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
秋元 利春 訳
魚 振江

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 and Junichi Ito (伊藤淳一)

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

This version was published on 2015-09-14

Tweet This Book!

Please help Aaron Sumner and Junichi Ito (伊藤淳一) by spreading the word about this book on Twitter!

The suggested hashtag for this book is #edrr-jp.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#edrr-jp

Contents

日本語版のまえがき . . . . . **i**

この版のまえがき . . . . . **ii**

謝辞 . . . . . **iii**

追加コンテンツ「RSpec ユーザのための Minitst チュートリアル」について . . . . . **iv**

1. イントロダクション **1**
- なぜ RSpec なのか？ 1
- 対象となる読者 2
- 私が考えるテストの原則 2
- 本書の構成 3
- サンプルコードのダウンロード 4
- コードの方針 6
- 間違いを見つけた場合 6
- サンプルアプリケーションについて 6

2. RSpec のセットアップ **8**
- Gemfile 8
- テストデータベース 10
- RSpec の設定 11
- ジェネレータ 12
- データベーススキーマをテスト環境に適用する 13
- Q&A 13
- 演習問題 13

3. モデルスペック **15**
- モデルスペックの構造 15
- モデルスペックを作成する 16
- RSpec の新しい構文 18
- バリデーションをテストする 19
- インスタンスメソッドをテストする 23
- クラスメソッドとスコープをテストする 23
- 失敗をテストする 24
- マッチャについてもっと詳しく 25
- describe、context、before、after を使ってスペックを DRY にする 25
- まとめ 31
- Q&A 31
- 演習問題 31

4. ファクトリを使ったテストデータの生成 **33**
ファクトリ対フィクスチャ 33
アプリケーションにファクトリを追加する 34
構文をシンプルにする 37
関連とファクトリの継承 38
もっとリアルなダミーデータを作成する 40
高度な関連 41
ファクトリの弊害 43
まとめ 43
演習問題 43

5. コントローラスペックの基礎 **45**
なぜコントローラをテストするのか? 46
なぜコントローラをテストしないのか? 46
コントローラテストの基礎 46
整理 47
テストデータをセットアップする 49
GET リクエストをテストする 50
POST リクエストをテストする 52
PATCH リクエストをテストする 55
DELETE リクエストをテストする 56
非 CRUD なメソッドをテストする 57
入れ子になったルーティングをテストする 58
コントローラの非 HTML 出力をテストする 58
まとめ 60
演習問題 60

6. 高度なコントローラスペック **61**
準備する 61
管理者ロールとユーザーロールをテストする 61
ゲストロールをテストする 64
まとめ 66
演習問題 67

7. コントローラスペックのクリーンアップ **68**
Shared examples 68
ヘルパーマクロを作成する 74
カスタム RSpec マッチャを使う 75
まとめ 76
演習問題 76

8. フィーチャスペック **78**
なぜフィーチャスペックなのか? 78
Cucumber はどうなのか? 79
必要な追加ライブラリ 79
フィーチャスペックの基礎 79
リクエストからフィーチャへ 81
フィーチャスペックを追加する 82
フィーチャスペックをデバッグする 82
ちょっとしたリファクタリング 83

JavaScript を利用するやりとりも含める 84
Capybara のドライバ 87
JavaScript の完了を待つ 88
まとめ 88
演習問題 88

9. スペックの高速化 89
構文を簡潔にするためのオプション 89
モックとスタブ 93
Guard と Spring を使ったテストの自動実行 94
タグ 96
高速化させる他の解決策 97
まとめ 98
演習問題 98

10. その他のテスト 99
メール送信をテストする 99
ファイルアップロードをテストする 102
時間をテストする 103
Web サービスをテストする 104
自分のアプリケーションが提供する API をテストする 106
rake タスクをテストする 108
まとめ 110
演習問題 110

11. テスト駆動開発に向けて 111
フィーチャを定義する 111
レッドからグリーンへ 113
クリーンアップ 120
まとめ 121
演習問題 121

12. 最後のアドバイス 122
小さなテストで練習してください 122
自分がやっていることを意識してください 122
短いスパイクを書くのは OK です 122
小さくコードを書き、小さくテストするのも OK です 123
できるだけフィーチャスペックから書き始めてください 123
テストをする時間を作ってください 123
常にシンプルにしてください 123
古い習慣に戻らないでください! 124
テストを使ってコードを改善してください 124
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 124
練習し続けてください 124
それではさようなら 124

Rails のテストに関するさらなる情報源 126
RSpec 126
Rails のテスト 127

訳者あとがき **128**
伊藤淳一 128
秋元利春 128
魚振江 129

日本語版の謝辞 **130**

Everyday Rails について **131**

著者について **132**

訳者紹介 **133**
伊藤淳一 133
秋元利春 133
魚振江 133

カバーの説明 **134**

変更履歴 **135**
2015/09/15 135
2015/06/30 135
2014/12/29 135
2014/11/23 135
2014/10/24 135
2014/07/17 135
2014/05/22 135
2014/04/22 136
2014/04/17 136
2014/02/28 136
2014/02/10 136
2014/02/07 136

日本語版のまえがき

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

この版のまえがき

RSpec 3 版の Everyday Rails - RSpec による Rails テスト入門 がついに完成しました!RSpec 3 版では多くの変更点を変更しました。みなさんには待った甲斐があったと思ってもらえると嬉しいです。

以前のアップデートと同様に、サンプルアプリケーションは再作成しています。再作成の際には Rails と RSpec、それに本書で使用したその他の gem の最新バージョンを使用しました。さらに、いくつかのセクションを大きく加筆しています。一番大きく加筆したのは第 10 章の その他のテスト です。また、必要に応じて RSpec 3 と Rails 4.1 の新機能を活かした加筆修正も行っています。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[1] にぜひ報告してください。すぐに対処します。(訳注: 日本語版のフィードバックはこちらからお願いします。https://leanpub.com/everydayrailsrspec-jp/feedback)

改めてみなさんに感謝します。みなさんにこの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。

[1] https://github.com/everydayrails/rails-4-1-rspec-3-0/issues

謝辞

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[2]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[3]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。

[2] http://railscasts.com/
[3] http://www.designojek.com/

追加コンテンツ「RSpec ユーザのための Minitst チュートリアル」について

本書「Everyday Rails - RSpec による Rails テスト入門」には日本語版独自の特典として「RSpec ユーザのための Minitest チュートリアル」という追加コンテンツ（電子書籍）が付いています。これは本書で使われている RSpec のテストコードを Minitest で書き換える方法を説明した電子書籍です。フォーマットは Everyday Rails と同様、EPUB、MOBI、PDF の 3 種類です。

この追加コンテンツは以下の手順でダウンロードできます。

1. ログインページ[4]から Leanpub にログインします。
2. Purchases ページ[5]に移動します。
3. 購入済み書籍の中から「Everyday Rails - RSpec による Rails テスト入門」を選択します。
4. DOWNLOAD EXTRAS のリンクから zip ファイルをダウンロードします。
5. ダウンロードした zip ファイルに「RSpec ユーザのための Minitest チュートリアル」の電子書籍ファイルが含まれています。

何か不明な点があればLeanpub のフィードバックページ[6]でお気軽にご質問ください。

[4] https://leanpub.com/login
[5] https://leanpub.com/dashboard/purchases
[6] https://leanpub.com/everydayrailsrspec-jp/feedback

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が、もしそれが好みでなければ自分が好きなものに置き換えることができます。私がこれを書いている時点で Ruby Toolbox だけでも *Unit Test Frameworks* カテゴリに 17 のプロジェクト[7]が挙がってきます。つまり、テストは Rails で大変重要視されています。とはいえ、Rails でテストを全く書かずに開発する人や、書いたとしてもモデルのバリデーション用にたった数個のスペックしか書かない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 web フレームワークは新しい技術であり、そこへさらに新しい技術が増えるのは彼らにとって余計な仕事になるのかもしれません。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ていませんが、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ことがその解決策になっています。私は 1995 年から web アプリケーションを開発しています。ただし、予算の乏しい公共セクターのプロジェクトを一人で担当することがほとんどです。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[8]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 上の情報源ではそういう傾向が間違いであったことを認めていますし、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道を歩んでいる開発者）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。

本書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。

なぜ RSpec なのか？

私は基本的に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 ライブラリを書く時は MiniTest をよく使っています。しかし、Rails アプリケーションをテストするときは、どういうわけか RSpec を使い続けています。

---

[7] https://www.ruby-toolbox.com/categories/testing_frameworks
[8] http://en.wikipedia.org/wiki/Spaghetti_code

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。

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（オンライン[9]で読めます）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。ただし、RSpecといくつかの追加ライブラリは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。

もしあなたのRails開発経験があまり長くなく、本番運用しているアプリケーションが一つか二つぐらいで、なおかつテストにはまだ馴染めていないのなら、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Model-View-Controllerアーキテクチャ
- gemの依存関係を管理するBundler
- Rakeタスクの実行方法
- 基本的なコマンドラインの使い方
- 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

一方、上級者に関していうと、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、十分なカバレッジが出せるような満足のいく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。ですが、正直に言って、現時点であなたは自動テストの重要性を十分理解できているはずですし、本書から得られる知識はほとんど必要ないものかもしれません。本書はテスト理論に関する本ではありません。また、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長い目で見たときに役立つかもしれません。

本書の最後にある Railsのテストに関するさらなる情報源 にはここに挙げた書籍と、それ以外の書籍やwebサイト、テスト用のチュートリアルを載せているので参考にしてください。

私が考えるテストの原則

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、VimとEmacsの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

[9] http://railstutorial.jp/?version=4.0

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、Rails のテストについて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。

Ruby のテスト駆動/ビヘイビア駆動開発コミュニティからはさらに異論が巻き起こるかもしれませんが、私のアプローチでは次のような原則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
- テストは簡単に理解できること

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。さらに、昨今のテスト駆動開発の意味がなんであるにせよ、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。

もちろん、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を書くことです。最適化されているとは言いがたいテストであっても、信頼性が高く、理解しやすいテストを書くことが大事な出発点になります。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、テストを書くようになって、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 4.1 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 3.1 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、いくつかの便利なテスト用ツールを使って、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 ファクトリを使ったテストデータの生成 ではテストデータの生成を簡単にしてくれるファクトリについて説明します。
- 第 5 章 コントローラスペックの基礎 ではコントローラのテストについて最初の説明を行います。
- 第 6 章 高度なコントローラスペック では認証と認可のレイヤが正しく機能していること、すなわちアプリケーションのデータがちゃんと守られていることをコントローラスペックで確認します。
- 第 7 章 コントローラスペックのクリーンアップ ではスペックのリファクタリングについて最初の説明を行います。このリファクタリングでは可読性を保ったまま冗長なコードを減らしていきます。
- 第 8 章 フィーチャスペック ではフィーチャスペックを使った統合テストを見ていきます。フィーチャスペックを使うと、アプリケーションの様々なパーツが他のパーツと協調して動作することをテストできます。

- 第 9 章 スペックの高速化 ではリファクタリングのテクニックとパフォーマンスを念頭に置いたテストの実行方法について説明します。
- 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、時間に依存する機能、API 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、多くの章は 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

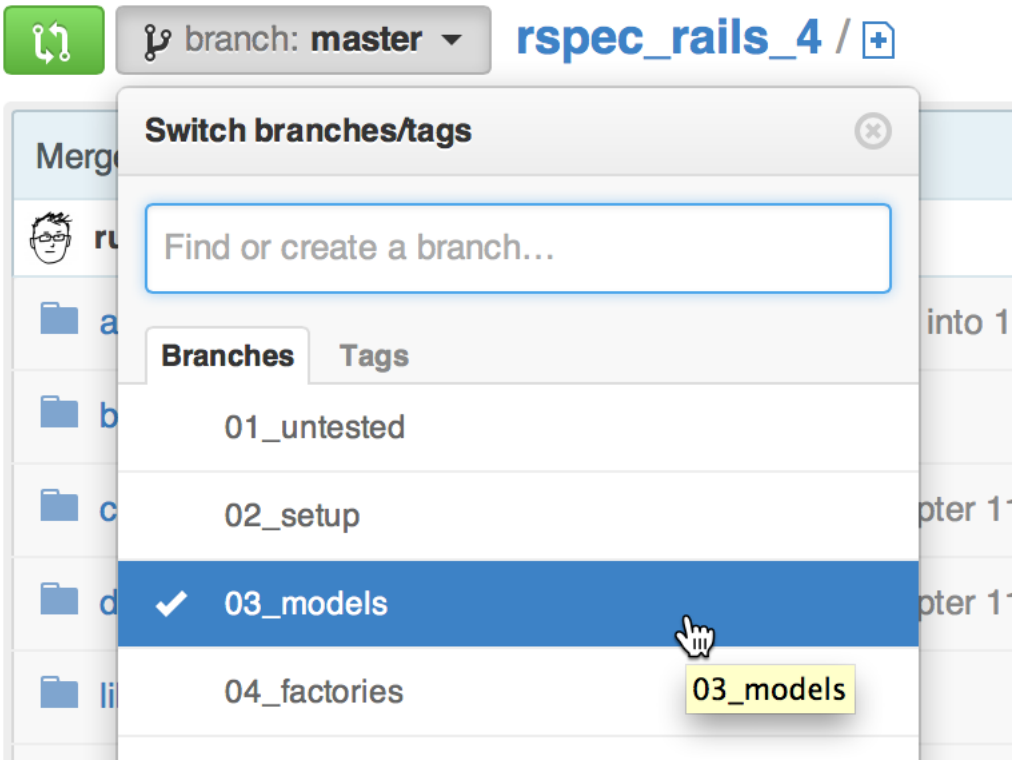
本書のサンプルコードは GitHub にあります。このアプリケーションではテストコードも完全に書かれています。

ソースを入手しましょう！

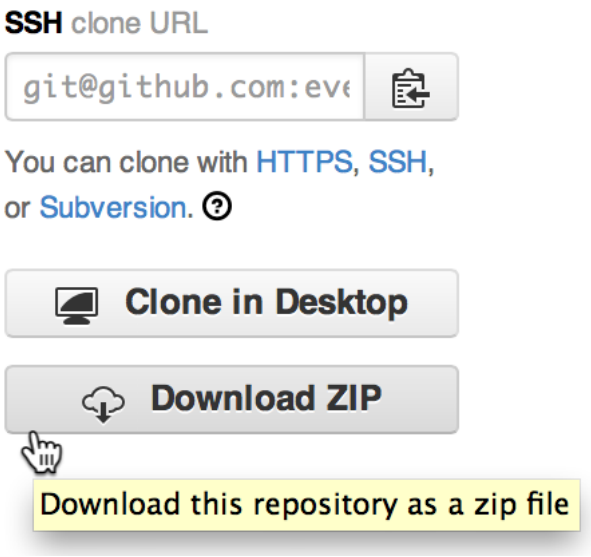
https://github.com/everydayrails/rails-4-1-rspec-3-0

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が、各章の最初でもチェックアウトすべきブランチを伝えていきます。

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 ダウンロードボタンをクリックします。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[10]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[11]もそのようなサイトです。基礎の復習にはGit Reference[12]が便利です。

[10] http://gitimmersion.com/
[11] http://try.github.io
[12] http://gitref.org

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 4.1**: 最新バージョンの Rails が本書のメインターゲットです。とはいえ Rails 3.0 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 2.1**: 1.9 や 2.0 を使っていても大きな差異はないと思います。しかし、今でも 1.8 を使っているのであれば、本書を読み進めるのが難しくなると思うのでお薦めできません。
- **RSpec 3.1**: RSpec 3.0 は 2014 年の春にリリースされました。その数ヶ月後に RSpec 3.1 が登場しましたが、大半は 3.0 と互換性が保たれています。RSpec 3.1 は RSpec 2.14 に近いシンタックスになっていますが、多少異なる点もあります。

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。もし上記のツール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。そうではなく、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[13]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 Web サイト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

完璧な人間はいません。私は特に完璧ではありません。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/everydayrails/rails-4-1-rspec-3-0/issues（訳注: 日本語版に関するフィードバックは https://leanpub.com/everydayrailsrspec-jp/feedback から投稿してください。）

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本当に質素で、本当にちっぽけな連絡先管理ツールです。このアプリケーションは企業 Web サイトの一部なのかもしれません。提供する機能は名前とメールアドレスと電話番号を一覧表示することと、単純な頭文字検索です。これらの機能はサイトに訪問すれば誰でも使えます。ログインが必要になるのは新しい連絡先を追加したり、既存の連絡先を変更したりする場合です。最後に、ユーザーは管理者権限を持っていないと新しいユーザーをシステムに追加することができません。

さて、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* 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

[13] http://learncodethehardway.org/

`rake 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。もっと良いソリューションをこれから作っていくので 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、もっと立派な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。Rails のジェネレータでコードを追加するときは、毎回適切なスペック（それにいくつかの便利なファイル）を作成するようにアプリケーションを設定します。

では始めましょう！

2. RSpec のセットアップ

第 1 章で述べたように、連絡先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、私たちに証明できることといえば、リンクをクリックし、ダミーのアカウントをいくつか作り、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ことを繰り返しているといつか破綻します。(私はこれよりもさらにテストできていない状態でアプリケーションをデプロイしたことがあります。読者のみなさんの中にもきっといるはずです。)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。RSpec といくつかのヘルパーgem を使ってそれを実現させましょう。

