しましょう。追加する場所は Rails のジェネレータで作成された既存の `destroy` アクションの下にします。

app/controllers/projects_controller.rb

```
def destroy
  # Rails ジェネレータのコードは省略 ...
end

def complete
end
```

中身も書き始めたいという誘惑に駆られますが、テスト駆動開発の信条は「テストを前進させる必要最小限のコードを書く」です。先ほど、テストはアクションが見つからないと訴えていました。私たちはそれを追加しました。テストを実行して、現在の状況を確認してみましょう。

```
1) Projects user completes a project
   Failure/Error: unless @project.owner == current_user

   NoMethodError:
     undefined method `owner' for nil
   # ./app/controllers/application_controller.rb:13:in `project_owner?'
```

ちょっと面白い結果になりました。まったく別のコントローラでテストが失敗しています！これはなぜでしょうか？Project コントローラではアクションを実行する前にいくつかのコールバックを設定しています。ですが、その中に新しく作った complete アクションを含めていませんでした。これが失敗の原因です。というわけで、アクションを追加しましょう。

app/controllers/projects_controller.rb

```
class ProjectsController < ApplicationController
  before_action :set_project, only: %i[ show edit update destroy complete ]
  before_action :project_owner?, except: %i[ index new create ]

  # コントローラのコードは以下省略 ...
```

ではスペックを再実行してください。

```
Failures:

  1) Projects user completes a project
     Failure/Error: expect(project.reload.completed?).to be true
```

この結果は一見、何歩か後退してしまったように見えます。この失敗メッセージは少し前にも見たんじゃないでしょうか？ですが実際にはゴールに近づいています。上の失敗メッセージはコントローラのアクションには到達したものの、そのあとに何も起きなかったことを意味しています。というわけで、正常系のシナリオを満足させるコードを書いていきましょう。ここでは何の問題もなくユーザーがプロジェクトを完了済みにできることを前提とします。

app/controllers/projects_controller.rb

```
1 def complete
2   @project.update!(completed: true)
3   redirect_to @project,
4     notice: "Congratulations, this project is complete!"
5 end
```

スペックをもう一度実行してください。もうそろそろ終わりに近づいてきているはずです！

```
Failures:

  1) Projects user completes a project
     Failure/Error: expect(page).to have_content "Completed"
       expected to find text "Completed" in "Toggle navigation
       Project Manager Projects Aaron Sumner Sign Out
       × Congratulations, this project is complete!
       Project 1 Edit Complete A test project. Owner: Aaron Sumner
       Due: October 11, 2017 (7 days from now) Tasks Add Task
       Name Actions Notes Add Note Term"
```

ここに表示されている画面内の文言を読んでいくと、どうやら *Completed* という文字列が画面に出力されていないようです。これはまだこの文字列を追加していないからであり、さらに言うと私たちがやっているのはテスト駆動開発だからです。では view に文字列を追加してみましょう。先ほど追加したボタンの隣によく目立つラベルの ‹span› タグを追加してください。

app/views/projects/_project.html.erb

```
<h1 class="heading">
  <%= project.name %>
  <%= link_to edit_project_path(project),
      class: "btn btn-light btn-sm btn-inline" do %>
    <span class="bi bi-pencil-fill" aria-hidden="true"></span>
    Edit
  <% end %>
  <%= button_to complete_project_path(project),
      method: :patch,
      form: { style: "display:inline-block;" },
      class: "btn btn-light btn-sm btn-inline" do %>
    <span class="bi bi-check-lg" aria-hidden="true"></span>
    Complete
  <% end %>
  <span class="badge bg-success">Completed</span>
</h1>
<!-- 残りの view は省略 ... -->
```

テストをもう一度実行してください。失敗しているのは最後のステップだけです！

```
Failures:

  1) Projects user completes a project
     Failure/Error: expect(page).to_not have_button "Complete"
       expected not to find button "Complete", found 1 match: "Complete"
```

プロジェクトは完了済みになっていますが、*Complete* ボタンがまだ表示されたままです。このままだとユーザーを混乱させてしまうので、完了済みのプロジェクトを開いたときは UI にボタンを表示しないようにすべきです。view に新たな変更を加えれば、これを実現できます。

app/views/projects/_project.html.erb

```
<h1 class="heading">
  <%= project.name %>
  <%= link_to edit_project_path(project),
      class: "btn btn-light btn-sm btn-inline" do %>
    <span class="bi bi-pencil-fill" aria-hidden="true"></span>
    Edit
  <% end %>
  <% unless project.completed? %>
    <%= button_to complete_project_path(project),
        method: :patch,
        form: { style: "display:inline-block;" },
        class: "btn btn-light btn-sm btn-inline" do %>
      <span class="bi bi-check-lg" aria-hidden="true"></span>
      Complete
    <% end %>
  <% end %>
  <span class="badge bg-success">Completed</span>
</h1>
```