とはいえ、スペックを書き始める前に私たちはいくつか設定を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は工夫が必要でしたが、今はもう違います。しかし、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残っています。

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 やテストの時に便利なその他の gem 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。

本章の完成後のコードを見たい場合はサンプルソースの *02_setup* ブランチをチェックアウトしてください。コマンドラインを使う場合は次のようにタイプします。

```
git checkout -b 02_setup origin/02_setup
```

本章を読みながら一緒にコードも書いていく場合は、前の章のブランチから始めてください。

```
git checkout -b 01_untested origin/01_untested
```

その他、詳細については第 1 章を読んでください。

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ません。なので一番最初にやることは RSpec とテストを書きやすくするいくつかの gem をインストールすることです。これらの gem をインストールするために Bundler を使います。`Gemfile` を開き、次のようなコードを追加してください。

Gemfile

```
group :development, :test do
  gem "rspec-rails", "~> 3.1.0"
  gem "factory_girl_rails", "~> 4.4.1"
end

group :test do
  gem "faker", "~> 1.4.3"
  gem "capybara", "~> 2.4.3"
  gem "database_cleaner", "~> 1.3.0"
  gem "launchy", "~> 2.4.2"
  gem "selenium-webdriver", "~> 2.43.0"
end
```

これらの gem のバージョンは RSpec 3.1/Rails 4.1 版の執筆時点（2014 年夏）での最新のバージョンです。もちろん、どれも頻繁にアップデートされます。ですので、Rubygems.org や GitHub、お気に入りの Ruby ニュースフィードなどで常に最新の情報をチェックしておきましょう。

Bundler について知っておいてください

もし前述のコードがよくわからなければ、本書を読むのは少しやめて Bundler のチュートリアルを見つけてください。(読み終わったら戻ってきてください!) Rails のチュートリアルでバージョン 3.0 以上を扱っているものであれば、Bundler に関する記述があるはずです。また、Bundler's getting started guide[14]を読むのもお薦めです。最近の他の分野の Rails 開発と同じく、本書でも Bundler をたくさん使っていきます。

なぜ 2 つのグループに分けてインストールするのか？

rspec-rails と *factory_girl_rails* は development と test の両方の環境で使用されます。具体的にいうと、development 環境ではこのすぐあとに紹介するジェネレータが使います。残りの gem はスペックを実行するときだけ使用します。ですから development 環境で読み込む必要はありません。

コマンドラインから `bundle install` を実行して、システムに gem をインストールしてください。

では何をインストールしたんでしょうか？

- *rspec-rails* は RSpec を含んでいる gem です。この gem は Rails 専用の機能を追加する RSpec のラッパーライブラリになっています。
- *factory_girl_rails* は Rails がデフォルトで提供するフィクスチャをずっと便利な ファクトリ で置き換えます。フィクスチャやファクトリはテストスイート用のテストデータを作成するために使われます。
- *faker* は名前やメールアドレス、その他のプレースホルダを ファクトリ に提供します。
- *capybara* はユーザと Web アプリケーションのやりとりをプログラム上で簡単にシミュレートできるようにします。
- *database_cleaner* はまっさらな状態で各 spec が実行できるように、テストデータベースのデータを掃除します。
- *launchy* はある一つの仕事をします。その仕事とはあなたの好きなタイミングでデフォルトの web ブラウザを開き、アプリケーションの表示内容を見せることです。テストをデバッグするときには大変便利です。

[14] http://bundler.io

- *selenium-webdriver* はブラウザ上で JavaScript を利用する機能を Capybara でテストできるようにします。

それぞれの詳細はこの先の章で説明しますが、堅牢なテストスイートを作るために必要な基本ライブラリはいつでも使える状態になっています。では次にテストデータベースの作成に進みましょう。

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

```
test:
  adapter: sqlite3
  database: db/test.sqlite3
  pool: 5
  timeout: 5000
```

MySQL を使っているのであればこうなります。

config/database.yml

```
test:
  adapter: mysql2
  encoding: utf8
  reconnect: false
  database: contacts_test
  pool: 5
  username: root
  password:
  socket: /tmp/mysql.sock
```

PostgreSQL を使っているのであればこうなります。

config/database.yml

```
test:
  adapter: postgresql
  encoding: utf8
  database: contacts_test
  pool: 5
  username: root # もしくは適切なユーザー名
  password:
```

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。*contacts_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。

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/rake db:create:all
```

もし Rails 4.1 よりも古いバージョンを使っている場合は、上記のコマンドを `bundle exec rake db:create:all` に置き換えてください。基本的に本書のコマンドラインで `bin/` になっている箇所はすべて `bundle exec` で置き換える必要があります。

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、rake 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。

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てくれます。

```
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` ）、それに 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薦めします。

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。*.rspec* ファイルを開き、次の行を追加してください。

.rspec

```
--format documentation
```

もし RSpec 3.1 ではなく、3.0 を使っている場合は、このファイルにある `--warnings` の行を削除した方が良いかもしれません。警告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薦めします。この設定はあとからいつでも戻せます。

ジェネレータ

これは任意ですが、もうひとつセットアップの作業があります。標準のジェネレータを使ってモデルやコントローラ、scaffold を追加する際に、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。

Railties[15]のおかげで、`rspec-rails` と `factory_girl_rails` を読み込むだけですべての準備が完了します。Rails のジェネレータを使ってもデフォルトだった Test::Unit 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（そしてファクトリが `spec/factories` 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。`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、ビュースペックのような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 `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

```
config.generators do |g|
  g.test_framework :rspec,
    fixtures: true,
    view_specs: false,
    helper_specs: false,
    routing_specs: false,
    controller_specs: true,
    request_specs: false
  g.fixture_replacement :factory_girl, dir: "spec/factories"
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: true` はモデルごとにフィクスチャを作成することを指定します（実際はフィクスチャの代わりに Factory Girl のファクトリを使います）。
- `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 フィーチャスペック でインターフェースをテストします。
- `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。
- `request_specs: false` を指定すると RSpec がデフォルトで追加する統合テストレベルのスペックを *spec/requests* に追加しなくなります。この内容は第 8 章で扱います。そこでは必要に応じて自分でファイルを作成しています。
- そして最後に、`g.fixture_replacement :factory_girl` はフィクスチャの代わりにファクトリを作成し、`spec/factories` ディレクトリにファクトリを保存することを指定しています。

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

[15] http://api.rubyonrails.org/classes/Rails/Railtie.html

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。命名規則はこのようになります。もし、*app/helpers/contacts_helper.rb* をテストするのであれば、*spec/helpers/contacts_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。

最後に、RSpec 用の binstub をインストールしましょう。

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

このコマンドを実行すると、*bin* ディレクトリ内に *rspec* の実行可能ファイルが作成されます。

データベーススキーマをテスト環境に適用する

4.1 よりも古いバージョンの Rails では、`rake db:test:prepare` や `rake db:test:clone` を使って開発環境のデータベース環境をテスト用データベースに手作業で反映させなければなりませんでした。しかし、Rails 4.1 ではマイグレーションを実行するたびに自動的にテスト用のデータベースにも変更点が反映されるようになっています。

というわけで、このサンプルアプリケーションは RSpec を使ってテストする準備が整いました！最初のテスト実行を行うこともできます。

```
$ bin/rspec
No examples found.

Finished in 0.00027 seconds (files took 0.06527 seconds to load)
0 examples, 0 failures
```

大丈夫ですね！次の章ではアプリケーションの機能を実際にテストしていきます。まずはモデルレイヤから始めましょう。

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 `bin/rake test` 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 の spec フォルダ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。これは本当です。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 とその他の必要な gem を `Gemfile` に追加し、Bundler を使ってインストールしてください。基本的に本書で使ったコードとテクニックは Rails 3.0 以上で使えます。
- 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 と Factory Girl を使うように Rails のジェネレータコマンドを設定してください。各 gem 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。
- 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 と関連する gem 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はずです。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/rake 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 と Factory Girl 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したときに、スペックとファクトリ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

いやいや、これをやっても実際にポイントを差し上げるわけではありません。ですが、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[16]から始めてみるのが良いと思います。

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

3. モデルスペック

堅牢で信頼性の高いテストスイートを構築するために必要なツールはすべて揃いました。今度はそのツールを使う時です。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。ここでは *Contact* モデルがその対象になります。
- 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。アプリケーションに新しいモデルを追加する場合は（大丈夫、第 11 章でもやります）、第 2 章で設定した便利な RSpec ジェネレータが私たちに代わって仮のファイルを作ってくれます。

本章の完成後のコードを見たい場合はサンプルソースの *03_models* ブランチをチェックアウトしてください。コマンドラインを使う場合は次のようにタイプします。

```
git checkout -b 03_models origin/03_models
```

本章を読みながら一緒にコードも書いていく場合は、前の章のブランチから始めてください。

```
git checkout -b 02_setup origin/02_setup
```

その他、詳細については第 1 章を読んでください。

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ることが必要不可欠です。堅牢な土台作りはコードレベル全体の信頼性を高めるための第一歩です。

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。

- 有効な属性が渡された場合、モデルの `create` メソッドが正常に完了すること
- バリデーションを失敗させるデータであれば、正常に完了しないこと
- 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

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 *Contact* モデルの要件を見てみましょう。

```
describe Contact do
  # 姓と名とメールがあれば有効な状態であること
  it "is valid with a firstname, lastname and email"
  # 名がなければ無効な状態であること
  it "is invalid without a firstname"
  # 姓がなければ無効な状態であること
  it "is invalid without a lastname"
  # 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"
  # 連絡先のフルネームを文字列として返すこと
  it "returns a contact'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 `Contact` 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 `firstname`、`lastname`、`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。*Contact is invalid without a firstname*(名がなければ連絡先は無効な状態である)、*Contact is invalid without a lastname*(姓がなければ連絡先は無効な状態である)、*Contact returns a contact's full name as a string*(連絡先は文字列として連絡先のフルネームを返す)。可読性は重要です!

こうしたベストプラクティスを念頭に置きながら *Contact* 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

まず、`spec` ディレクトリを開きます。必要であれば `models` という名前のサブディレクトリを作成します。そのサブディレクトリの中に `contact_spec.rb` という名前のファイルを作り、次のようなコードを追加しましょう。

spec/models/contact_spec.rb

```
1  require 'rails_helper'
2
3  describe Contact do
4    # 姓と名とメールがあれば有効な状態であること
5    it "is valid with a firstname, lastname and email"
6    # 名がなければ無効な状態であること
7    it "is invalid without a firstname"
8    # 姓がなければ無効な状態であること
9    it "is invalid without a lastname"
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   # 連絡先のフルネームを文字列として返すこと
15   it "returns a contact's full name as a string"
16 end
```

最初の行にある `require 'rails_helper'` に気をつけてください。そしてこれをタイプするのに慣れてください。今後すべてのスペックにこの行が含まれることになります。これは RSpec 3 で新しく登場した内容です。以前のバージョンでは `spec_helper` と書いていました。Rails 固有の設定はこのファイルに移動しています。その結果、spec_helper はかなり小さくなりました。この内容は第 9 章でもう少し詳しく説明します。

場所、場所、場所

スペックファイルの名前と場所は重要です！RSpec のファイル構造は `app` ディレクトリの構造と同じになります。中に含まれるファイルについても同様です。モデルスペックであれば、`contact_spec.rb` は `contact.rb` に対応します。本書の後半ではアプリケーション側のコードを変更したときに対応するスペックを自動実行させるようにするので、この対応関係はより重要になってきます。

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 `bin/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。

```
Contact
  is valid with a firstname, lastname and email
    (PENDING: Not yet implemented)
  is invalid without a firstname
    (PENDING: Not yet implemented)
  is invalid without a lastname
    (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 contact's full name as a string
    (PENDING: Not yet implemented)

Pending:
```

```
  Contact is valid with a firstname, lastname
    and email
    # Not yet implemented
    # ./spec/models/contact_spec.rb:4
  Contact is invalid without a firstname
    # Not yet implemented
    # ./spec/models/contact_spec.rb:5
  Contact is invalid without a lastname
    # Not yet implemented
    # ./spec/models/contact_spec.rb:6
  Contact is invalid without an email address
    # Not yet implemented
    # ./spec/models/contact_spec.rb:7
  Contact is invalid with a duplicate email address
    # Not yet implemented
    # ./spec/models/contact_spec.rb:8
  Contact returns a contact's full name as a string
    # Not yet implemented
    # ./spec/models/contact_spec.rb:9

Finished in 0.00105 seconds (files took 2.42 seconds to load)
6 examples, 0 failures, 6 pending
```

すばらしい！6 つの保留中の example ができました。まず最初の example から書き始めて、全部のスペックをパスさせましょう。

Rails の `model` ジェネレータ、もしくは `scaffold` ジェネレータを使って連絡先管理アプリケーションに新しいモデルを追加すると、モデルのスペックファイルが自動的に作られます。作られなかった場合はジェネレータの設定を見直してください。もしくは第 2 章で説明したとおりに *rspec-rails* gem が正しくインストールされていることを確認してください。インストールだけでなく、ジェネレータの設定も必要になりますよ。

RSpec の新しい構文

2012 年 6 月、RSpec の開発チームはそれまで使われてきた `should` に代わる新しい構文をバージョン 2.11 に追加しました。そうです、これは本書の完成版を最初にリリースしたわずか数日後の出来事でした。こうした変更に追い付いていくのは大変です！

この新しいアプローチを使うと古い `should` 構文で発生していた、いくつかの技術的な問題が起こりにくくなります[17]。これからは「～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）」と言う代わりに、「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）」と言いましょう。

例として、このエクスペクテーション（expectation）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。

[17] http://myronmars.to/n/dev-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
```

新しい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すると、今でも古い should 構文を使った情報をよく見かけると思います。この構文は RSpec 3 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?Contact 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/contact_spec.rb

```
1  require 'rails_helper'
2
3  describe Contact do
4    # 姓と名とメールがあれば有効な状態であること
5    it "is valid with a firstname, lastname and email" do
6      contact = Contact.new(
7        firstname: 'Aaron',
8        lastname: 'Sumner',
9        email: 'tester@example.com')
10     expect(contact).to be_valid
11   end
12
13   # 残りの example が続きます
14 end
```

この単純な example は RSpec の be_valid 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 Contact クラスのインスタンスを作成し、contact 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。

さて、コマンドラインからもう一度 RSpec を実行してみると(bin/rspec もしくは bundle exec rspec を使います。どちらを使うかは前の章で rspec の binstub をインストールしたかどうかによります。)…… example が一つパスしています!うまくいっているようです。ではもっとたくさんコードをテストしていきましょう。

バリデーションをテストする

自動テストに切り替えていくならバリデーションから始めると良いでしょう。バリデーションのテストはたいてい 1 ～ 2 行で書けます。ファクトリを活用すると特に短く書きやすくなります(ファクトリについては次章で説明します)。では firstname 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/contact_spec.rb

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

今回は新しく作った連絡先(*firstname* には明示的に `nil` をセットします)が有効な状態にならず、連絡先の *firstname* 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。

誤判定ではないことを証明するため、`to` を `not_to` に変えてエクスペクテーションを反転させましょう。

spec/models/contact_spec.rb

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

当然のごとく、RSpec はテストの失敗を報告します。

```
Failures:

  1) Contact is invalid without a firstname
     Failure/Error: expect(contact.errors[:firstname]).not_to
         include("can't be blank")
       expected ["can't be blank"] not to include "can't be blank"
     # ./spec/models/contact_spec.rb:15:in `block (2 levels) in
         <top (required)>'
```

RSpec ではこうしたエクスペクテーションを書くために `not_to` と `to_not` 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 `not_to` を使います。

これはテストが正しく動作していることを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。先に進む前に `not_to` を `to` に戻すことをお忘れなく。

以前のバージョンの RSpec では、バリデーションエラーをチェックするのに `have` マッチャと `errors_on` ヘルパーメソッドがよく使われていました。この機能は RSpec 3 ではコア機能から削除されました。ただし、`rspec-collection_matchers` を Gemfile の `:test` グループに追加すれば `have` マッチャを引き続き使用することができます。

同じやり方で `:lastname` のバリデーションもテストできます。

spec/models/contact_spec.rb

```
# 姓がなければ無効な状態であること
it "is invalid without a lastname" do
  contact = Contact.new(lastname: nil)
  contact.valid?
  expect(contact.errors[:lastname]).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。)

メールアドレスがユニークであることをテストする場合も大変シンプルです。

spec/models/contact_spec.rb

```
# メールアドレスが重複する場合は無効な状態である
it "is invalid with a duplicate email address" do
  Contact.create(
    firstname: 'Joe', lastname: 'Tester',
    email: 'tester@example.com'
  )
  contact = Contact.new(
    firstname: 'Jane', lastname: 'Tester',
    email: 'tester@example.com'
  )
  contact.valid?
  expect(contact.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連絡先を保存しました(new の代わりに create を呼んでいます)。それから 2 件目の連絡先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、最初に保存された連絡先は有効な状態(姓と名がどちらもある)かつ、メールアドレスも設定されている必要があります。あとの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。

ではもっと複雑なバリデーションをテストしてみましょう。たとえばユーザーは重複した電話番号を持ってはいけない、という要件を考えます。つまり、ユーザーの自宅、会社、携帯電話の電話番号がそれぞれユニークでなければいけない、ということです。あなたならこれをどうやってテストしますか?

Phone モデルのスペックに移って、次のような example を書きます。

spec/models/phone_spec.rb

```ruby
require 'rails_helper'

describe Phone do
  # 連絡先ごとに重複した電話番号を許可しないこと
  it "does not allow duplicate phone numbers per contact" do
    contact = Contact.create(
      firstname: 'Joe',
      lastname: 'Tester',
      email: 'joetester@example.com'
    )
    contact.phones.create(
      phone_type: 'home',
      phone: '785-555-1234'
    )
    mobile_phone = contact.phones.build(
      phone_type: 'mobile',
      phone: '785-555-1234'
    )

    mobile_phone.valid?
    expect(mobile_phone.errors[:phone]).to include('has already been taken')
  end

  # 2 件の連絡先で同じ電話番号を共有できること
  it "allows two contacts to share a phone number" do
    contact = Contact.create(
      firstname: 'Joe',
      lastname: 'Tester',
      email: 'joetester@example.com'
    )
    contact.phones.create(
      phone_type: 'home',
      phone: '785-555-1234'
    )
    other_contact = Contact.new
    other_phone = other_contact.phones.build(
      phone_type: 'home',
      phone: '785-555-1234'
    )

    expect(other_phone).to be_valid
  end
end
```

Contact モデルと Phone モデルは Active Record の関連を持っているので、今回はその追加情報をコードに提供してやる必要があります。最初の example では 2 件の電話番号が設定されている 1 件の連絡先を作成しました。二つ目の example では 2 件の異なる連絡先に同じ電話番号を設定していま

す。注意してほしいのはどちらの example でも連絡先を *create* している、つまりデータベースに保存している点です。これはテストしようとしている電話番号に連絡先を設定するためです。

そして Phone モデルには次のようなバリデーションを設定してあります。

app/models/phone.rb

```
validates :phone, uniqueness: { scope: :contact_id }
```

よってこれらのスペックは問題なくパスします。

もちろん、バリデーションはスコープ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。

インスタンスメソッドをテストする

連絡先の姓と名を毎回連結して新しい文字列を作るより、`@contact.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 `Contact` クラスに作ってあります。

app/models/contact.rb

```
1  def name
2    [firstname, lastname].join(' ')
3  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/contact_spec.rb

```
1  # 連絡先のフルネームを文字列として返すこと
2  it "returns a contact's full name as a string" do
3    contact = Contact.new(firstname: 'John', lastname: 'Doe',
4      email: 'johndoe@example.com')
5    expect(contact.name).to eq 'John Doe'
6  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

では、*Contact* モデルの機能をテストしましょう。テストするのは指定された一文字で名前が始まる連絡先を返す機能です。例えば、私が *S* をクリックすると *Smith* や *Sumner* は返ってくるが、*Jones* は返ってこない、といった感じです。実装方法はたくさんありますが、あくまでデモンストレーションが目的なので一つだけお見せします。

`Contact` モデルは次のようなシンプルなメソッドでこの機能を実装しています。

app/models/contact.rb

```
def self.by_letter(letter)
  where("lastname LIKE ?", "#{letter}%").order(:lastname)
end
```

この機能をテストするため、*Contact* のスペックに次のような example を追加しましょう。

spec/models/contact_spec.rb

```
require 'rails_helper'

describe Contact do

  # 前に書いたバリデーションの example は省略...

  # マッチした結果をソート済みの配列として返すこと
  it "returns a sorted array of results that match" do
    smith = Contact.create(
      firstname: 'John',
      lastname: 'Smith',
      email: 'jsmith@example.com'
    )
    jones = Contact.create(
      firstname: 'Tim',
      lastname: 'Jones',
      email: 'tjones@example.com'
    )
    johnson = Contact.create(
      firstname: 'John',
      lastname: 'Johnson',
      email: 'jjohnson@example.com'
    )
    expect(Contact.by_letter("J")).to eq [johnson, jones]
  end
end
```

クエリの結果とソートの順番を両方テストしている点に注意してください。jones が先にデータベースから取得されるはずですが、姓（lastname）でソートしているのでクエリの結果としては johnson が先に取得されます。

失敗をテストする

正常系のテストは終わりました。ユーザーが名前を選ぶと結果が返ってきます。しかし、結果が返ってこない文字を選んだ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/contact_spec.rb

```
require 'rails_helper'

describe Contact do

  # バリデーションの example が並ぶ...

  # マッチしなかったものは結果に含まれないこと
  it "omits results that do not match" do
    smith = Contact.create(
      firstname: 'John',
      lastname: 'Smith',
      email: 'jsmith@example.com'
    )
    jones = Contact.create(
      firstname: 'Tim',
      lastname: 'Jones',
      email: 'tjones@example.com'
    )
    johnson = Contact.create(
      firstname: 'John',
      lastname: 'Johnson',
      email: 'jjohnson@example.com'
    )
    expect(Contact.by_letter("J")).not_to include smith
  end
end
```

このスペックでは RSpec の include マッチャを使って Contact.by_letter("J") で返ってきた配列を検証しています。そしてパスしました!これで理想的な結果、すなわちユーザーが結果が返ってくる文字列を選んだ場合だけでなく、結果が返ってこない文字を選んだ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

これまで三つのマッチャ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[18]の README が参考になるかもしれません。また、第 7 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

もしあなたがここまでサンプルコードを見てきたのであれば、ここで説明した内容との食い違いを見つけているはずです。サンプルコードではまた別の RSpec の機能を使っています。それは before フッ

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

クです。before フックを使うとスペックのコードがシンプルになり、タイプ量も減らせます。実際、スペックの example にはいくつか冗長な部分があります。たとえば各 example で全く同じ三つのオブジェクトを作っているところです。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 のトリックを使ってテストコードをきれいにしてみましょう。

最初にやることは describe Contact ブロックの 中に describe ブロックを作ることです。これはフィルタ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/contact_spec.rb

```
1  require 'rails_helper'
2
3  describe Contact do
4
5    # バリデーションの example が並ぶ...
6
7    # 文字で姓をフィルタする
8    describe "filter last name by letter" do
9      # フィルタの example が並ぶ...
10   end
11 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。context の一つは文字にマッチする場合で、もう一つは文字にマッチしない場合です。

spec/models/contact_spec.rb

```
1  require 'rails_helper'
2
3  describe Contact do
4
5    # バリデーションの example が並ぶ...
6
7    # 文字で姓をフィルタする
8    describe "filter last name by letter" do
9      # マッチする文字の場合
10     context "matching letters" do
11       # マッチする場合の example が並ぶ...
12     end
13
14     # マッチしない文字の場合
15     context "non-matching letters" do
16       # マッチしない場合の example が並ぶ...
17     end
18   end
19 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文字が選ばれた状態で、もう一つは返ってこない文字が選ばれた状態です。

気付いたかもしれませんが、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。

spec/models/contact_spec.rb

```
1  require 'rails_helper'
2
3  describe Contact do
4
5    # バリデーションの example が並ぶ...
6
7    # 文字で姓をフィルタする
8    describe "filter last name by letter" do
9      before :each do
10       @smith = Contact.create(
11         firstname: 'John',
12         lastname: 'Smith',
13         email: 'jsmith@example.com'
14       )
15       @jones = Contact.create(
16         firstname: 'Tim',
17         lastname: 'Jones',
18         email: 'tjones@example.com'
19       )
20       @johnson = Contact.create(
21         firstname: 'John',
22         lastname: 'Johnson',
23         email: 'jjohnson@example.com'
24       )
25     end
26
27     # マッチする文字の場合
28     context "matching letters" do
29       # マッチする場合の example が並ぶ...
30     end
31
32     # マッチしない文字の場合
33     context "non-matching letters" do
34       # マッチしない場合の example が並ぶ...
35     end
36   end
37 end
```

RSpec の before フックはスペック内の汚い重複コードをきれいにするために必要不可欠な機能です。ご想像の通り、before ブロックの中にあるコードは、describe ブロックの中にある 各 example の前（before）に実行されます。ただし、describe ブロックの外にある example は対象外となります。このケースでは describe ブロックの中にある 各 example の前に、before フックを実行させるようにしたので、RSpec は example を実行するたびに別個のテストデータを作ってくれるのです。また、before フックは describe "filter last name by letter" ブロックの中で のみ 呼ばれます。つまり、元々あったバリデーションのスペックからは @smith 、@jones 、@johnson にアクセスできません。

:each は before の初期値です。Rubyist の多くは before ブロックを作る際に before do と短く書きます。私は before :each do と明示的に書く方が好みなので、本書では今後もこの記法を使います。

また、三つの連絡先（テストデータ）で注意したいのは、これらの連絡先は各 example の内部では作成されないので、インスタンス変数にセットしなければならないという点です。インスタンス変数に設定しているからこそ、before ブロックの外側、すなわち各 example の内部で作成した連絡先にアクセスできるのです。

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。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 after を使うことは滅多にありません。しかし、before フックはなくてはならないものです。

さて、整理後の全スペックを見てみましょう。

spec/models/contact_spec.rb

```
1  require 'rails_helper'
2
3  describe Contact do
4    # 姓と名とメールがあれば有効な状態であること
5    it "is valid with a firstname, lastname and email" do
6      contact = Contact.new(
7        firstname: 'Aaron',
8        lastname: 'Sumner',
9        email: 'tester@example.com')
10     expect(contact).to be_valid
11   end
12
13   # 名がなければ無効な状態であること
14   it "is invalid without a firstname" do
15     contact = Contact.new(firstname: nil)
16     contact.valid?
17     expect(contact.errors[:firstname]).to include("can't be blank")
18   end
19
20   # 姓がなければ無効な状態であること
21   it "is invalid without a lastname" do
22     contact = Contact.new(lastname: nil)
23     contact.valid?
24     expect(contact.errors[:lastname]).to include("can't be blank")
25   end
26
27   # メールアドレスがなければ無効な状態であること
28   it "is invalid without an email address" do
29     contact = Contact.new(email: nil)
30     contact.valid?
31     expect(contact.errors[:email]).to include("can't be blank")
32   end
33
34   # 重複したメールアドレスなら無効な状態であること
```

```
35    it "is invalid with a duplicate email address" do
36      Contact.create(
37        firstname: 'Joe', lastname: 'Tester',
38        email: 'tester@example.com'
39      )
40      contact = Contact.new(
41        firstname: 'Jane', lastname: 'Tester',
42        email: 'tester@example.com'
43      )
44      contact.valid?
45      expect(contact.errors[:email]).to include("has already been taken")
46    end
47
48    # 連絡先のフルネームを文字列として返すこと
49    it "returns a contact's full name as a string" do
50      contact = Contact.new(
51        firstname: 'John',
52        lastname: 'Doe',
53        email: 'johndoe@example.com'
54      )
55      expect(contact.name).to eq 'John Doe'
56    end
57
58    # 文字で姓をフィルタする
59    describe "filter last name by letter" do
60      before :each do
61        @smith = Contact.create(
62          firstname: 'John',
63          lastname: 'Smith',
64          email: 'jsmith@example.com'
65        )
66        @jones = Contact.create(
67          firstname: 'Tim',
68          lastname: 'Jones',
69          email: 'tjones@example.com'
70        )
71        @johnson = Contact.create(
72          firstname: 'John',
73          lastname: 'Johnson',
74          email: 'jjohnson@example.com'
75        )
76      end
77
78      # マッチする文字の場合
79      context "with matching letters" do
80        # マッチした結果をソート済みの配列として返すこと
81        it "returns a sorted array of results that match" do
82          expect(Contact.by_letter("J")).to eq [@johnson, @jones]
83        end
```

```
84      end
85
86      # マッチしない文字の場合
87      context "with non-matching letters" do
88        # マッチしなかったものは結果に含まれないこと
89        it "omits results that do not match" do
90          expect(Contact.by_letter("J")).not_to include @smith
91        end
92      end
93    end
94  end
```

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）。

```
Contact
  is valid with a firstname, lastname and email
  is invalid without a firstname
  is invalid without a lastname
  is invalid without an email address
  is invalid with a duplicate email address
  returns a contact's full name as a string
  filter last name by letter
    with matching letters
      returns a sorted array of results that match
    with non-matching letters
      omits results that do not match

Phone
  does not allow duplicate phone numbers per contact
  allows two contacts to share a phone number

Finished in 0.51654 seconds (files took 2.24 seconds to load)
10 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 `filter last name by letter` と書いたラベルは `#by_letter` 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。すでに述べたように、`before` ブロックはスペックを整理する際に不可欠なテクニックです。しかし、これは弊害を起こしやすいテクニックでもあります。

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 `@jones` と `@johnson` をテスト用の連絡先として使いました。こうした名前は `@user1` と `@user2` と書くよりもずっと理解しやすいです。なぜならテスト用オブジェクトの値の一部をテストで使っているからです。ここでは頭文字検索機能が正しく動いているかどうかを確認するために、オブジェクトの値を使っているのでした。第 6 章で特定のロールを持ったユーザをテストするときには、`@admin_user` や `@guest_user` といった名前の変数を使うともっと良いです。変数には意味のある名前を付けましょう!

まとめ

本章では私がモデルをテストする方法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。次章ではここで説明したテクニックをさらに深く掘り下げて、アプリケーションコントローラへつなげていきます。

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ある状態[19]をまとめるために `context` を使うこともできます。

演習問題

ここまで私たちは、自分が書いたスペックは誤判定していない、という仮定のもとで話を進めてきました。スペックはどれも途中でも失敗することなく、保留中の状態からパスしている状態に変わりました。次のようなことをやってみて、スペックが本当に有効かどうかをチェックしてください。

[19] http://lmws.net/describe-vs-context-in-rspec

- **テスト中のアプリケーションコードをコメントアウトしてみる。**たとえば、連絡先のスペックでは名の必須バリデーションを検証する example があります。そこで、`validates :firstname, presence: true` をコメントアウトしてからスペックを実行し、`it "is invalid without a firstname"` が失敗するのを確認してみましょう。それからコメントを元に戻して、スペックがもう一度パスするのを確認してください。
- **エクスペクテーションの中で `create` メソッドに渡されるパラメータを変更してみる。**今度は `it "is invalid without a firstname"` を変更して、`:firstname` に `nil` 以外の値を渡してみましょう。スペックは失敗するはずです。それからまた `nil` に戻して、スペックが再度パスすることを確認してください。

4. ファクトリを使ったテストデータの生成

ここまで私たちは ごく普通の Ruby オブジェクト（plain old Ruby objects） を使ってテスト用の一時データを作ってきました。そしてここまではあまり複雑なテストではなかったので、これで十分でした。しかし、ここからはより複雑なシナリオをテストするので、テストデータ作成のプロセスを単純化し、テストデータよりもテストそのものにフォーカスした方が良いはずです。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 Factory Girl に焦点を当てます。Factory Girl を使う方が多くの開発者にとっては良いアプローチとなるはずです。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。
- 次に Faker gem を使ってより本物っぽいテストデータを作成します。
- 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。

本章の完成後のコードを見たい場合はサンプルソースの *04_factories* ブランチをチェックアウトしてください。コマンドラインを使う場合は次のようにタイプします。

```
git checkout -b 04_factories origin/04_factories
```

本章を読みながら一緒にコードも書いていく場合は、前の章のブランチから始めてください。

```
git checkout -b 03_models origin/03_models
```

その他、詳細については第 1 章を読んでください。

factory_girl_rails と *faker* gem のインストールが済んでいなければ、第 2 章で説明した方法に従ってインストールしておいてください。

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最初から提供されています。フィクスチャは本質的には 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、`Contact` モデルのフィクスチャなら次のようになります。

contacts.yml

```
aaron:
  firstname: "Aaron"
  lastname: "Sumner"
  email: "aaron@everydayrails.com"

john:
  firstname: "John"
  lastname: "Doe"
  email: "johndoe@nobody.org"
```

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

フィクスチャが向いている場面も確かにあるのですが、同時に欠点も持っています。私はフィクスチャの悪口を言うために時間を使いたくありません。そもそも、そんなことは私よりも頭のいい人たちが Rails のテストコミュニティでさんざん議論してきた話です。端的に言うと、フィクスチャには避けて通りたい二つの問題があります。一つはフィクスチャのデータは脆くて簡単に壊れやすいという点です（これはつまり、テストデータのメンテナンスにかかる時間が、コードとテストを書くのと同じぐらいになってしまうということです）。もう一つの問題は Rails が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 を使わないという点です。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。これは最悪です！

そこで**ファクトリ**を導入しましょう。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。もし私がテストの素晴らしさに目覚めるきっかけにな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 Girl[20]を挙げると思います。Factory Girl は使いやすくて頼もしい gem です。Factory Girl を使えばフィクスチャのような壊れやすさに悩まされることなくテストデータを作成できます。

Ruby コミュニティではいつでもベストプラクティスに関する議論が繰り広げられています。なので、当然 Factory Girl に反対する人もいます。2012 年の夏にはネット上でファクトリの利点に関する議論[21]が巻き起こりました。反対意見として多かったのは、ファクトリを使うとテストが遅くなる一番の原因になることと、ファクトリで複雑な関連を扱おうとすると途端に面倒になる、というものでした。反対意見を述べる人の中には Rails の作者である David Heinemeier Hansson も含まれていました。