スペックを実行してください。・・・ついにパスしました！

```
Projects
  user completes a project

Finished in 0.58043 seconds (files took 0.35844 seconds to load)
1 example, 0 failures
```

私は view のロジックに変更が入ったらブラウザでどうなったのか確認するようにしています（API のエンドポイントを書いているときは curl かその他の HTTP クライアントでテストします）。Rails の開発用サーバーが動いていなければ、サーバーを起動してください。それからプロジェクト画面をブラウザで開いてください。*Complete* ボタンをクリックすると本当にプロジェクトが完了済みになり、ボタンが表示されなくなることを確認してください・・・と言いたいところですが、プロジェクトの完了状態にかかわらず、先ほど追加した *Completed* のラベルが表示されています！どうやらこの新機能が完成したと宣言する前に、テストをちょっと変更した方がいいようです。新しいアクションを実行する前に、*Completed* のラベルが画面に表示されて いない ことを確認しましょう。

spec/system/projects_spec.rb

```
# ユーザーはプロジェクトを完了済みにする
scenario "user completes a project" do
  user = FactoryBot.create(:user)
  project = FactoryBot.create(:project, owner: user)
  sign_in user

  visit project_path(project)

  expect(page).to_not have_content "Completed"

  click_button "Complete"

  expect(project.reload.completed?).to be true
  expect(page).to \
    have_content "Congratulations, this project is complete!"
  expect(page).to have_content "Completed"
  expect(page).to_not have_button "Complete"
end
```

こうするとスペックはまた失敗してしまいます。ですが、これは一時的なものです。

```
Failures:

  1) Projects user completes a project
     Failure/Error: expect(page).to_not have_content "Completed"
       expected not to find text "Completed" in "Toggle navigation
       Project Manager Projects Aaron Sumner Sign Out Project 1 Edit
       Complete Completed A test project. Owner: Aaron Sumner
       Due: October 11, 2017 (7 days from now) Tasks Add Task
       Name Actions Notes Add Note Term"
```

ボタンの周りに付けていた条件分岐を変更し、ラベルの表示も制御するようにしましょう。

app/views/projects/_project.html.erb

```
<h1 class="heading">
  <%= project.name %>
  <%= link_to edit_project_path(project),
      class: "btn btn-light btn-sm btn-inline" do %>
    <span class="bi bi-pencil-fill" aria-hidden="true"></span>
    Edit
  <% end %>
  <% if project.completed? %>
    <span class="badge bg-success">Completed</span>
  <% else %>
    <%= button_to complete_project_path(project),
        method: :patch,
        form: { style: "display:inline-block;" },
        class: "btn btn-light btn-sm btn-inline" do %>
      <span class="bi bi-check-lg" aria-hidden="true"></span>
      Complete
    <% end %>
  <% end %>
</h1>
```

そしてスペックをもう一度実行してください。

```
Projects
  user completes a project

Finished in 0.82131 seconds (files took 0.47196 seconds to load)
1 example, 0 failures
```

グリーンに戻りました！

ここまでは一つのスペックだけを実行してきましたが、別のことを始める前はいつもテストスイート全体をチェックするようにすべきです。`bundle exec rspec` を実行し、今回の変更が既存の機能を何も壊していないことを確認してください。TDD の最中に `focus: true` タグ、もしくは `:focus` タグを付けていた場合は、それも外すようにしましょう。

テストスイートはグリーンになりました。これなら大丈夫そうです！

外から中へ進む（Going outside-in）

私たちは高レベルの統合テストを使い、途中でソフトウェアがどんな振る舞いを持つべきか検討しながら、この新しい機能を完成させました。この過程の大半において、私たちは次にどんなコードを書くべきか、すぐにわかりました。なぜなら RSpec がその都度私たちにフィードバックを伝えてくれたからです。ですが、数回は何が起きていたのかじっくり調べる必要もありました。今回のケースでは、Launchy がデバッグツールとして大変役に立ちました。しかし、問題を理解するために、さらにテストを書かなければいけないこともよくあります。こうやって追加したテストはあなたの書いたコードをいろんなレベルから突っつき、高レベルのテストだけでは集めてくるのが難しい、有益な情報を提供してくれます。これがまさに、外から中へ（outside-in） のテストです。私はいつもこのような方法でテスト駆動開発を進めています。

私はブラウザのシミュレートの実行コストが成果に見合わないと思った場合に、低レベルのテストを活用することもよくやります。私たちが書いた統合テストは正常系の操作です。プロジェクトを完了済みにする際、ユーザーはエラーに遭遇することはありませんでした。では、更新に失敗する場合のテストも高レベルのテストとして新たに追加する必要があるでしょうか？いいえ、これは追加する必要はないかもしれません。なぜなら私たちはすでに *Complete* ボタンが正しく実装され、画面にフラッシュメッセージが正しく表示されていることを確認したからです。この次はコントローラのテストに降りていって、異常系の操作をいろいろとテストするのが良いかもしれません。たとえば、適切なフラッシュメッセージが設定され、プロジェクトオブジェクトが変わらないことを検証する、といったテストです。この場合、すでにコードは書いてあるので、コントローラのテストではプロジェクトを完了済みにする際に発生したエラーも適切に処理されることを確認するだけです。このテストは既存の Project コントローラのスペックに追加できます。

spec/controllers/projects_controller_spec.rb

```
require 'rails_helper'

RSpec.describe ProjectsController, type: :controller do
  # 他の describe ブロックは省略 ...

  describe "#complete" do
    # 認証済みのユーザーとして
    context "as an authenticated user" do
      let!(:project) { FactoryBot.create(:project, completed: nil) }

      before do
        sign_in project.owner
      end

      # 成功しないプロジェクトの完了
      describe "an unsuccessful completion" do
        before do
          allow_any_instance_of(Project).
            to receive(:update).
            with(completed: true).
            and_return(false)
        end

        # プロジェクト画面にリダイレクトすること
        it "redirects to the project page" do
          patch :complete, params: { id: project.id }
          expect(response).to redirect_to project_path(project)
        end

        # フラッシュを設定すること
        it "sets the flash" do
          patch :complete, params: { id: project.id }
          expect(flash[:alert]).to eq "Unable to complete project."
        end

        # プロジェクトを完了済みにしないこと
        it "doesn't mark the project as completed" do
```

```
38            expect {
39              patch :complete, params: { id: project.id }
40            }.to_not change(project, :completed)
41          end
42        end
43      end
44    end
45  end
```

上の example はいずれも失敗をシミュレートするのに十分なものです。ここでは allow_any_instance_of を使って、失敗を再現しました。allow_any_instance_of は第9章で使った allow メソッドの仲間です。このコードでは あらゆる（any） プロジェクトオブジェクトに対する update の呼び出しに割って入り、プロジェクトの完了状態を保存しないようにしています。allow_any_instance_of は問題を引き起こしやすいメソッドなので、describe "an unsuccessful completion" ブロックの中で だけ 使われるように注意しなければなりません。

このテストを実行し、何が起きるか見てみましょう。

```
Failures:

  1) ProjectsController#complete as an authenticated user an
  unsuccessful completion sets the flash
     Failure/Error: expect(flash[:alert]).to eq "Unable to complete
     project."

       expected: "Unable to complete project."
            got: nil

       (compared using ==)
     # ./spec/controllers/projects_controller_spec.rb:246:in
     `block (5 levels) in <top (required)>'
```

この結果を見ると、ユーザーは期待どおりにリダイレクトされ、プロジェクトも意図した通り、完了済みになっていないようです。しかし、問題の発生を知らせるメッセージが設定されていません。アプリケーションコードに戻り、これを処理する条件分岐を追加しましょう。

app/controllers/projects_controller.rb

```
def complete
  if @project.update(completed: true)
    redirect_to @project,
      notice: "Congratulations, this project is complete!"
  else
    redirect_to @project, alert: "Unable to complete project."
  end
end
```

このように変更すれば、新しいテストは全部パスします。これで正常系も異常系も、どちらもテストすることができました。

ここまでの内容をまとめると、私は「外から中へ」のテストをするときは、高レベルのテストから始めて、ソフトウェアが意図したとおりに動いていることを確認するようにします。このとき、すべての前提条件とユーザーの入力値は正しいものとします（つまり 正常系 ）。今回は統合テストとして、ブラウザのシミュレーションを行うシステムテストの形式を利用しました。ですが、API のテストをする場合はリクエストスペックを使います（第7章 を参照）。それから、低いレベルのテストに降りていき、可能な限り直接、細かい内容をテストします。今回はコントローラスペックを使いましたが、たいていの場合、モデルを直接テストしても大丈夫です。このテクニックはビジネスロジックを普通の Rails のモデルやコントローラから取り出して、サービスオブジェクトのような単体のクラスに移動させるかどうか検討する場合にも使えます。