彼らの意見にも同意できる部分はありますし、気軽にファクトリを使うとスピード面でのコストが増えてしまうというのもわかります。しかし、遅くてもテストが全くないよりかは良いですし、ファクトリを使ったテストはシンプルになるのでテスト初心者にも易しいです。私はそのように考えています。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。

その時までこのアプリケーションではファクトリを使うようにしましょう。*factory_girl_rails* gem はインストール済みなので（第 2 章を参照）、もう準備はできています。

アプリケーションにファクトリを追加する

spec ディレクトリに戻り、*factories* という名前のサブディレクトリを追加してください。そのサブディレクトリに *contacts.rb* というファイルを作って次の内容を追加してください。

spec/factories/contacts.rb

```
1  FactoryGirl.define do
2    factory :contact do
3      firstname "John"
4      lastname "Doe"
5      sequence(:email) { |n| "johndoe#{n}@example.com"}
6    end
7  end
```

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。`FactoryGirl.create(:contact)` を使ってテストデータを作れば、その連絡先の名前は毎回基本的に *John Doe* になります。ではメールアドレスはどうなるのでしょうか？ここでは**シーケンス（sequences）**と呼ばれる Factory Girl の便利機能を使っています。この Ruby のコードを読めば予想が付くかもしれませんが、シーケンスはブロックの内部で `n` を自動的にインクリメントします。新しい連絡先を作るためにファクトリが呼ばれると、

[20] https://github.com/thoughtbot/factory_girl

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

`johndoe1@example.com`、`johndoe2@example.com` というようにメールアドレスが生成されます。シーケンスは `uniqueness`（一意な）バリデーションを持つモデルには必要不可欠です。(ちなみに本章の後半では Faker というライブラリも紹介します。このライブラリは住所や氏名を生成するのに役立ちます。)

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。Ruby コードから動的に値を設定することもできます。その場合はブロックの内部で値を作ることを忘れないでください（上で紹介したシーケンスの使用例と同じです）。たとえば連絡先に誕生日を保存する場合、`33.years.ago` や `Date.parse('1980-05-13')` のような Ruby の datetime メソッドを使って簡単に日付を生成することができます。

ファクトリのファイル名にはスペックファイルのような厳密なルールはありません。実際、一つのファイルに全てのファクトリを含めることもやろうと思えばできます。しかし、慣例として Factory Girl のジェネレータは、対応するモデルの複数形をファクトリのファイル名にして *spec/factories* ディレクトリに保存します（よって Contact モデルであれば *spec/factories/contacts.rb* になります）。私もだいたいその慣例に従います。最低限必要なことは、ファクトリの定義が構文的に正しく、なおかつそのファイルが *spec/factories/* ディレクトリに配置されていることです。そうすれば特に問題は起きないはずです。

ファクトリができたので、前の章で作った *contact_spec.rb* に戻って簡単な example を追加してみましょう。

spec/models/contact_spec.rb

```
1  require 'rails_helper'
2
3  describe Contact do
4    # 有効なファクトリを持つこと
5    it "has a valid factory" do
6      expect(FactoryGirl.build(:contact)).to be_valid
7    end
8
9    # その他のスペック
10 end
```

こうすると新しい連絡先がインスタンス化されます（ただし保存はされません）。属性はファクトリがセットします。続いて新しい連絡先の有効性をテストしています。前の章で書いたスペックと比較してみましょう。以前のスペックではテストをパスさせるのに必要な属性をすべて含める必要がありました。

```
1  # 姓と名とメールがあれば有効な状態であること
2  it "is valid with a firstname, lastname and email" do
3    contact = Contact.new(
4      firstname: 'Aaron',
5      lastname: 'Sumner',
6      email: 'tester@example.com')
7    expect(contact).to be_valid
8  end
```

既存のスペックに戻り、Factory Girl を使って簡潔にデータを作成してみましょう。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/contact_spec.rb

```
# 名がなければ無効な状態であること
it "is invalid without a firstname" do
  contact = FactoryGirl.build(:contact, firstname: nil)
  contact.valid?
  expect(contact.errors[:firstname]).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 lastname" do
  contact = FactoryGirl.build(:contact, lastname: nil)
  contact.valid?
  expect(contact.errors[:lastname]).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
  contact = FactoryGirl.build(:contact, email: nil)
  contact.valid?
  expect(contact.errors[:email]).to include("can't be blank")
end

# 連絡先のフルネームを文字列として返すこと
it "returns a contact's full name as a string" do
  contact = FactoryGirl.build(:contact,
    firstname: "Jane",
    lastname: "Smith"
  )
  expect(contact.name).to eq 'Jane Smith'
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 Girl の `build` メソッドを使って新しい(ただし保存されていない)`Contact` を作成しています。最初の example では `firstname` が空になっている `Contact` を `contact` 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 `lastname` を *nil* で置き換えています。`Contact` モデルは `firstname` と `lastname` 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テストの期待値です。`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。

4 番目の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 `firstname` と `lastname` に特定の値をセットして新しい `Contact` を作っています。それから変数に入れた `contact` 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。

次のスペックにはちょっと新しい内容が出てきます。

spec/models/contact_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
  FactoryGirl.create(:contact, email: 'aaron@example.com')
  contact = FactoryGirl.build(:contact, email: 'aaron@example.com')
  contact.valid?
  expect(contact.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別の `Contact` 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、`FactoryGirl.create` を使って同じメールアドレスの連絡先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。`FactoryGirl.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。`FactoryGirl.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。

構文をシンプルにする

私が知っている大半のプログラマは必要以上にタイピングするのが大嫌いです。新しい連絡先が必要になるたびに `FactoryGirl.build(:contact)` と毎回入力するのはもう面倒になってきました。幸いなことに、Factory Girl 3.0 以降では簡単な設定をするだけで Rails プログラマの日常生活をちょっぴりシンプルにしてくれます。*rails_helper.rb* にある `RSpec.configure` ブロックの内側に、次のような設定を追加してください。場所はどこでも構いません。

spec/rails_helper.rb

```
1  RSpec.configure do |config|
2    # ファクトリを簡単に呼び出せるよう、Factory Girl の構文をインクルードする
3    config.include FactoryGirl::Syntax::Methods
4
5    # その他の設定は省略...
6  end
```

このついでに `config.fixture_path` の設定を削除、またはコメントアウトすることもできます。なぜならフィクスチャはもう使っていないからです！

こうすると `build(:contact)` という構文が使えるようになり、スペックが短く書けます。同様に、`create(:contact)` という構文も使えますし、これよりあとの章で使用する `attributes_for(:contact)` や `build_stubbed(:contact)` も短く書けます。

ではここでシンプルに生まれ変わったモデルスペックを見てみましょう。

spec/models/contact_spec.rb

```
require 'rails_helper'

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

  # 名がなければ無効な状態であること
  it "is invalid without a firstname" do
    contact = build(:contact, firstname: nil)
    contact.valid?
    expect(contact.errors[:firstname]).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 lastname" do
    contact = build(:contact, lastname: nil)
    contact.valid?
    expect(contact.errors[:lastname]).to include("can't be blank")
  end

  # 残りの example は省略 ...
end
```

私に言わせるとずっと読みやすくなったと思いますが、これをあなた自身のコードで採用するかどうかはお任せします。

関連とファクトリの継承

Phone モデルのファクトリを作る場合、これまでに学んだテクニックを使うとこんなふうになります。

spec/factories/phones.rb

```
FactoryGirl.define do
  factory :phone do
    association :contact
    phone '123-555-1234'
    phone_type 'home'
  end
end
```

ここで登場する新しい内容は :association メソッドです。:association を使うと、Factory Girl はこの電話番号が属する新しい Contact を自動的に作成します。ただし、build（または create ）メソッドに Contact が明示的に渡された場合はそのオブジェクトが使われます。

しかし、連絡先の電話番号には自宅用、会社用、携帯用の 3 種類があります。ここまでの知識を使ってスペック内で自宅用電話番号を作ろうとすると、次のようなコードを書くことになります。

spec/models/phone_spec.rb

```
# 2 件の連絡先で同じ電話番号を共有できること
it "allows two contacts to share a phone number" do
  create(:phone,
    phone_type: 'home',
    phone: "785-555-1234")
  expect(build(:phone,
    phone_type: 'home',
    phone: "785-555-1234")).to be_valid
end
```

このコードをきれいにするためにちょっとリファクタリングしてみましょう。Factory Girl では属性を必要に応じてオーバーライドするために 継承 ファクトリを作ることができます。たとえばスペック内で会社用の電話番号を作りたい場合には、`build(:office_phone)`（もしくは省略しない方が好みであれば `FactoryGirl.build(:office_phone)`）と書くことができるのです。コードは次のようになります。

spec/factories/phones.rb

```
FactoryGirl.define do
  factory :phone do
    association :contact
    phone '123-555-1234'

    factory :home_phone do
      phone_type 'home'
    end

    factory :work_phone do
      phone_type 'work'
    end

    factory :mobile_phone do
      phone_type 'mobile'
    end
  end
end
```

そしてスペックも次のようにシンプルになります。

spec/models/phone_spec.rb

```
require 'rails_helper'

describe Phone do
  # 連絡先ごとに重複した電話番号を許可しないこと
  it "does not allow duplicate phone numbers per contact" do
    contact = create(:contact)
    create(:home_phone,
      contact: contact,
      phone: '785-555-1234'
    )
```

```
11      mobile_phone = build(:mobile_phone,
12        contact: contact,
13        phone: '785-555-1234'
14      )
15
16      mobile_phone.valid?
17      expect(mobile_phone.errors[:phone]).to include('has already been taken')
18    end
19
20    # 2 件の連絡先で同じ電話番号を共有できること
21    it "allows two contacts to share a phone number" do
22      create(:home_phone,
23        phone: '785-555-1234'
24      )
25      expect(build(:home_phone, phone: "785-555-1234")).to be_valid
26    end
27  end
```

このテクニックは異なるユーザタイプ(管理者と非管理者)を作るときに大変役立ちます。詳しくは次章以降で認証と認可の機能をテストする際に説明します。

もっとリアルなダミーデータを作成する

本章の前半で私たちは シーケンス を使って、ファクトリが一意なメールアドレスを作り出せるようにしました。この仕組みはさらに改善することができます。もっとリアルなテストデータを作り出すことができるのです。そんなダミーデータを作り出すライブラリを紹介しましょう。その名も Faker です。Faker は由緒ある Perl ライブラリの Ruby バージョンです。Faker を使えば名前や住所、文章、その他たくさんのダミーデータを作ることができます。テストにはぴったりのライブラリです。

ではファクトリにダミーデータを組みこんでみましょう。

spec/factories/contacts.rb

```
1   FactoryGirl.define do
2     factory :contact do
3       firstname { Faker::Name.first_name }
4       lastname { Faker::Name.last_name }
5       email { Faker::Internet.email }
6     end
7   end
```

こうするとスペックは電話番号のファクトリが使われるたびにランダムなメールアドレスを使うようになります。(自分の目で確かめたい人は、*contact_spec.rb* を走らせたあとに *log/test.log* を開き、データベースにインサートされたメールアドレスを確認してみてください。)ここで確認しておきたい重要なポイントは次の 2 点です。一つ目はファクトリの最初の行で Faker ライブラリを require して読み込んでいる点です。二つ目はブロックの内部で `Faker::Internet.email` を渡している点です。これまでのファクトリで使われてきた静的な文字列とは異なり、ブロックを使うと Factory Girl はそれを “遅延評価すべき属性” と見なします。

それでは電話番号のファクトリに戻ってこの演習問題を締めくくりましょう。毎回新しい電話番号オブジェクトに決まった番号を付与するのではなく、一意で、ランダムで、本物っぽい番号を付与するようにしてみます。

spec/factories/phones.rb

```
FactoryGirl.define do
  factory :phone do
    association :contact
    phone { Faker::PhoneNumber.phone_number }

    # 子ファクトリの記述は省略...
  end
end
```

厳密に言えば、確かにこれは絶対必要な仕組みではありません。このままシーケンスを使い続けることもできますし、そうしてもスペックはずっとパスします。しかし、Faker を使えばテストデータがそれよりもちょっとリアルなデータになります（そればかりではなく、Faker は時々とてもゆかいなデータを生成します）。

Faker はランダムな住所やダミーの企業名、キャッチコピー、*lorem ipsum* 風のプレースホルダテキストなど、様々なタイプのデータを生成します。詳しくは Faker のドキュメント[22]を参照してください。

Faker の代替候補を探すのであれば、Forgery[23]をチェックしてみてください。Forgery も Faker によく似た機能を持っていますが、構文が少し異なります。また、ffaker[24]という gem もあります。これは Faker を書き直したライブラリで、Faker よりも 20 倍以上速く動作します。そして、こうした gem はテストの際にだけ役立つわけではありません。気になる方は Everyday Rails blog のhow to use Faker to obfuscate data for screenshots[25]（Faker を使ってスクリーンショットをごまかす方法）を読んでみてください。

高度な関連

私たちがこれまでに作ってきたスペックは大半が比較的簡単なデータのテストばかりでした。テストの対象のモデルを見るだけで良かったのです。つまり、連絡先を作ったときに 3 件の電話番号も同時に作られることは検証していませんでした。これはどうやってテストするのでしょうか？そしてファクトリを使えばいつでも本物と変わらないテスト用の連絡先が作られることをどうやって保証すれば良いでしょうか？

その答えは Factory Girl の コールバック（callback） を使い、ファクトリにコードを追加することです。コールバックは特にネストした属性（nested attributes）をテストするときに便利です。ネストした属性は連絡先管理アプリケーションのユーザインタフェースでも使われていて、連絡先を作ったり編集したりするときに電話番号も入力できるようになっています。たとえば、連絡先のファクトリで `after` コールバックを使うように変更してみます。次のように、ファクトリで新しい連絡先を作った場合は 3 種類の電話番号も一緒に作成し、連絡先に関連付けることができます。

[22] http://rubydoc.info/gems/faker/1.4.3/frames
[23] https://github.com/sevenwire/forgery
[24] https://github.com/emmanueloga/ffaker
[25] http://everydayrails.com/2013/05/20/obfuscated-data-screenshots.html

spec/factories/contacts.rb

```
FactoryGirl.define do
  factory :contact do
    firstname { Faker::Name.first_name }
    lastname { Faker::Name.last_name }
    email { Faker::Internet.email }

    after(:build) do |contact|
      [:home_phone, :work_phone, :mobile_phone].each do |phone|
        contact.phones << FactoryGirl.build(:phone,
          phone_type: phone, contact: contact)
      end
    end
  end
end
```

`after(:build)` がブロックを受け取っている点に注目してください。さらにそのブロックの中では連絡先の電話番号を作るために 3 種類の電話タイプの配列が使われています。このコールバックの動作を検証する example は次のようになります。

spec/models/contact_spec.rb

```
# 3 つの電話番号を持つこと
it "has three phone numbers" do
  expect(create(:contact).phones.count).to eq 3
end
```

この example はパスします。既存の example もすべてパスします。つまり、ファクトリを変更しても既存のテストが壊れることはありませんでした。この検証をもう一歩進めることができます。Contact モデルにバリデーションを追加して、必ず電話番号が作られることを保証するのです。

app/models/contact.rb