筆者はテストを使って、サービスオブジェクトやそれに類似したパターンにリファクタリングする方法[60] を、Everyday Rails ブログで説明しています。ただし、本書の範疇を超えてしまうため、ここでは実施しません。

テストコードからフィードバックをもらい、その内容に従ってください。もし、今使っているテストから十分なフィードバックが得られない場合は、レベルをもう一段下げてみてください。

レッド・グリーン・リファクタのサイクル

新機能はひととおり完成しました。ですが、これで終わりではありません。ちゃんとパスするテストコードを使って、ここからさらに自分が書いたコードを改善することができま

[60] https://everydayrails.com/2017/11/20/replace-rspec-controller-tests.html

す。「レッド・グリーン・リファクタ」でいうところの「リファクタ」の段階に到達したわけです。新しいテストコードを使えば、他の実装方法を検討したり、先ほど書いたコードをきれいにまとめたりすることができます。

リファクタリングは非常に複雑で、詳しく説明し始めると本書の範疇を超えてしまいます。ですが、検討すべき選択肢はいくつか存在します。簡単なものから難しいものの順に並べてみましょう。

- 私たちは *projects/_project* の view に新しい条件分岐を追加しました。この分岐は *Complete* ボタンを表示するか、もしくはプロジェクトが完了済みであることを表示するかを決めるためのものです。私たちはこのコードをパーシャル view に抽出して、メイン view をシンプルに保つべきでしょうか？
- プロジェクトを完了させるルーティングを新たに実装する際、私たちは違う形でコントローラを構築する方法についても簡単に議論しました。今回選択した実装方法は本当にベストでしょうか？（この機能をテストするために何をやったのか見直してみると、私はちょっと自信がなくなります。コントローラのテストで Active Record のメソッドをスタブ化するのは、コントローラが多くのことをやりすぎているヒントかもしれません。）
- ビジネスロジックをコントローラから取り出して他の場所に移動させると、テストがよりシンプルになるかもしれません。ですが、どこがいいでしょうか？Project モデルに移動して `Project#mark_completed` のような新しいメソッドを作るとシンプルになりそうです。ですが、こういったアプローチを採用しすぎると、巨大なモデルができあがってしまう恐れがあります。別のアプローチとして、プロジェクトを完了済みにすることだけに責任を持つ、サービスオブジェクト にロジックを移動させる方法もあります。そうすれば、このサービスオブジェクトを直接テストするだけで済むので、コントローラを動かす必要がなくなります。
- もしくはまったく異なるコントローラの構成を選択することもできます。既存のコントローラにアクションを追加するかわりに、ネストしたリソースフル（resourceful）なコントローラを新たに作り、`update` アクションを定義して親のプロジェクトを完了させるのはどうでしょうか？

別の実装方法を検討する際は、テストを活用してください。さらに、コードを書いていたときに、テストコードが教えてくれたことも思い出してください。ほとんどの場合、コードを書く方法は一つだけとは限りません。いくつかの選択肢があり、それぞれに長所と短所があります。リファクタリングをするときは、小さくてインクリメンタルな変更を加えていってください。そして、テストスイートをグリーンに保ってください。これがリファクタリン

グの鍵となる手順です。どんな変更を加えても、テストは常にパスさせる必要があります。（もしくは、失敗したとしても、それは一時的なものであるべきです。）リファクタリングの最中は、高いレベルのテストと低いレベルのテストの間を行ったり来たりするかもしれません。システムスペックから始まり、モデルやコントローラ、もしくは独立した Ruby オブジェクトへと対象のレベルが下がっていくこともあります。対象となるテストのレベルは、アプリケーション内のどこにコードを置くともっとも都合が良いのか、そしてどのレベルのテストが最も適切なフィードバックを返してくれるのかによって、変わってくるものです。

まとめ

以上が RSpec を使って Rails アプリケーションに新しいフィーチャを開発するときの私のやり方です。紙面で見るとステップがかなり多いように見えるかもしれませんが、実際は短期的な視点で考えてもそれほど大した作業ではありません。そして長期的な視点で考えても、新機能を実装する際にテストを書いて早期にリファクタリングすれば、将来的な時間をかなり節約できます。さあこれであなたは自分のプロジェクトでも同じように活用できるツールを全部手に入れました！

演習問題

- 私たちはプロジェクトを完了済みにする新機能の最初の要件を一緒に実装しました。ですが、ユーザーのダッシュボードから完了済みのプロジェクトを非表示にする要件が残ったままになっています。これをテスト駆動開発で実装してみましょう。最初はこんな統合テストから書き始めてください。すなわち、完了済みのプロジェクトと未完了のプロジェクトを準備し、それからプロジェクトオーナーのダッシュボードを開いて適切なプロジェクトが表示されるか（または表示されないか）を検証してください。
- この機能が実装されると、ユーザーはどうやって完了済みのプロジェクトにアクセスするのでしょうか？どう動くべきかをまず検討し、それからその仕様を TDD で実装してください。
- だいたいできたと思ったら、「レッド・グリーン・リファクタ」の項で挙げたリファクタリングの選択肢から、一つ、もしくはそれ以上のアプローチを選択して実験してみてください。リファクタリングをすると、テストの性質がどう変わるでしょうか？とくに異常系のテストケースに注目してみてください。

12. 最後のアドバイス

よくやりました！もしあなたが本書で学んだパターンやテクニックを使って自分のアプリケーションにテストを追加してきたのであれば、あなたは十分にテストされた Rails アプリケーションを作り始めたことになります。あなたがここまで頑張ってきたことを私は嬉しく思います。そして今ではテストに慣れただけでなく、本物のテスト駆動開発者のように考えることができ、さらにスペックを使ってアプリケーションの内部設計も改善できるようになっていることを期待しています。そして、あなたはきっとこのプロセスを楽しいとすら考えているはずです！

本書を締めくくるために、あなたに覚えておいてほしいことをいくつか述べます。この内容を心に留めながら、引き続きこの道を進んでいってください。

小さなテストで練習してください

複雑な新しいフィーチャを題材にして TDD を始めようとするのは、TDD のプロセスを学ぶための最善な方法とは言えないかもしれません。それよりも取り組みやすそうなアプリケーションの課題に注目してください。バグ修正や基本的なインスタンスメソッドであれば、たいてい簡単なテストで済みます。セットアップはわずかで済みますし、必要なエクスペクテーションも一つで済むことが多いです。ただし、コードを書こうとする前にスペックを忘れずに書いてください！

自分がやっていることを意識してください

何か作業をするときは、自分が取り組んでいるプロセスを意識してください。そして、紙とペンを使いながら考えてください。今からやろうとしていることのためにスペックを書きましたか？スペックを使って境界値テストやエラー発生時のテストを書いていますか？チェックリストを作って、作業中はいつでも取り出せるようにしておいてください。そして、これからやらなければいけないことを事前に確認してください。

短いスパイクを書くのは OK です

テスト駆動開発は必ずしもテストがないとコードが書けない、というプロセスではありません。そうではなく、スペックを書かないと本番用のコードが書けない、というプロセスです。スパイク（訳注: アーキテクチャを確立するためのプロトタイプ）を作るのは全く問題ありません！フィーチャのスコープにもよりますが、私は新しいアイデアを試すときにはよく新しい Rails アプリケーションを作ったり、Git に一時的なブランチを作ったりします。特によくあるのは、新しいライブラリを試したり、比較的大きな変更を実施する場合です。

たとえば、私は一時期あるデータマイニングアプリケーションの開発をしていました。そのアプリケーションで私はユーザーインターフェイスに影響を与えることなく、モデルレイヤを完全に作り替えなければなりませんでした。私は基本的なモデルの構造がどうなるかはわかっていましたが、問題を完全に理解するためにもうちょっと良い方法を探る必要がありました。そのとき私は別のアプリケーションを用意して、この問題のスパイクを作ってみました。こうすることで、私はこれから解決しようとしている問題のために自由にハックしたり、実験したりすることができたのです。そして問題を理解し、良い解決策が見つかったところで本番用のアプリケーションに移り、実験から学んだことをベースにしてスペックを書きました。それからスペックをパスさせるコードを書きました。

もっと小さな問題であれば私はフィーチャブランチを作って作業します。そして同じような実験をして、変更されたファイルはどれなのか確認します。先ほどのデータマイニングプロジェクトに話を戻すと、私は集計したデータをユーザーに見せるフィーチャを作ったこともありました。そのデータはすでにシステムに存在していたので、私は Git にブランチを作り、シンプルな解決策を試すスパイクを作ってその問題を確実に理解しました。解決策が確定したら、私は自分の書いた一時的なコードを削除し、スペックを書きました。それから先ほどの作業内容を機械的に再適用しました。

私の場合、一般的なルールとして単にコメントを付けたり外したりする（もしくはコピー＆ペーストする）のではなく、もう一度コードを打ち直します。なぜなら、そうした方が最初に書いたコードをリファクタリングできたり、改良できたりすることが多いからです。