```
validates :phones, length: { is: 3 }
```

実験的にバリデーションの値を別の数字に変更し、テストスイートを再実行してみてください。連絡先の有効性を検証するテストはすべて失敗するはずです。二つ目の実験として連絡先ファクトリの `after` ブロックをコメントアウトしてからテストスイートを実行してみてください。今度もやはり、真っ赤になって終わるはずです。

今回の例では連絡先に関連付けられる 3 件の電話番号の動作にしか使いませんでしたが、コールバックの使い方に全く制限はありません。この機能の便利な使い方をもっと知りたい人は、Thoughtbot の Get Your Callbacks On with Factory Girl 3.3[26] というブログポストをチェックしてみてください。

この例はちょっとわざとらしく思えるかもしれませんが、これはあなたが遅かれ早かれ複雑なアプリケーションに出くわすことを予想してのことです。実際、この例は私が昔作ったスケジュール管理システムに基づいています。このシステムでは参加者の最低人数が決まっていました。このときは Factory Girl のドキュメントやコードを調べたり、インターネットをあちこち検索したりして、ファクトリを要件通りに動かすまでに結構時間がかかりました。

[26] http://robots.thoughtbot.com/post/23039827914/get-your-callbacks-on-with-factory-girl-3-3

`after(:build)` はあなたが自由に使えるコールバックのひとつに過ぎません。ご想像どおり、`before(:build)`、`before(:create)`、`after(:create)` といったコールバックも使えます。これらはすべて同じように動作します。

ファクトリの弊害

ファクトリは素晴らしいものです。しかし、時にはそうでないときもあります。本章の冒頭で述べたように、むやみにファクトリを使うとテストが急に遅くなる場合があります。複雑な関連が持ち込まれると特に遅くなりやすいです。実際、最後のファクトリで、ファクトリが呼び出されるたびに毎回三つの余分なオブジェクトが作られるようにしたのは、ちょっとやりすぎだったかもしれません。とはいえ、少なくとも現時点では欠点よりも利便性の方が勝っています。何回もメソッドを呼び出す代わりに、一回の呼び出しでテストデータが生成できるからです。

ファクトリを使って関連するオブジェクトを作成するのはテストの価値を高める簡単な方法です。しかし、それは同時に弊害を起こしやすい機能でもあります。テストスイートの実行時間が遅くなった原因を調べてみたらファクトリが原因だった、ということもよくあります。そんなときはファクトリから関連を取り除き、手動でテストデータを作った方が良いでしょう。第 3 章でやったように ごく普通の Ruby オブジェクト（plain old Ruby object） に戻すこともできますし、ファクトリでその両方を使うハイブリッドなアプローチも考えられます。

もしあなたがこれまでに一般的なテスト技法や RSpec 固有の情報源を見てきたのであれば、きっとモック（mocks） や スタブ（stubs） という用語に出会っているはずです。多少なりともテストの経験がある人は、私がずっとファクトリばかり使ってモックやスタブを使わないのを不思議に思っているかもしれません。私がここでモックやスタブを使わない理由は、自分の経験上、普通のオブジェクトやファクトリの方が初心者に易しく、テストに早く慣れることができると考えているからです。また、当然ですが、モックやスタブを濫用すると別の問題を引き起こす可能性があります。

現段階ではこのアプリケーションはまだまだ小さいので、より手の込んだアプローチを使ってスピードを上げても大した違いにはなりません。とはいえ、モックとスタブもテストにおける重要な要素です。モックとスタブについては第 9 章と第 10 章で詳しく説明します。

まとめ

Factory Girl は本章で大変役に立ちました。短い構文でスペックを書くこともできますし、特定の型のデータやより本物っぽいダミーデータを様々な方法で作成することもできます。必要に応じて複雑な関連を構築することもできます。こうした知識は今後も大半のテストで活用できるはずです。とはいえ、もっと多くの使用例を学ぶためにFactory Girl のドキュメント[27]も参照するようにしてください。Factory Girl にはそれだけで短い本が書けるくらい様々な使い方があります。

Factory Girl は完璧ではありませんが、本書ではこのあとも Factory Girl を使っていきます。なぜならテストのスキルが上がっていくほど、スピードの問題よりも Factory Girl によって得られる利便性の方が重要になるからです。実際、Factory Girl 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

- 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。
- スペック内で Factory Girl の短い構文が使えるように RSpec を設定してください。構文が短くなると example の可読性にどんな影響があるでしょうか？

[27] https://github.com/thoughtbot/factory_girl/blob/master/GETTING_STARTED.md

- あなたのファクトリをじっくり見てみましょう。どうすれば継承ファクトリを使ったリファクタリングができるでしょうか？
- あなたのモデルには Faker に置き換えやすいデータ型がありますか？必要であればもう一度 Faker のドキュメントを読み、使えそうな場所に Faker のメソッドを適用してください。そのあともスペックはパスするでしょうか？
- あなたのモデルにネストした属性（nested attributes）はありますか？`after(:build)` コールバックを使うともっと本物に近いテストデータになりませんか？

5. コントローラスペックの基礎

コントローラは不憫です。なぜなら Rails 開発者は薄い（skinny）コントローラを良しとして（これは良いことですが）、あまりテストを書こうとしないからです（これは良くないです。このあと詳しく説明します）。しかし、私たちはもっとテストのカバレッジを上げていこうとしているので、次はコントローラのテストをターゲットにしましょう。

コントローラのテストで難しいところの一つは、コントローラがたくさんの要素に依存していることです。たとえばモデル同士の関連の仕方やルーティングの設定によって依存関係の複雑さが変わってきます。本章ではこうした難しさに対処していきます。一度全体を理解できれば、今後自分のソフトウェアでどのようにコントローラのスペックを書けば良いのかはっきり理解できるはずです。がんばりましょう。

この章ではやや早足で次のような内容を説明します。

- 最初に、なぜコントローラをテストすべきなのか、という点について考えてみます。
- 続いてコントローラのテストの基礎を説明します（コントローラスペックはただの単体テストにすぎない、という説明です）。
- 次にコントローラスペックを整理して、アウトラインっぽくまとめます。
- それからファクトリを使ってスペック用のデータをセットアップします。
- そのあと、コントローラで一般的な 7 つの CRUD メソッドをテストします。また、非 CRUD なメソッドの例も続けて見ていきます。
- 次に入れ子になったルーティング（nested routes）を説明します。
- 最後に、CSV や JSON のような非 HTML メソッドをテストして締めくくります。

本章の完成後のコードを見たい場合はサンプルソースの *05_controller_basics* ブランチをチェックアウトしてください。コマンドラインを使う場合は次のようにタイプします。

```
git checkout -b 05_controller_basics origin/05_controller_basics
```

本章を読みながら一緒にコードも書いていく場合は、前の章のブランチから始めてください。

```
git checkout -b 04_factories origin/04_factories
```

その他、詳細については第 1 章を読んでください。

この演習問題ではサンプルアプリケーションの `contacts_controller.rb` ファイルを少し修正する必要があります。コントローラテストの基礎だけに集中するため、この章の間だけアプリケーションの認証機能を取り除きましょう。最も簡単な方法は認証用のコードをコメントアウトすることです:

app/controllers/contacts_controller.rb

```
1  class ContactsController < ApplicationController
2    # before_action :authenticate, except: [:index, :show]
3    before_action :set_contact, only: [:show, :edit, :update, :destroy]
4
5    # etc.
```

なぜコントローラをテストするのか？

コントローラのメソッドを個別にテストするのにはちゃんとした理由がいくつかあります。

- **コントローラもメソッドを持ったクラスである。**この点についてはPiotr Solnica が素晴らしいブログ記事を書いています[28]。そして Rails アプリケーションにおいて、コントローラはかなり重要なクラス（とメソッド）です。なので、スペック的にモデルと平等に扱うのは良い考えです。
- **コントローラスペックは統合テスト用のスペックで同じ事をやるのに比べて速く書けることが多い。**私の場合、この点は非常に重要です。なぜならコントローラレベルで存在しているバグに対処したり、ちょっとしたリファクタリングを検証するためにスペックを追加したくなったりすることがあるからです。堅牢なコントローラスペックを書くのは比較的単純な作業です。なぜかといえば、フィーチャスペックとは異なり、実行コストを上げずに目的のメソッドに特定の入力値を与えることができるからです。そしてこれは次のようなことも意味しています。
- **コントローラスペックは通常、統合テスト用のスペックよりも素早く動作する。**これはバグを直したり、異常系の操作（もちろん正常系の操作も）を確認したりする際にとても助かります。

なぜコントローラをテストしないのか？

それではコントローラのスペックをしっかり書いているオープンソースの Rails プロジェクトをあまり見かけないのはなぜでしょうか？考えられる理由を挙げてみます。

- **コントローラは薄い。**薄いのでテストをしても意味がないと言う人もいます。
- **フィーチャスペックよりも速いとはいえ、コントローラスペックもモデルや普通の Ruby オブジェクトのスペックよりは遅い。**この点については第 9 章でスペックの速度を上げる方法を説明するので、いくらか軽減することはできます。とはいえ、確かにもっともな意見です。
- **フィーチャスペックが一つあれば、複数のコントローラスペックを書いたことになる。**なので複数のスペックを書いたりメンテナンスしたりする代わりに、一つのスペックでそうする方がシンプルになるかもしれません。

結局、本当の答えはこれらのどこか中間にあるんだろうと私は考えています。本書を発行してまもない頃の版では、自分自身でもコントローラスペックを書くことには葛藤があるとお話ししました。私が RSpec と TDD を学んでいるときは、ツールとプロセスの隅々まで理解するためにコントローラスペックを理解することは必要不可欠でした。そしてそれが本章でしっかりコントローラスペックを見ていこうとする理由です。コントローラスペックは数多くの RSpec の機能を使う非常に良い練習台です。なぜならモデルやフィーチャのスペックでは普段使わない機能がたくさん登場するからです。加えて、フィーチャスペックのように実行コストを上げることなくコントローラの動きをテストできるのは、やはり良いことです。

コントローラテストの基礎

Scaffold は正しく利用すればそこからコーディングテクニックを学ぶことができるので、非常に役立ちます。少なくとも RSpec 2.8 の時点ではコントローラ用に生成されたスペックファイルは非常に良くできていて、自分でスペックを書く際の良いテンプレートになります。rspec-rails のソース[29] を見るか、適切に設定された RSpec-Rails アプリケーションで scaffold を生成してみましょう。これらのファイルはコントローラーのテストがどんなものかを理解するのに役立つはずです。

コントローラのスペックはコントローラのメソッドごとに分かれています。各 example は一つのアクションに割り当てられ、必要に応じてパラメータが渡されます。簡単な example は次のようになります。

[28] http://solnic.eu/2012/02/02/yes-you-should-write-controller-tests.html
[29] https://github.com/rspec/rspec-rails/tree/3-1-maintenance/lib/generators/rspec/scaffold

```
# 保存が完了したらホームページにリダイレクトすること
it "redirects to the home page upon save" do
  post :create, contact: FactoryGirl.attributes_for(:contact)
  expect(response).to redirect_to root_url
end
```

このコードを見ると、これまでに書いてきたスペックと似ている点があることに気付くかもしれません。

- example の説明は 明示的な能動形の言葉 で書かれています。
- example で期待(expect)することは一つだけです。ここで期待しているのは post リクエストが処理されたあとに、リダイレクトがブラウザに返されることです。
- ファクトリでコントローラのメソッドに渡すテストデータが作られています。注目してほしいのは FactoryGirl の `attributes_for` オプションを使っている点です。これを使うと、Ruby オブジェクトではなく値のハッシュが作られます。もちろん、外部ライブラリを呼び出さずに、普通のハッシュを渡すこともできますが、こちらの方が便利なので本書では FactoryGirl を使います。

しかし、新しい内容も二つあります。

- コントローラスペックの基本構文 – HTTP メソッド(`post`)、コントローラメソッド(`:create`)、それにメソッドに渡すパラメータ(任意)が登場しています。この機能は Rack::Test gem で提供されているものです。このあとで API をテストする際にも活躍します。
- 前述した Factory Girl の `attributes_for` メソッド – 難しい話ではありませんが、もう一度触れておきましょう。なぜなら私は昔これを使うのを忘れてデフォルトのファクトリをよく使っていたからです。`attributes_for()` で作られるのは属性のハッシュです。オブジェクトではありませんので、お間違えなく。

整理

まずトップダウンのアプローチで始めましょう。モデルスペックを見ているときに話したように、スペックを次のように考えるとわかりやすいです。つまり、スペックを Ruby のクラスがしなければならないことのアウトラインだと考えるのです。ではサンプルアプリケーションの連絡先コントローラのスペックから始めましょう(繰り返しますが、認証の件はしばらく無視します)。

spec/controllers/contacts_controller_spec.rb

```
1  require 'rails_helper'
2
3  describe ContactsController do
4
5    describe 'GET #index' do
6      # params[:letter] がある場合
7      context 'with params[:letter]' do
8        # 指定された文字で始まる連絡先を配列にまとめること
9        it "populates an array of contacts starting with the letter"
10       # :index テンプレートを表示すること
11       it "renders the :index template"
12     end
13
14     # params[:letter] がない場合
15     context 'without params[:letter]' do
16       # 全ての連絡先を配列にまとめること
```

```
17        it "populates an array of all contacts"
18        # :index テンプレートを表示すること
19        it "renders the :index template"
20      end
21    end
22
23    describe 'GET #show' do
24      # @contact に要求された連絡先を割り当てること
25      it "assigns the requested contact to @contact"
26      # :show テンプレートを表示すること
27      it "renders the :show template"
28    end
29
30    describe 'GET #new' do
31      # @contact に新しい連絡先を割り当てること
32      it "assigns a new Contact to @contact"
33      # :new テンプレートを表示すること
34      it "renders the :new template"
35    end
36
37    describe 'GET #edit' do
38      # @contact に要求された連絡先を割り当てること
39      it "assigns the requested contact to @contact"
40      # :edit テンプレートを表示すること
41      it "renders the :edit template"
42    end
43
44    describe "POST #create" do
45      # 有効な属性の場合
46      context "with valid attributes" do
47        # データベースに新しい連絡先を保存すること
48        it "saves the new contact in the database"
49        # contacts#show にリダイレクトすること
50        it "redirects to contacts#show"
51      end
52
53      # 無効な属性の場合
54      context "with invalid attributes" do
55        # データベースに新しい連絡先を保存しないこと
56        it "does not save the new contact in the database"
57        # :new テンプレートを再表示すること
58        it "re-renders the :new template"
59      end
60    end
61
62    describe 'PATCH #update' do
63      # 有効な属性の場合
64      context "with valid attributes" do
65        # データベースの連絡先を更新すること
```

```
66        it "updates the contact in the database"
67        # 更新した連絡先のページへリダイレクトすること
68        it "redirects to the contact"
69      end
70
71      # 無効な属性の場合
72      context "with invalid attributes" do
73        # 連絡先を更新しないこと
74        it "does not update the contact"
75        # :edit テンプレートを再表示すること
76        it "re-renders the :edit template"
77      end
78    end
79
80    describe 'DELETE #destroy' do
81      # データベースから連絡先を削除すること
82      it "deletes the contact from the database"
83      # contacts#index にリダイレクトすること
84      it "redirects to contacts#index"
85    end
86  end
```

モデルスペックと同様に、RSpec の describe ブロックと context ブロックを使って example をきれいな階層構造に整理しました。この階層関係はコントローラのアクションとこれからテストするコンテキストに基づいています。ここでのコンテキストとはすなわち、正常系（メソッドが有効な属性を受け取る場合）と異常系（メソッドが無効、または不完全な属性を受け取る場合）です。

テストデータをセットアップする

モデルスペックと同じく、コントローラスペックにもデータが必要です。ここでも再度ファクトリを使います。テストデータの作成に慣れてしまえば、もっと効率の良い方法に変更することもできます。しかし、本書の目的を考えると（そしてアプリケーションの大きさも考えると）ファクトリで十分です。

次のコードは作成済みの連絡先ファクトリです。ここに 無効な 連絡先を作成する子ファクトリを追加しましょう。

spec/factories/contacts.rb

```
1   FactoryGirl.define do
2     factory :contact do
3       firstname { Faker::Name.first_name }
4       lastname { Faker::Name.last_name }
5       email { Faker::Internet.email }
6
7       after(:build) do |contact|
8         [:home_phone, :work_phone, :mobile_phone].each do |phone|
9           contact.phones << FactoryGirl.build(:phone,
10            phone_type: phone, contact: contact)
11        end
12      end
13
```

```
14      factory :invalid_contact do
15        firstname nil
16      end
17    end
18  end
```

前章でファクトリの継承機能を使ったことを覚えていますか？前章では親の :phone ファクトリから :home_phone 、:office_phone 、:mobile_phone を作りました。これと同じテクニックを使ってベースとなる :contact ファクトリから :invalid_contact を作ることができます。このテクニックを使えば特定の属性（このケースでは firstname ）を独自の値に変更できます。その他の属性は全て元の :contact ファクトリから引き継がれます。

GET リクエストをテストする

標準的な CRUD 系の Rails コントローラには 4 つの GET 系メソッドがあります。つまり、*index* 、*show* 、*new* 、*edit* です。これらのメソッドは通常最も簡単なテストになります。最初はシンプルな方が良いので、*show* メソッドから始めてみましょう。

spec/controllers/contacts_controller_spec.rb

```
1   describe 'GET #show' do
2     # @contact に要求された連絡先を割り当てること
3     it "assigns the requested contact to @contact" do
4       contact = create(:contact)
5       get :show, id: contact
6       expect(assigns(:contact)).to eq contact
7     end
8
9     # :show テンプレートを表示すること
10    it "renders the :show template" do
11      contact = create(:contact)
12      get :show, id: contact
13      expect(response).to render_template :show
14    end
15  end
```

このスペックを分解してみましょう。ここでは二つのことをチェックしています。一つ目はコントローラのメソッドが保存済みの連絡先を見つけ、それを特定のインスタンス変数に正しく割り当てていることです。これをチェックするために assigns() メソッドを利用しています。このメソッドを使って @contact に *assign* されている値が期待通りかどうかをチェックします。

二つ目のエクスペクテーションは説明不要かもしれません。これもきれいで読みやすい RSpec の構文のおかげです（訳注: 前述の RSpec のコードは *expect response to render template show* と自然な英文のように読める）。つまり、ここでの期待値はコントローラからブラウザに返されるレスポンスが show.html.erb テンプレートを使って表示されることです。

この二つのシンプルなエクスペクテーションから、次のようなコントローラテストのキーコンセプトを説明することができます。

- コントローラとやりとりするための基礎的な DSL は次のようになっています。まず、各 HTTP メソッドはそれぞれ対応するメソッドを持っています（この場合であれば get ）。次に、そのメソッドはコントローラのメソッド名をシンボル（ここでは :show ）として受け取り、さらに続けてパラメータ（id: contact ）を受け取ります。

- コントローラのメソッドでインスタンス化された変数は `assigns(: 変数名)` で評価されます。
- コントローラのメソッドから返された最終的な成果は `response` を使って評価されます。

では次に、さっきよりも少しややこしい *index* メソッドを見てみましょう。

spec/controllers/contacts_controller_spec.rb