小さくコードを書き、小さくテストするのも OK です

もしあなたが最初にスペックから書き始めることをまだ難しいと感じているなら、コードを書いてからテストを書くのも良いと思います。ただし、その二つのプラクティスは必ずセットで行うというのが条件です。とはいえ、このやり方はテストを最初に書く場合に比べて

より多くの自制心が要求されることを強調しておきます。言いかえるなら、私は OK とは言ったものの、それが 理想的である とは考えていません。しかし、そうした方がテストしやすいのであれば、そうしても構いません。

統合スペックを最初に書こうとしてください

基本のプロセスが身について、さまざまなレベルでアプリケーションをテストできるようになったら、今度は全部を逆の順番にしてみましょう。つまり、モデルスペックを作り、それからコントローラスペック、システムスペック（またはリクエストスペック）、と進んでいくのではなく、まずシステムスペック（またはリクエストスペック）から スタートする のです。その際はエンドユーザーがアプリケーションを使ってタスクを完了させる手順を考えてください。これは 外から中へ（outside-in） のテストと呼ばれる一般的なアプローチです。このアプローチは第11章 でも実践しました。

システムスペックをパスさせるために開発していると、他のレベルでテストした方が良い機能が見つかります。たとえば、前章ではモデルレベルでバリデーションをテストし、コントローラレベルで認証機能をテストしました。よくできたシステムスペックはそのフィーチャが関連するテストのアウトラインを導き出してくれます。なので、システムスペックから書き始められるようになるのが、身につけるべきスキルなのです。

テストをする時間を作ってください

確かに、テストは今後の保守が必要になる余分なコードです。その余分なコードをメンテナンスするのにも時間がかかります。計画を立てるときはそれを見込んでください。あなたが1〜2時間で完成できると思ったフィーチャは、もう少し時間がかかるかもしれません。この問題は特にテストを書き始めたばかりの開発者によく発生します。しかし、長い目で見ればその時間は取り返せるはずです。なぜなら、信頼性の高いコードベースでいつでも作業を始められるからです。

常にシンプルにしてください

もしあなたがまだいくつかの分野のテスト（統合テストであれ、モックであれ、スタブであれ）に慣れていなくても、心配することはありません。そうした分野のテストは単に 動かす だけでは終わらずに、テストのためだけにより多くのセットアップや考えが必要になりま

す。しかし、シンプルな部分のテストをやめてはいけません。なぜならそうしたスキルも将来的にはもっと複雑なスペックを書くスキルにつながっていくからです。

古い習慣に戻らないでください！

失敗しないはずのテストが失敗してしまい、ずっと悩んでしまうというのはよくある話です。あなたがもしパスしないテストに出くわしたら、あとで見直すためにメモを取ってください。そして、しばらくしてからそのテストを見直してください。ブラウザを使って手作業で画面をクリックするようなテストは、アプリケーションが大きくなるほど時間がかかり、どんどん退屈な作業になっていくことを思い出しましょう。そんなことをするより、スペックをもっとうまく書くために時間を使って将来の時間を節約する方がよっぽど良いですよね？

テストを使ってコードを改善してください

レッド - グリーン - リファクタ の リファクタ ステージを無視してはいけません。テストの声に耳を傾けるスキルを身につけてください。あなたのコードがどこかおかしい場合、テストはあなたにそれを伝えます。そして重要な機能を壊さずにコードをきれいにするお手伝いをしてくれます。

自動テストのメリットを周りの人たちに売り込んでください

「テストを書く時間なんてあるわけがない」と今でも考えている開発者はたくさんいます。（中にはスパゲッティコードを理解している人間が世界で自分一人ならこの先も仕事は安泰だ、とまで考える開発者がいることも私は知っています。ですが、あなたがそんなバカではないことはわかっています。）もしくはあなたの上司が理解してくれないかもしれません。上司はなぜ次のフィーチャをリリースするのにわざわざ時間をかけてテストを書くのか理解できないかもしれません。

そんな同僚を教育する時間を少し用意してください。テストは開発のためだけにあるのではないと教えてあげてください。テストはアプリケーションを長期にわたって安定させるためのものであり、さらに人々の心の平和を安定させるものでもあります。テストが動作する様子を彼らに見せてあげてください。私の場合、第7章 で見たような JavaScript を実行するシステムスペックを披露したことがあります。このデモは彼らを驚かせ、スペックを書く時間

の価値を理解してもらうのに大変役立ちました。

練習し続けてください

最後に、言うまでもないことですが、たくさん練習しなければこのプロセスに対する技能を上達させることはできません。繰り返しになりますが、ちっぽけな Rails アプリケーションはこうした目的にうってつけです。新しいアプリ（ブログアプリと To-do リストがいつも人気です）を作り、フィーチャを実装しながら TDD を練習してください。そのフィーチャは何によって決まるのでしょうか？それはあなたが鍛えようとしているテストのスキルです。メールのスペックを上手に書けるようになりたいですか？それではブログアプリを改造して、購読者に記事の要約をメール送信してください。外部 API のテストスキルを伸ばしたいのであれば、多くのサービスが無料もしくは格安の API を開発者向けに用意してくれています。これは練習用にぴったりです。本番のプロジェクトで機能要望が上がってくるのを待っていてはいけません。自分で作るのです！

それではさようなら

さあこれであなたは Rails プロジェクトで基本的な自動テストを実践するために必要なツールをすべて手に入れました。RSpec、Factory Bot、それに Capybara。こうしたツールを使って自動テストを構築することができます。これらは私が日常的に Rails の開発で使っているコアツールです。そして、本書で紹介したテクニックは私がコードの信頼性を向上させるために学んできたやり方と同じものです。私の説明であなたもこうしたツールを同じように使えるようになったのであれば、私は嬉しいです。

これで「Everyday Rails - RSpec による Rails テスト入門」はおしまいですが、あなたがテスト駆動開発者を目指して頑張っていくなら、私にその成長ぶりを知らせてもらえると嬉しいです。また、本書の改善に役立つコメントや感想、気づき、提案、発見、不満、内容の訂正等があれば、お気軽にお知らせください。

- Email: aaron@everydayrails.com
- Twitter: https://twitter.com/everydayrails
- Facebook: https://facebook.com/everydayrails
- GitHub: https://github.com/everydayrails/everydayrails-rspec-2017/issues
- 日本語版に関するフィードバック: https://github.com/JunichiIto/everydayrails-rspec-jp-2024/issues

それから、Everyday Rails blog(https://everydayrails.com/) の新しい投稿もチェックしてもらえればと思います。

本書を読んでいただき、どうもありがとうございました。改めて感謝します。

Rails のテストに関するさらなる情報源

完全に網羅できているわけではありませんが、次に挙げる情報源はどれもあなたが Rails のテストについてより理解を深めるために役立つはずです。以下の情報源はすべて私も読んでいます。

RSpec

RSpec の公式ドキュメント

本書では Rails プロジェクトを RSpec でテストする方法にフォーカスしましたが、Rails 以外のプロジェクトでも RSpec を使うのであれば、まず公式ドキュメントを読んでみましょう。RDoc と Examples の2種類がありますが、Examples の方が具体的な使用例が多く、理解しやすいです。Examples ではバージョン3.12以降のドキュメントを読むことができます。https://rspec.info/documentation/

また、rspec-rails の公式ドキュメントもあります。こちらの Examples はバージョン6.0以降のドキュメントを読むことができます。https://rspec.info/features/6-0/rspec-rails/

Effective Testing with RSpec 3

Pragmatic Programmers から出版された *The RSpec Book* は時間が経って情報が古くなっていましたが、本書はその代わりとなる本です。新たに出版されたこの本は現行の RSpec メンテナである Myron Marston と Ian Dees によって書かれています（訳注：Myron Marston 氏は2018年に RSpec のメンテナから退きました）。もしみなさんが Rails を使わない普通の RSpec を使う必要があったり、純粋に RSpec というフレームワークへの理解を深めたいと思った場合は、本書が大変役立つはずです。

Better Specs

Better Specs には本当に素晴らしく、こだわりの強いベストプラクティスの説明がたくさん載っています。ぜひあなたのテストスイートにも適用してください。私は必ずしもすべてのプラクティスに賛成しているわけではありませんが、どれが賛成でどれが反対なのかはここでは言いません。何がベストなのかは、みなさんも自分自身の頭で考えてみてください。https://www.betterspecs.org

RSpec the Right Way

Pluralsight と Peepcode に在籍する Geoffrey Grosenbach が TDD のプロセスを動画で説明してくれます。使用しているツールは本書とほぼ同じです。視聴には Pluralsight のサブスクリプションが必要になります。https://www.pluralsight.com/courses/rspec-the-right-way

Railscasts

あなたが Rails 初心者なら、Ryan Bates による一級品のスクリーンキャストである *Railscasts* をあまり知らないかもしれません。本シリーズは私が Ruby on Rails を習得する際に大変大きな影響を与えました。現在は更新されなくなったことを大変残念に思います。Ryan はテストに関するエピソードもたくさん作っていました。中には古くなってしまった構文もありますが、考え方自体は今でも通用します。ぜひ"How I Test" というエピソードを視聴してみてください。私はこのエピソードを見て本書の着想を得ました。http://railscasts.com/?tag_id=7 / 日本語版: http://railscasts.com/episodes/275-how-i-test?language=ja&view=asciicast