```
1  describe 'GET #index' do
2    # params[:letter] がある場合
3    context 'with params[:letter]' do
4      # 指定された文字で始まる連絡先を配列にまとめること
5      it "populates an array of contacts starting with the letter" do
6        smith = create(:contact, lastname: 'Smith')
7        jones = create(:contact, lastname: 'Jones')
8        get :index, letter: 'S'
9        expect(assigns(:contacts)).to match_array([smith])
10     end
11
12     # :index テンプレートを表示すること
13     it "renders the :index template" do
14       get :index, letter: 'S'
15       expect(response).to render_template :index
16     end
17   end
18
19   # params[:letter] がない場合
20   context 'without params[:letter]' do
21     # 全ての連絡先を配列にまとめること
22     it "populates an array of all contacts" do
23       smith = create(:contact, lastname: 'Smith')
24       jones = create(:contact, lastname: 'Jones')
25       get :index
26       expect(assigns(:contacts)).to match_array([smith, jones])
27     end
28
29     # :index テンプレートを表示すること
30     it "renders the :index template" do
31       get :index
32       expect(response).to render_template :index
33     end
34   end
35 end
```

最初のコンテキスト（context）から順に分解していきましょう。ここでは二つのことをチェックしています。最初にチェックするのは、頭文字検索に合致した連絡先の配列が作られ、それが `@contacts` に割り当てられていることです。ここでもまた `assigns()` メソッドが登場します。そして RSpec の `match_array` マッチャを使って、コレクション（つまり `@contacts` に *assign* された値）が期待する値になっていることをチェックします。このケースでは、example の中で作られた `smith` が一つだけ含まれている配列を探します。`jones` ではありません。二つ目の example では `response` を経由して `index.html.erb` テンプレートが表示されていることを確認しています。

match_array は配列の中身を確認しますが、順番は確認しません。順番が重要になる場合は、代わりに eq マッチャを使ってください。

二つ目の context も同じ構成になっています。唯一の違いは文字をパラメータとしてメソッドに渡していない点です。結果として、最初のエクスペクテーションでは作成された 両方の連絡先 が返されることになります。ええ、コードの重複が発生しているのはわかっています。とりあえず今はこう書いてください。タイプすれば構文を覚えるのに役立ちます。コードの重複はすぐあとできれいにするので大丈夫です。

残りの GET メソッドは *new* と *edit* だけです。これも追加してみましょう。

spec/controllers/contacts_controller_spec.rb

```
describe 'GET #new' do
  # @contact に新しい連絡先を割り当てること
  it "assigns a new Contact to @contact" do
    get :new
    expect(assigns(:contact)).to be_a_new(Contact)
  end

  # :new テンプレートを表示すること
  it "renders the :new template" do
    get :new
    expect(response).to render_template :new
  end
end

describe 'GET #edit' do
  # @contact に要求された連絡先を割り当てること
  it "assigns the requested contact to @contact" do
    contact = create(:contact)
    get :edit, id: contact
    expect(assigns(:contact)).to eq contact
  end

  # :edit テンプレートを表示すること
  it "renders the :edit template" do
    contact = create(:contact)
    get :edit, id: contact
    expect(response).to render_template :edit
  end
end
```

これらの example を上から下まで読んでみてください。ご覧の通りです。あなたはもう典型的な GET 系メソッドのテスト方法を理解しているので、標準的な手法で大半の GET 系メソッドをテストできるはずです。

POST リクエストをテストする

今度はコントローラの *create* メソッドをテストする番です。RESTful なアプリケーションでは *create* メソッドは POST 経由でアクセスされます。GET メソッドと異なる大事な点は、GET メソッドで渡していた

:id ではなく、params[:contact] に相当するものを渡す必要があることです。params[:contact] はユーザーが新しい連絡先を作成するために入力したフォームの値です。前述した通り、ここでは Factory Girl の attributes_for() メソッドを使って連絡先の属性をハッシュに変換し、そのハッシュをコントローラに渡します。基本的となるやり方は次のようになります。

```
# post#create で〜すること
it "does something upon post#create" do
  post :create, contact: attributes_for(:contact)
end
```

このやり方に従うと、対象となるメソッドのスペックは次のようになります。最初は有効な属性が渡された場合です。

spec/controllers/contacts_controller_spec.rb

```
1  describe "POST #create" do
2    before :each do
3      @phones = [
4        attributes_for(:phone),
5        attributes_for(:phone),
6        attributes_for(:phone)
7      ]
8    end
9
10   # 有効な属性の場合
11   context "with valid attributes" do
12     # データベースに新しい連絡先を保存すること
13     it "saves the new contact in the database" do
14       expect{
15         post :create, contact: attributes_for(:contact,
16           phones_attributes: @phones)
17       }.to change(Contact, :count).by(1)
18     end
19
20     # contacts#show にリダイレクトすること
21     it "redirects to contacts#show" do
22       post :create, contact: attributes_for(:contact,
23         phones_attributes: @phones)
24       expect(response).to redirect_to contact_path(assigns[:contact])
25     end
26   end
```

続いて無効な属性のブロックを見てみます。

spec/controllers/contacts_controller_spec.rb

```
  # 無効な属性の場合
  context "with invalid attributes" do
    # データベースに新しい連絡先を保存しないこと
    it "does not save the new contact in the database" do
      expect{
        post :create,
          contact: attributes_for(:invalid_contact)
      }.not_to change(Contact, :count)
    end

    # :new テンプレートを再表示すること
    it "re-renders the :new template" do
      post :create,
        contact: attributes_for(:invalid_contact)
      expect(response).to render_template :new
    end
  end
end
```

このコードには注目すべき点がいくつかあります。

まず、第 3 章で説明した context ブロックを使っている点に注目してください。覚えているでしょうか？describe と context は交換可能ですが、異なる状態を表現したいときは context を使うのがベストプラクティスだと説明しました。このケースでは一つが有効な属性が渡された状態で、もう一つが無効な属性が渡された状態です。無効な属性を使う example ではこの章の前半で作った :invalid_contact ファクトリを利用しています。

二つ目に、describe ブロックの先頭にある before フックを見てください。Contact モデルにはバリデーションの要件（つまり、三つの電話番号を持たなければいけないこと）があるので、電話番号の属性も忘れずに渡さなければいけません。ここではその実現方法の一つを紹介しています。POST リクエストに渡す電話番号の属性を三つまとめて一つの配列にする方法です。後ほどもっと効率の良い別の方法を説明します。アプリケーションコードの中でこう書くと「臭うコード」と言われるかもしれませんが、とりあえず今はこうしておきましょう。

attributes_for と関連（associations）についてもっと詳しくなりたい人は、Factory Girl の README にあるcustom strategies and custom callbacks[30]を読んでみてください。

最後に、expect の使い方が最初の example のときとわずかに異なる点に注目してください。今回は HTTP リクエスト全体を expect のブロックに渡しています。これはこれまで見てきた expect の使い方よりも少し複雑です。この HTTP リクエストは Proc として渡され、その結果が実行前と実行後の両方で評価されます。こうすると予期した変化が起きたかどうか（この example では起き なかった かどうか、も）をシンプルに判定できます。

お決まりになってきましたが、RSpec の読みやすさはここでも際立ちます。ほら、「*expect* this code to (or to not) something（このコードが〜する、またはしないと期待する）」と読めますよね。この小さな example はオブジェクトが生成され、保存されることを手短にテストしています。このテクニックに慣れてくれば、コントローラやモデルの様々なメソッドをテストする際に大変役立ちます。そして最終的には統合テストのレベルでも活用することができます。

[30] https://github.com/thoughtbot/factory_girl/blob/master/GETTING_STARTED.md#custom-strategies

PATCH リクエストをテストする

コントローラの *update* メソッドに関しては二つの点をチェックする必要があります。一つはメソッドに渡された属性が更新したいモデルに割り当てられることです。もう一つは私たちが意図したとおりにリダイレクトすることです。それでは Rails 4.1 の HTTP メソッド、PATCH を使ってみましょう。

Rails 4.0 より古いバージョンでは PATCH の代わりに PUT を使います。

spec/controllers/contacts_controller_spec.rb

```
describe 'PATCH #update' do
  before :each do
    @contact = create(:contact,
      firstname: 'Lawrence',
      lastname: 'Smith')
  end

  # 有効な属性の場合
  context "valid attributes" do
    # 要求された @contact を取得すること
    it "locates the requested @contact" do
      patch :update, id: @contact, contact: attributes_for(:contact)
      expect(assigns(:contact)).to eq(@contact)
    end

    # @contact の属性を変更すること
    it "changes @contact's attributes" do
      patch :update, id: @contact,
        contact: attributes_for(:contact,
          firstname: 'Larry',
          lastname: 'Smith')
      @contact.reload
      expect(@contact.firstname).to eq('Larry')
      expect(@contact.lastname).to eq('Smith')
    end

    # 更新した連絡先のページへリダイレクトすること
    it "redirects to the updated contact" do
      patch :update, id: @contact, contact: attributes_for(:contact)
      expect(response).to redirect_to @contact
    end
  end

  # ...
end
```

それから一つ前で見た POST の example と同じように、無効な属性が params に渡された場合は同じようになら ない ことをテストしなければなりません。

spec/controllers/contacts_controller_spec.rb

```
describe 'PATCH #update' do
  # ...

  # 無効な属性の場合
  context "with invalid attributes" do
    # 連絡先の属性を変更しないこと
    it "does not change the contact's attributes" do
      patch :update, id: @contact,
        contact: attributes_for(:contact,
          firstname: "Larry", lastname: nil)
      @contact.reload
      expect(@contact.firstname).not_to eq("Larry")
      expect(@contact.lastname).to eq("Smith")
    end

    # edit テンプレートを再表示すること
    it "re-renders the :edit template" do
      patch :update, id: @contact,
        contact: attributes_for(:invalid_contact)
      expect(response).to render_template :edit
    end
  end
end
```

注目すべき点は次の通りです。

- 既存の Contact を更新するので、まず何かを保存しなければいけません。ここでは `before` フックを使い、あとでアクセスできるように保存した Contact を `@contact` に割り当てています。(繰り返しますが、以降の章でもっと良いやり方を紹介します。)
- 二つの example は *update* メソッドで本当にオブジェクトの属性が本当に変更されたかどうか、もしくはされなかったかどうかを検証しています。ここでは `expect{}` Proc は使えません。更新内容が本当に保存されたかどうかをチェックするためには `@contact` の `reload` メソッドを代わりに呼び出す必要があります。それ以外は POST 関連のスペックで使ったパターンとほぼ同じです。

DELETE リクエストをテストする

では最後に *destroy* メソッドをテストしましょう。このテストは比較的単純です。

spec/controllers/contacts_controller_spec.rb

```
describe 'DELETE #destroy' do
  before :each do
    @contact = create(:contact)
  end

  # 連絡先を削除すること
  it "deletes the contact" do
    expect{
      delete :destroy, id: @contact
```

```
10      }.to change(Contact,:count).by(-1)
11    end
12
13    # contacts#index にリダイレクトすること
14    it "redirects to contacts#index" do
15      delete :destroy, id: @contact
16      expect(response).to redirect_to contacts_url
17    end
18  end
```

今ならもう、ここでやっていることは全部予想が付くと思います。最初のエクスペクテーションではコントローラの destroy メソッドが本当にオブジェクトを削除したかどうかをチェックしています（お馴染みの expect{} Proc を使っています）。二つ目のエクスペクテーションでは成功時に index ページへリダイレクトすることを確認しています。

非 CRUD なメソッドをテストする

コントローラに定義されたその他のメソッドのテストは、Rails 標準の RESTful リソースのテストとほとんど変わりません。ContactsController に hide_contact というメソッドがあると仮定して example を作ってみましょう。このメソッドを使うと管理者は実際に連絡先を削除せずに画面上から連絡先を隠すことができます（私は実装しないので、良かったら読者のみなさんが自分で実装してみてください）。

コントローラレベルでテストするならこんな感じになります。

```
1   describe "PATCH hide_contact" do
2     before :each do
3       @contact = create(:contact)
4     end
5
6     # 連絡先を hidden 状態にすること
7     it "marks the contact as hidden" do
8       patch :hide_contact, id: @contact
9       expect(@contact.reload.hidden?).to be_true
10    end
11
12    # contacts#index にリダイレクトすること
13    it "redirects to contacts#index" do
14      patch :hide_contact, id: @contact
15      expect(response).to redirect_to contacts_url
16    end
17  end
```

何をやっているかわかりますか？ここでは PATCH メソッドを使っています。なぜなら既存の連絡先を編集しているからです。それに続く :hide_contact はアクセスすべきコントローラのメソッドを表しています。それ以外は update メソッドのテストとよく似ています。違うところはユーザが入力した属性のハッシュを渡していない点です。この example では boolean 型の hidden? がサーバー側でセットされます。

expect(@contact.reload.hidden?).to be_true はカスタムマッチャで置き換えるのに適しています。この内容は第 7 章で扱います。

もし自分で作ったメソッドが PATCH 以外の HTTP メソッドを使っているなら、それに対応する CRUD 系メソッドのテストを真似してください。

入れ子になったルーティングをテストする

もしあなたのアプリケーションが 入れ子になったルーティング（nested routes）を使っている、すなわち /contacts/34/phones/22 のようなルーティングになっているなら、example にもう少し情報を増やしてやる必要があります。

入れ子になったルーティングの概要を知りたい場合は、Rails Routing from the Outside In[31]を見てください。

もう一つ別の example を仮定してみましょう。たとえば電話番号の実装を入れ子になった属性ではなく、入れ子になったルーティングにしたとします。これはつまり、連絡先のフォームに電話番号の属性を入力するのではなく、独立したコントローラとビューを使って電話番号のデータを集めたり処理したりするということです。*config/routes.rb* のルーティング設定は次のようになるでしょう。

config/routes.rb

```
1 resources :contacts do
2   resources :phones
3 end
```

アプリケーションのルーティングを見ると（コマンドラインから `rake routes` と入力してください）、PhoneController の `:show` メソッドを呼び出すパスは `/contacts/:contact_id/phones/:id` になっています。よって私たちは電話番号の `:id` と、親となる連絡先を指定する `:contact_id` を渡す必要があります。このスペックは次のようになります。

```
1 describe 'GET #show' do
2   # 電話番号用の:show テンプレートを表示すること
3   it "renders the :show template for the phone" do
4     contact = create(:contact)
5     phone = create(:phone, contact: contact)
6     get :show, id: phone, contact_id: contact.id
7     expect(response).to render_template :show
8   end
9 end
```

抑えておくべきキーポイントは、`:parent_id` のような形式で親のルーティングをサーバーに渡す必要がある点です。このケースでは `:contact_id` がそれに該当します。こうすればコントローラは *routes.rb* ファイルで指定されている通りに処理を行います。入れ子になったコントローラのメソッドであれば、どれでもこれと同じテクニックが使えます。

コントローラの非 HTML 出力をテストする

ここまでは HTML を返すコントローラメソッドだけをテストしてきました。もちろん、Rails は一つのコントローラのメソッドから複数の形式でデータを送信することができます。他の形式は HTML 出力に追加することもできますし、HTML 出力と置き換えることもできます。

[31] http://guides.rubyonrails.org/routing.html#nested-resources

引き続き仮想的な example を使ってみましょう。たとえば連絡先を CSV ファイルにエクスポートする必要があるとします。自分のアプリケーションで非 HTML 形式のデータを返したことがある人なら、次のようにデフォルトの HTML をメソッドの引数でオーバーライドする方法を知っているかしれません。

```
link_to 'Export', contacts_path(format: :csv)
```

このとき、コントローラのメソッドには次のようなコードが書かれていることが前提になります。

```
def index
  @contacts = Contact.all

  respond_to do |format|
    format.html # index.html.erb
    format.csv do
      send_data Contact.to_csv(@contacts),
        type: 'text/csv; charset=iso-8859-1; header=present',
          disposition: 'attachment; filename=contacts.csv'
    end
  end
end
```

そしてこれをテストするには、レスポンスのデータ型を検証するのが一番シンプルです。

```
describe 'CSV output' do
  # CSV ファイルを返すこと
  it "returns a CSV file" do
    get :index, format: :csv
    expect(response.headers['Content-Type']).to match 'text/csv'
  end

  # 中身を返すこと
  it 'returns content' do
    create(:contact,
      firstname: 'Aaron',
      lastname: 'Sumner',
      email: 'aaron@sample.com')
    get :index, format: :csv
    expect(response.body).to match 'Aaron Sumner,aaron@sample.com'
  end
end
```

ここで match マッチャを使っていることに注意してください。これは実行結果を正規表現で比較する場合に使用します。

このテストではコントローラが有効な Content-Type で CSV データを返却していることを検証しています。しかし、CSV の中身を実際に 生成する ための仕組みがあれば、つまり *Contact* にクラスメソッドがあれば、(コントローラレイヤではなく)モデルレベルでその機能をテストする方が理想的かもしれません。

```
# カンマ区切りの値を返すこと
it "returns comma separated values" do
  create(:contact,
    firstname: 'Aaron',
    lastname: 'Sumner',
    email: 'aaron@sample.com')
  expect(Contact.to_csv).to match /Aaron Sumner,aaron@sample.com/
end
```

CSV データの生成に関する説明は本書のスコープを超えてしまいます。ここで使ったような CSV 生成の一般的なアプローチについては Railscasts のエピソード 362、Exporting to CSV and Excel[32]を参考にしてください。

さらに、コントローラレベルで JSON や XML の出力結果をテストするのも結構簡単です。