RSpec の Google グループ

RSpec の Google グループはリリースのアナウンスや、使い方の説明、全般的なサポート等を行っているとても活発なグループです。RSpec に関して自力で答えが見つけられないときはここで質問するのが一番です。https://groups.google.com/group/rspec

Rails のテスト

Rails 5 Test Prescriptions: Build a Healthy Codebase

Noel Rappin によって書かれたこの本は、Rails のテストに関する本の中でも特に私が好きな一冊です。Noel は Rails のテストに関する幅広い分野を上手に説明してくれています。Test::Unit も、RSpec も、Cucumber も、クライアントサイド JavaScript のテストも説明されていますし、さらにはすべての技術要素を凝集度の高い堅牢なテストスイートにまとめるコンポーネントや考え方も教えてくれます。https://pragprog.com/titles/nrtest3/rails-5-test-prescriptions/

Noel はテストに関する講演も多数行っています。スライドは Speakerdeck で https://speakerdeck.com/noelrap チェックできます。

Ruby Tapas

Ruby Tapas は私のお気に入りの購読型動画サービスです。500を超えるエピソードが提供され、Ruby のあらゆる側面を説明してくれます。テスト関連のエピソードは素晴らしく、無

料で視聴できるものもいくつかあります（ですが、ぜひ 課金して購読することをお勧めします）。https://www.rubytapas.com/

Rails チュートリアル

Michael Hartl によって書かれたこの本は、私が Rails を学び始めた頃に読みたかった本です。第3版では RSpec が MiniTest に置き換えられましたが、だからといって読む価値がなくなったわけではありません。セットアップの手順や構文は多少異なるものの、日常的によく使う機能はどちらのテスティングフレームワークもそれほど違わないからです。本書の中で私が特に気に入っているのは、テストから書き始めるときと、コードから書き始めるとき[61]の実践的なガイドラインです。動画で学ぶのが好きな方にはスクリーンキャスト版もあります。https://ruby.railstutorial.org / 日本語版: https://railstutorial.jp

Agile Web Development with Rails 7

Sam Ruby と David Bryant Copeland によって執筆された *Agile Web Development with Rails* は、私が Rails を始めた頃から あった 本で、今でも 手に入れることができます。https://pragprog.com/titles/rails7/agile-web-development-with-rails-7/

（訳注: 本書の第4版は「Rails によるアジャイル Web アプリケーション開発」というタイトルで日本語版が発売されています。https://www.ohmsha.co.jp/book/9784274068669/ ）

Learn Ruby on Rails

Daniel Kehoe によって書かれた *Learn Ruby on Rails* は Rails プログラミングの初心者を対象にした書籍です。ただし、初心者向けとはいえ、MiniTest を使ったテストの章はなかなか素晴らしいです。MiniTest は Rails に最初から組みこまれているテストフレームワークで、RSpec に代わる選択肢の一つです。https://learn-rails.com

[61] https://www.railstutorial.org/book/static_pages#aside-when_to_test

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[62]、木原忠大さん[63]、森田高士さん[64]、遠藤大介さん[65]、安達輝雄さん[66]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[67]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[68]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[69]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[70]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。

[62] https://github.com/shirogani
[63] https://github.com/tkiha
[64] https://github.com/moritamori
[65] https://twitter.com/ruzia
[66] https://twitter.com/interu
[67] https://twitter.com/dentomo
[68] https://twitter.com/daipresents
[69] https://leanpub.com
[70] 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[71]、西川茂伸さん[72]、遠藤大介さん[73]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[74]のイギリス人プログラマ、マイケル（P. Michael Holland）[75]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

[71] https://twitter.com/joker1007
[72] https://twitter.com/shishi4tw
[73] https://twitter.com/ruzia
[74] http://nishiwaki-higashinadarb.doorkeeper.jp/
[75] 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[76]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[77]」のメンターでもある。ブログ[78]やQiita[79]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[80]」（技術評論社）がある。Twitter アカウントは@jnchito[81]。

[76] https://www.sonicgarden.jp
[77] https://bootcamp.fjord.jp/
[78] https://blog.jnito.com
[79] https://qiita.com/jnchito
[80] https://gihyo.jp/book/2021/978-4-297-12437-3
[81] 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[82]はiStockphoto の投稿者であるHabman_18[83]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。

[82] http://www.istockphoto.com/stock-photo-16071171-old-truck-in-early-morning-light.php?st=1e7555f
[83] 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）。