```
# JSON 形式の中身を返すこと
it "returns JSON-formatted content" do
  contact = create(:contact)
  get :index, format: :json
  expect(response.body).to have_content contact.to_json
end
```

API のテストについては第 10 章でさらに詳しく説明します。

まとめ

というわけで、以上がコントローラをテストする方法です。この方法のキーポイントはテストすべき項目を分解し、それから目的の機能が網羅されるまで少しずつテストを積み上げていく点です。

あいにく、コントローラスペックはいつもこんなに単純だとは限りません。ユーザーのログイン機能や手作業で追加した scaffold 以外のコード、もしくは特定の要件を満たすバリデーションを持つモデルなど、ある程度複雑なコードを扱うことも多いと思います。次章ではこうした内容を説明していきます。

演習問題

- 第 4 章を振り返ってください。あなたならどうやって `:invalid_contact` ファクトリをテストしますか?
- 賢明な読者なら example で使ったコントローラの *index* メソッドに設計ミスがあると気付いたかもしれません。それが何かわかりますか?コントローラ内のロジックを減らすためにリファクタリングできますか?ヒントが知りたい方は次の問題を見てください。
- 一つのコントローラメソッドをテストするのに大量のセットアップが必要になると気付いたら、それはコントローラのリファクタリングが必要になっている証拠かもしれません。もしかするとコントローラのメソッド内に、モデルやヘルパーメソッドに移動した方が良いコードが含まれているかもしれません。そんなコードが出てきたら、コードをきれいにする時間を用意してください。相性の悪いコードはモデルに移動させ、そこで必要なスペックを書きましょう。そしてコントローラのスペックをシンプルにしてください。それでもスペックはパスするはずです。

[32] http://railscasts.com/episodes/362-exporting-csv-and-excel

6. 高度なコントローラスペック

もうコントローラテストの基礎は理解できたと思います。それではコントローラが期待通りに動くことを RSpec で確認する方法を見ていきましょう。ただし今回は、少々退屈だった CRUD のスペックに認証と認可のレイヤーを追加していきます。もう少し詳しく言うと次のようになります。

- 最初にもっと複雑なスペックを準備します。
- 次にコントローラを使った認証、つまりログイン機能のテストを説明します。
- 続いて認可、つまりロール（roles）をテストします。これもコントローラを使って実現します。
- また、コントローラスペックがちゃんとアプリケーションの要件を処理できているかどうか確認するテクニックも紹介します。

本章の完成後のコードを見たい場合はサンプルソースの *06_advanced_controllers* ブランチをチェックアウトしてください。コマンドラインを使う場合は次のようにタイプします。

```
git checkout -b 06_advanced_controllers origin/06_advanced_controllers
```

本章を読みながら一緒にコードも書いていく場合は、前の章のブランチから始めてください。

```
git checkout -b 05_controller_basics origin/05_controller_basics
```

その他、詳細については第 1 章を読んでください。

準備する

前章では ContactsController の `before_action` をコメントアウトして認証機能を無効にしていました。このコメントを外し、再び認証機能を有効にしましょう。それから RSpec を実行し、スペックがいくつ壊れるか確認してください！

```
Finished in 1.11 seconds (files took 3.15 seconds to load)
32 examples, 13 failures
```

先へ進むためにはまず、コントローラスペックの中で認証の処理をシミュレートしなければなりません。特にここではユーザーがログインしているのかそうでないのか、そしてログインしているユーザーのロールは何か、というテストに取り組む必要があります。ちなみに、ユーザーにならなければ連絡先の追加や編集ができないことと、管理者ユーザーにならなければ他のユーザーを追加できないことを思い出してください。そこでコントローラの基本的な仕組みを使って認証機能を処理し、その設定をコントローラレベルでテストします。

管理者ロールとユーザーロールをテストする

今回は今までとは異なるアプローチでコントローラスペック全体を見ていきます。すなわち、ロールごとにスペックを見ていきます。用意されているロールはゲスト、ユーザー、管理者の三つです。では現在失敗中のスペックを直していきましょう。このアプリケーションではユーザーと管理者（`:admin` フラグが有効になっているユーザー）は連絡先に対して全く同じ権限を持っています。言い換えるとつまり、正しいアカウントでログインしているユーザーなら誰でも連絡先を作成、編集、削除できます。

まず、ユーザー用の新しいファクトリを作りましょう。

spec/factories/users.rb

```
FactoryGirl.define do
  factory :user do
    email { Faker::Internet.email }
    password 'secret'
    password_confirmation 'secret'

    factory :admin do
      admin true
    end
  end
end
```

こうすれば create(:user)（もしくは省略化しないなら FactoryGirl.create(:user)）というコードで新しいユーザーオブジェクトを簡単に作ることができます。また、admin という子ファクトリも用意したので、管理者ロールを持つユーザーも作れます。この場合、admin フラグが true に設定されます。

ではコントローラスペックに戻ります。ファクトリを使って管理者がアクセスした場合のテストをしましょう。最初の数行を注意して見てください。

spec/controllers/contacts_controller_spec.rb

```
describe "administrator access" do
  before :each do
    user = create(:admin)
    session[:user_id] = user.id
  end

  describe 'GET #index' do
    # params[:letter] がある場合
    context 'with params[:letter]' do
      # 指定された文字で始まる連絡先を配列に集めること
      it "populates an array of contacts starting with the letter" do
        smith = create(:contact, lastname: 'Smith')
        jones = create(:contact, lastname: 'Jones')
        get :index, letter: 'S'
        expect(assigns(:contacts)).to match_array([smith])
      end

      # :index テンプレートを表示すること
      it "renders the :index template" do
        get :index, letter: 'S'
        expect(response).to render_template :index
      end
    end

    # params[:letter] がない場合
    context 'without params[:letter]' do
      # すべての連絡先を配列に集めること
      it "populates an array of all contacts" do
        smith = create(:contact, lastname: 'Smith')
```

```
30          jones = create(:contact, lastname: 'Jones')
31          get :index
32          expect(assigns(:contacts)).to match_array([smith, jones])
33        end
34
35        # :index テンプレートを表示すること
36        it "renders the :index template" do
37          get :index
38          expect(response).to render_template :index
39        end
40      end
41    end
42
43    describe 'GET #show' do
44      # @contact に要求された連絡先を割り当てること
45      it "assigns the requested contact to @contact" do
46        contact = create(:contact)
47        get :show, id: contact
48        expect(assigns(:contact)).to eq contact
49      end
50
51      # :show テンプレートを表示すること
52      it "renders the :show template" do
53        contact = create(:contact)
54        get :show, id: contact
55        expect(response).to render_template :show
56      end
57    end
58
59    # などなど ...
60  end
```

ここでは何が起きているのでしょうか?実はとてもシンプルなことです。まず既存の全 example を新しい `describe` ブロックで囲みます。それからその内側に `before` ブロックを追加し、管理者のログインをシミュレートしています。これを実現するにはまず、先ほど作った `:admin` ファクトリを使って管理者オブジェクトをインスタンス化し、それからその管理者をセッションの値に割り当てます。

このケースではこれで終わりです。有効なログインがシミュレートできれば、コントローラスペックは再びパスします。

簡潔に終わらせたいので、非管理者用のスペックは一部しか載せません。完全な実装を見たい場合は、本章のサンプルコードを見てください。ただし、`before :each` ブロックを除き、エクスペクテーションは全く同じです。

spec/controllers/contacts_controller_spec.rb

```
describe "user access" do
  before :each do
    user = create(:user)
    session[:user_id] = user.id
  end

  # スペックは管理者と同じ
```

ええ、これはとても冗長なコードです。でもご心配なく。RSpec にはこうした問題に対処する機能があります。それは次の章で説明します。とりあえず今は、テストすべき別のユースケースに焦点を移しましょう。

ゲストロールをテストする

ゲストロール、すなわちまだログインしていないユーザーの存在は見落とされることが多いです。しかしこのサンプルアプリケーションのように一般向けに公開されるアプリケーションでは、ゲストロールが一番よく使われるロールかもしれません！なので、ゲストロールもスペックに追加しましょう。他のロールのスペックとは異なり、少し変更を加える必要があります。さっきのようにコピーアンドペーストでは作れませんが、とても簡単に書くことができます。

spec/controllers/contacts_controller_spec.rb

```
describe "guest access" do
  # GET #index と GET #show の example は管理者と
  # ユーザーの example と同じ

  describe 'GET #new' do
    # ログインを要求すること
    it "requires login" do
      get :new
      expect(response).to redirect_to login_url
    end
  end

  describe 'GET #edit' do
    # ログインを要求すること
    it "requires login" do
      contact = create(:contact)
      get :edit, id: contact
      expect(response).to redirect_to login_url
    end
  end

  describe "POST #create" do
    # ログインを要求すること
    it "requires login" do
      post :create, id: create(:contact),
        contact: attributes_for(:contact)
      expect(response).to redirect_to login_url
```

```
28      end
29    end
30
31    describe 'PATCH #update' do
32      # ログインを要求すること
33      it "requires login" do
34        put :update, id: create(:contact),
35          contact: attributes_for(:contact)
36        expect(response).to redirect_to login_url
37      end
38    end
39
40    describe 'DELETE #destroy' do
41      # ログインを要求すること
42      it "requires login" do
43        delete :destroy, id: create(:contact)
44        expect(response).to redirect_to login_url
45      end
46    end
47  end
```

new のブロックに至るまでは新しい内容は何もありませんでした。コントローラの最初にあった before_action メソッドはアクセスされるとログインを要求します。なので、今回テストしなければならないのは、ゲストがコントローラの処理を実行 できない ことです。代わりに login_url にリダイレクトされなければいけません。login_url はゲストユーザーがログインを要求される画面です。ご覧の通り、ログインを要求するメソッドであれば どれでも このテクニックが使えます。

もう一度スペックを実行し、スペックがパスすることを確かめてください。実験的にコントローラの before_action :authenticate の行をもう一度コメントアウトし、スペックを実行してみてください。何が起きるでしょうか?また、expect(response).to redirect_to login_url を expect(response).not_to redirect_to login_url に変えたり、login_url を違うパスに変更したりするのも良いでしょう。

このようにわざとテストを壊してみるのは良いアイデアです。こうすれば誤判定を見逃す可能性が少なくなります。

ロールに応じた認可機能をテストする

最後に、別のコントローラで認可機能をテストする方法を考えなければいけません。認可とはつまり、ログインしたユーザーにどんな操作を許可するか、ということです。このサンプルアプリケーションでは管理者だけが新しいユーザーを追加できます。普通のユーザー(:admin フラグが無効になっているユーザー)はアクセスを拒否されます。

テストのやり方はここまで見てきた内容と基本的に同じです。before :each ブロック内でシミュレートしたいユーザーをセットアップします。続いて user_id をセッション変数に割り当ててログインをシミュレートします。それからスペックを書きます。ただし、今回ユーザーはログイン画面ではなくルート URL にリダイレクトされます。これは app/controllers/application_controller.rb で定義されている振る舞いです。このシナリオを検証するスペックは次のようになります。

spec/controllers/users_controller_spec.rb

```
describe 'user access' do
  before :each do
    @user = create(:user)
    session[:user_id] = @user.id
  end

  describe 'GET #index' do
    # ユーザーを @users に集めること
    it "collects users into @users" do
      user = create(:user)
      get :index
      expect(assigns(:users)).to match_array [@user,user]
    end

    # :index テンプレートを表示すること
    it "renders the :index template" do
      get :index
      expect(response).to render_template :index
    end
  end

  # GET #new はアクセスを拒否すること
  it "GET #new denies access" do
    get :new
    expect(response).to redirect_to root_url
  end

  # POST #create はアクセスを拒否すること
  it "POST #create denies access" do
    post :create, user: attributes_for(:user)
    expect(response).to redirect_to root_url
  end
end
```

まとめ

本章と前章では非常にたくさんの内容を説明しました。しかし、コントローラレベルで網羅的なテストを書けば、実際アプリケーションの機能を数多くテストできます。これが一番重要なことです。

前の章で説明したように、私はいつもこれほど徹底的に自分のコントローラをテストしているわけではありません。私がコントローラスペックを書くときはコントローラを個別に見て、効果が高そうなものを選ぶことが多いです（よくあるのは自動生成ではないコードです）。とはいえ、RSpec が自動生成した example を見れば、コントローラレベルでもできること、そしてすべきことがあるとわかるはずです。

そしてしっかりテストされたコントローラがあれば、アプリケーション全体のテスト網羅率をうまく向上させることできます。現時点であなたは RSpec や Factory Girl、その他のヘルパーを使う上でのグッドプラクティスやテクニックを徐々に身につけてきているはずです。こうしたテクニックがあれば、テストによってコードの信頼性をさらに向上させることができます。

とはいえ、もっとうまくやることもできます。次の章ではもう一度このスペックを全体的に眺め、ヘルパーメソッドや shared examples を使ってスペックをきれいにします。

演習問題

適当なコントローラをあなたのアプリケーションから選び、どのメソッドがどのユーザーからアクセスできるかを表に書いてください。たとえば、プレミアムコンテンツを扱うブログアプリケーションがあったとします。そのコンテンツにアクセスするにはメンバーにならなくてはいけません。しかし、ユーザーの見てみたいという気持ちを刺激するために、タイトルのリストだけはユーザーに見せます。実際のユーザーはロールによってアクセスできるレベルが異なります。アプリケーションに Posts（記事）Controller があったと仮定すると、次のような権限が割り振られるかもしれません。

ロール	Index	Show	Create	Update	Destroy
管理者	すべて	すべて	すべて	すべて	すべて
編集者	すべて	すべて	すべて	すべて	すべて
作者	すべて	すべて	すべて	すべて	なし
メンバー	すべて	すべて	なし	なし	なし
ゲスト	すべて	なし	なし	なし	なし

この表があれば、テストすべき様々なシナリオを洗い出すのに役立ちます。この例では *new* と *create* を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* のフォームを表示させても大して意味がないからです）。*edit* と *update* も同様です。一方、*index* と *show* は別々に分けています。この表とあなたのアプリケーションの要件を比べてみてください。あなたのアプリケーションの認証と認可の要件はどうなっているでしょうか？また、あなたが変更すべき項目はどれになるでしょうか？

7. コントローラスペックのクリーンアップ

もしあなたがここまで学んできたことを自分のコードに適用すれば、堅牢なテストスイートを構築できるはずです。しかし、前章では重複したコードをたくさん書いてしまいました。さらにテストも壊れやすくなっています。たとえば、未認証のリクエストを `root_path` へリダイレクトさせる代わりに、専用の `denied_path` を作ったらどうなるでしょうか？私たちはきれいにしなければならないスペックを大量に作ってしまいました。

アプリケーションコードと同じく、スペックをきれいにする機会も作るべきです。本章では重複をなくし、スペックを壊れにくくする三つの方法を説明します。しかも可読性は犠牲にしません。

- はじめに、複数の `describe` ブロックや `context` ブロックで同じ example を共有します。
- 次にヘルパーマクロを使って重複をさらに減らします。
- 最後に、仕上げとしてカスタム RSpec マッチャ（custom RSpec matchers）を作ります。

本章の完成後のコードを見たい場合はサンプルソースの *07_controller_cleanup* ブランチをチェックアウトしてください。コマンドラインを使う場合は次のようにタイプします。

```
git checkout -b 07_controller_cleanup origin/07_controller_cleanup
```

本章を読みながら一緒にコードも書いていく場合は、前の章のブランチから始めてください。

```
git checkout -b 06_advanced_controllers origin/06_advanced_controllers
```

その他、詳細については第 1 章を読んでください。

Shared examples

第 1 章を振り返ってください。第 1 章では私の一般的なテストのアプローチについて説明しました。その中で私は、100 パーセント DRY であることよりも、読みやすいスペックになっていることの方が絶対重要である、と述べました。私のその意見は変わりません。とはいえ、`contacts_controller_spec.rb` を見てみると、ちょっと限界が来ています。今の状態だと多くの example が 2 回ずつ登場しています（一つは管理者用で、もう一つは普通のユーザー用です）。さらに example のいくつかは 3 回も 使われています。なぜなら、ゲストと管理者と普通のユーザーは全員 `:index` と `:show` メソッドにアクセスできるからです。これだとコードの量が増え、読みやすさや将来の保守性にも悪影響を及ぼします。

RSpec では *shared examples* という便利な機能を使ってこうした重複をなくすことができます。shared examples のセットアップはとても簡単です。まず次のように example のブロックを作ります。

spec/controllers/contacts_controller_spec.rb

```
1  shared_examples 'public access to contacts' do
2    before :each do
3      @contact = create(:contact,
4        firstname: 'Lawrence',
5        lastname: 'Smith'
6      )
7    end
8
9    describe 'GET #index' do
```

```
10      # 連絡先を配列にまとめること
11      it "populates an array of contacts" do
12        get :index
13        expect(assigns(:contacts)).to match_array [@contact]
14      end
15
16      # :index テンプレートを表示すること
17      it "renders the :index template" do
18        get :index
19        expect(response).to render_template :index
20      end
21    end
22
23    describe 'GET #show' do
24      # @contact に要求された連絡先を割り当てること
25      it "assigns the requested contact to @contact" do
26        get :show, id: @contact
27        expect(assigns(:contact)).to eq @contact
28      end
29
30      # :show テンプレートを表示すること
31      it "renders the :show template" do
32        get :show, id: @contact
33        expect(response).to render_template :show
34      end
35    end
36  end
```

それから、これらの example を使いたい describe ブロック、もしくは context ブロックにこの内容をインクルードしてください。たとえばこんな感じです（実際のコードは取り除いて簡潔にしてあります。前後のコードを確認したい場合はサンプルコードを参照してください）。

spec/controllers/contacts_controller_spec.rb

```
1  describe "guest access" do
2    it_behaves_like "public access to contacts"
3
4    # ゲストがアクセスした場合のスペックが続く...
5  end
```

続いて 2 つ目の shared example を作っていきましょう。今回は連絡先の管理機能をカバーする shared example を作成します。この機能は管理者ロールとユーザロールの両方で共有（shared）されます。新しく shared_examples のブロックを作り、共通する機能を書いていきましょう。ここでは具体的なテストコードの記述は省略しますが、実際の spec では当然中身を書く必要があります。ただし、それぞれのエクスペクテーションは一つ前の章で説明したものと同じです。ここでやっていることは重複をなくすために、単にコードを整理しているだけです。

この変更によって、contacts_controller_spec.rb はずっときれいになりました。このアウトラインを見れば明らかです。（訳注: アウトラインを確認しやすくするため、各 example の日本語訳は省略します。）

spec/controllers/contacts_controller_spec.rb

```
require 'rails_helper'

describe ContactsController do
  shared_examples 'public access to contacts' do
    describe 'GET #index' do
      context 'with params[:letter]' do
        it "populates an array of contacts starting with the letter"
        it "renders the :index template"
      end

      context 'without params[:letter]' do
        it "populates an array of all contacts"
        it "renders the :index template"
      end
    end

    describe 'GET #show' do
      it "assigns the requested contact to @contact"
      it "renders the :show template"
    end
  end

  shared_examples 'full access to contacts' do
    describe 'GET #new' do
      it "assigns a new Contact to @contact"
      it "assigns a home, office, and mobile phone to the new contact"
      it "renders the :new template"
    end

    describe 'GET #edit' do
      it "assigns the requested contact to @contact"
      it "renders the :edit template"
    end

    describe "POST #create" do
      context "with valid attributes" do
        it "saves the new contact in the database"
        it "redirects to contacts#show"
      end

      context "with invalid attributes" do
        it "does not save the new contact in the database"
        it "re-renders the :new template"
      end
    end

    describe 'PATCH #update' do
      context "valid attributes" do
```

```
49          it "locates the requested @contact"
50          it "changes the contact's attributes"
51          it "redirects to the updated contact"
52        end
53
54        context "invalid attributes" do
55          it "locates the requested @contact"
56          it "does not change the contact's attributes"
57          it "re-renders the edit method"
58        end
59      end
60
61      describe 'DELETE #destroy' do
62        it "deletes the contact"
63        it "redirects to contacts#index"
64      end
65    end
66
67    describe "administrator access" do
68      it_behaves_like 'public access to contacts'
69      it_behaves_like 'full access to contacts'
70    end
71
72    describe "user access" do
73      it_behaves_like 'public access to contacts'
74      it_behaves_like 'full access to contacts'
75    end
76
77    describe "guest access" do
78      it_behaves_like 'public access to contacts'
79
80      describe 'GET #new' do
81        it "requires login"
82      end
83
84      describe 'GET #edit' do
85        it "requires login"
86      end
87
88      describe "POST #create" do
89        it "requires login"
90      end
91
92      describe 'PATCH #update' do
93        it "requires login"
94      end
95
96      describe 'DELETE #destroy' do
97        it "requires login"
```

```
98      end
99    end
100 end
```

そして、bin/rspec controllers/contacts_controller_spec.rb を実行すると、ドキュメント形式のアウトプットも同じように読みやすくなります。

```
ContactsController
  administrator access
    behaves like public access to contacts
      GET #index
        with params[:letter]
          populates an array of contacts starting with the letter
          renders the :index template
        without params[:letter]
          populates an array of all contacts
          renders the :index template
      GET #show
        assigns the requested contact to @contact
        renders the :show template
    behaves like full access to contacts
      GET #new
        assigns a new Contact to @contact
        assigns a home, office, and mobile phone to the new contact
        renders the :new template
      GET #edit
        assigns the requested contact to @contact
        renders the :edit template
      POST #create
        with valid attributes
          saves the new contact in the database
          redirects to contacts#show
        with invalid attributes
          does not save the new contact in the database
          re-renders the :new template
      PATCH #update
        valid attributes
          locates the requested @contact
          changes the contact's attributes
          redirects to the updated contact
        invalid attributes
          locates the requested @contact
          does not change the contact's attributes
          re-renders the edit method
      DELETE #destroy
        deletes the contact
        redirects to contacts#index
  user access
    behaves like public access to contacts
```

```
    GET #index
      with params[:letter]
        populates an array of contacts starting with the letter
        renders the :index template
      without params[:letter]
        populates an array of all contacts
        renders the :index template
    GET #show
      assigns the requested contact to @contact
      renders the :show template
  behaves like full access to contacts
    GET #new
      assigns a new Contact to @contact
      assigns a home, office, and mobile phone to the new contact
      renders the :new template
    GET #edit
      assigns the requested contact to @contact
      renders the :edit template
    POST #create
      with valid attributes
        saves the new contact in the database
        redirects to contacts#show
      with invalid attributes
        does not save the new contact in the database
        re-renders the :new template
    PATCH #update
      valid attributes
        locates the requested @contact
        changes the contact's attributes
        redirects to the updated contact
      invalid attributes
        locates the requested @contact
        does not change the contact's attributes
        re-renders the edit method
    DELETE #destroy
      deletes the contact
      redirects to contacts#index
guest access
  behaves like public access to contacts
    GET #index
      with params[:letter]
        populates an array of contacts starting with the letter
        renders the :index template
      without params[:letter]
        populates an array of all contacts
        renders the :index template
    GET #show
      assigns the requested contact to @contact
      renders the :show template
```

```
    GET #new
      requires login
    GET #edit
      requires login
    POST #create
      requires login
    PATCH #update
      requires login
    DELETE #destroy
      requires login
```

ヘルパーマクロを作成する

さて、今度はコントローラで複数回使っている別のコードに注目してみましょう。ログイン中のユーザーができること、またはできないことをテストするときは、ログインをシミュレートするために毎回ファクトリで作ったユーザーの `:id` をセッションに入れています。この機能を RSpec の マクロ(macro) に移動させましょう。マクロを利用すれば簡単にテストスイート全体で使われるメソッドを定義できます。マクロは慣習として *spec/support* ディレクトリ内に作られ、RSpec の設定でモジュールとしてインクルードされます。

まず、セッション変数に値を入れるマクロは次のようになります。

spec/support/login_macros.rb

```
1  module LoginMacros
2    def set_user_session(user)
3      session[:user_id] = user.id
4    end
5  end
```

とても単純な Ruby のモジュールとメソッドです。このメソッドは `user` オブジェクトを受け取り、`session[:user_id]` にユーザーの `:id` を割り当てます。

この新しいヘルパーをスペックで使う前に、RSpec にこのヘルパーの居場所を教えなければいけません。次のように、*spec/rails_helper.rb* の `RSpec.configure` ブロックの内部に `config.include LoginMacros` という一行を追加してください。

spec/rails_helper.rb

```
1  Dir[Rails.root.join("spec/support/**/*.rb")].each {|f| require f}
2
3  RSpec.configure do |config|
4    # 他の RSpec の設定は省略します...
5
6    config.include LoginMacros
7  end
```

Devise のような認証ライブラリは同じような機能を持っています。もしあなたがそういったライブラリを使っているのであれば、ドキュメントの手順を読んでテストスイートにその機能を組みこんでください。

設定が済んだら、コントローラのスペックでこのメソッドを使ってみましょう。`before` ブロックの中で新しい管理者ユーザーを作成し、セッションにそのユーザーをセットします。これがすべて一行で書けるのです。

spec/controllers/contacts_controller_spec.rb

```
describe "administrator access" do
  before :each do
    set_user_session create(:admin)
  end

  it_behaves_like "public access to contacts"
  it_behaves_like "full access to contacts"
end
```

たった一行しかないヘルパーメソッドを完全に切り離して定義するのは馬鹿げているように見えるかもしれません。しかし現実には認証システム全体が変更され、ログインをシミュレートする方法も変えなければいけない、という事態は起こりえます。しかし、こういった方法でログインをシミュレートしていれば、変更するのは一箇所だけで済むのです。

次章で統合テストを書くようになったら、このテクニックのおかげで 複数 行のコードを再利用できます。ちなみにそのマクロはユーザーログインのステップをシミュレートするときに使います。

カスタム RSpec マッチャを使う

ここまでは RSpec 標準のマッチャが大変役に立ってきました。正直に話すと、標準のマッチャだけでアプリケーション全体をテストできるかもしれません。(私は実際そうしたこともあります。)しかし、前節でヘルパーマクロを作成したときと同様、カスタムマッチャを導入すると長い目で見たときにテストの可読性が上がるかもしれません。今見ているアドレス帳であれば、ログインフォームのルーティングや、許可されていない操作をしようとした場合のリダイレクト先を変更したりするとどうなるでしょうか？今のままだと新しいルーティングに合わせて多くの example を変更しなければなりません。しかしもう一つの選択肢として、カスタムマッチャを作って一箇所の変更だけで終わらせる方法もあります。カスタムマッチャを *spec/support/matchers* に配置すると(1 ファイルにつき 1 マッチャです)、RSpec は自動的にそれを読み込んで、カスタムマッチャがスペックで使えるようになります。

RSpec のデフォルト設定は自動的にそれを読み込んで、カスタムマッチャがスペックで使えるようになります。

以下に例を載せます。

spec/support/matchers/require_login.rb

```
RSpec::Matchers.define :require_login do |expected|
  match do |actual|
    expect(actual).to redirect_to \
      Rails.application.routes.url_helpers.login_path
  end

  failure_message do |actual|
    "expected to require login to access the method"
  end

  failure_message_when_negated do |actual|
    "expected not to require login to access the method"
  end

  description do
    "redirect to the login form"
```

```
17    end
18  end
```

このコードをさらっと見ていきましょう。match ブロックに書くのは私たちが 期待(expect)する ことです。基本的にはスペック内で expect(something).to の後ろに続くコードを置き換えます。注意してほしいのは、RSpec が Rails の UrlHelpers ライブラリを読み込まない点です。なので、パスの呼び出しは省略せずに全部書いてやる必要があります。私たちがチェックするのはマッチャに渡された *actual* が(ここでは response が)、期待した通りに動いているかどうか(ログインフォームにリダイレクトするかどうか)です。期待通りならマッチャは成功と報告します。

次に、example がパスしなかったときに返すメッセージを定義します。最初の定義は *true* を期待しているときのメッセージです。二つ目は *false* を返してほしいときのものです。つまり、一つのマッチャで expect(:foo).to と expect(:foo).not_to の両方を定義したことになります。二つのマッチャを定義する必要はありません。

さて、example で使われているマッチャを置き換えるのは簡単です。

spec/controllers/contacts_controller_spec.rb

```
describe 'GET #new' do
  # ログインを要求すること
  it "requires login" do
    get :new
    expect(response).to require_login
  end
end
```

これはカスタムマッチャで実現できる一つの例にすぎません。ただ、ここでもっと凝った例を登場させるとやり過ぎになってしまいます(混乱の原因にもなります)。カスタムマッチャについて詳しく知りたい場合は、RSpec のドキュメントに載っている定義例[33]を読んでみてください。

まとめ

コントローラのスペックは放置するとすぐ制御不能になって肥大化します。しかしちょっと管理してやれば(そして RSpec の便利なサポートメソッドを活用すれば)、そうした問題を食い止め、長期的にメンテナンスしやすい状態に保つことができます。コントローラスペックを無視すべきではないのと同様、責任を持ってコントローラスペックをきれいにしておくことも忘れないでください。今そうしておけば、きっと将来「やっておいて良かった」と思えるはずです。

私たちはここまでコントローラのテストに多くの時間を費やしてきました。第 5 章の冒頭で述べたように、コントローラレベルのテストがあれば大きなコストをかけずにコードベースの大部分を信頼できるようになります。そしてまた、他のレベルをテストするときのベストプラクティスにもなります。なぜなら、同じ考え方は他のレベルのテストでも応用できるからです。テストをクリーンで読みやすい状態に保てば、アプリケーションの生涯全体に渡ってコードを信頼し続けることができるはずです。

それでは次のレベルのテスト、すなわち統合テストに進みましょう。ここまでやってきたことは、アプリケーションの構成要素に安心感を与えるものでした。次に確認することは各要素がきちんとかみ合って一つのシステムとして成り立っていることです。

演習問題

- あなたのテストスイートにきれいに整理できる部分がないか調べてみてください。コントローラスペックが第一のターゲットですが、モデルスペックもチェックしてください。その部分をきれいに

[33] https://www.relishapp.com/rspec/rspec-expectations/v/3-1/docs/custom-matchers

する一番良い方法は何でしょうか。shared examples？カスタムマッチャ？ヘルパーマクロ？必要に応じてあなたのスペックを変更してください。そしてそのあとも引き続きテストがパスすることを確認してください。

- 第 5 章で私は、`expect(@contact.reload.hidden?).to be_true` はカスタムマッチャにまとめられると述べました。それはどんなやり方になるでしょうか？

8. フィーチャスペック

現時点で私たちは連絡先管理ツール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 フィーチャスペック（feature specs） と呼んでいます。フィーチャスペックは 受入テスト と呼ばれることもあります。フィーチャ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。またフィーチャ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、あなたは堅牢なフィーチャ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。フィーチャ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 Girl を使ってテストデータを生成することもできます。しかし、この章の主役は *Capybara* です。*Capybara* は大変便利な Ruby ライブラリで、フィーチャ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。

本章ではフィーチャスペックの基礎を説明します。

- まず最初に、フィーチャスペックをいつ、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。
- それからフィーチャ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、フィーチャ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。

本章の完成後のコードを見たい場合はサンプルソースの *08_features* ブランチをチェックアウトしてください。コマンドラインを使う場合は次のようにタイプします。

```
git checkout -b 08_features origin/08_features
```

本章を読みながら一緒にコードも書いていく場合は、前の章のブランチから始めてください。

```
git checkout -b 07_controller_cleanup origin/07_controller_cleanup
```

その他、詳細については第 1 章を読んでください。

なぜフィーチャスペックなのか？

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 だからです。コントローラのテストは確かにソフトウェアの重要なコンポーネントをテストするものの、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ません。フィーチャ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、フィーチャ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。

Cucumber はどうなのか？

Cucumber[34]はこの種のテストでよく出てくるもう一つの選択肢です。正直に言うと、この数年間私は Cucumber が大好きになったり大嫌いになったりしてきました。しかし、現在はもう使っていません。Cucumber は導入のコストも非常に高いですし、正しい使い方を理解していないと壊れやすくて全く使い物にならないテストを作ってしまう恐れがあります。

もしあなたがコードをあまり見たがらない非プログラマのプロダクトオーナーと直接 一緒に 働いているのなら、Cucumber を使いたくなる気持ちもわかります。しかし私の経験からすると、Capybara の DSL を使えば非プログラマでもフィーチャスペックを読んで、そこでやっていることを理解するのは十分可能だと思います。そして、もしあなたが非プログラマと働いて いない のであれば、Cucumber を導入するためにかかるコストは割に合わないものになると思います。

もちろん、Cucumber にも熱烈な支持者がいるのは知っています。Cucumber は今なおたくさんの現場で使われているので、あなたは結局 Cucumber を使えるようにならなければいけないのかもしれません。ただ嬉しいことに、もしあなたが将来的に Cucumber を使いたい、もしくは使う必要があるのなら、フィーチャスペックの習得は無駄になりません。なぜなら Capybara と RSpec の使い方を理解しておけば、Cucumber の理解も楽になるからです。

それでもあなたが Cucumber を使おうとするなら、2011 年 12 月以前に存在していたチュートリアルに注意してください。なぜならそのタイミングで Cucumber の *web_steps.rb* は “学習用の練習台”[35] に過ぎないということが明らかになったからです。ちなみに *web_steps.rb* は、Cucumber で `When I fill in "Email" with "aaron@everydayrails.com"` といったステップを提供してくれるヘルパーファイルです。2011 年 12 月以降は、Cucumber のシナリオをもっと簡潔にして、Capybara を使ったカスタムステップ定義（custom step definitions）を積極的に活用する方が良いとされています。

必要な追加ライブラリ

第 2 章ではCapybara[36]とDatabaseCleaner[37]、それにLaunchy[38]を Gemfile の `test` グループに追加しました。もしこれらの gem をまだ追加していなければ、今追加してください。今からついにこれらの gem を使います。

DatabaseCleaner はちょっとした設定の追加も必要になります。ただし、最初は DatabaseCleaner がいらないシンプルなスペックを見ていきます。

フィーチャスペックの基礎

Capybara を使うと、ユーザーがブラウザを通じてあなたのアプリケーションとやりとりする様子をシミュレートすることができます。シミュレートする際は `click_link` 、`fill_in` 、`visit` といったわかりやすいメソッドを使います。そしてこれらのメソッドを使ってアプリケーションのテストシナリオを記述します。あなたはこのフィーチャスペックが何をやっているのかわかるでしょうか？

[34] http://cukes.info
[35] http://aslakhellesoy.com/post/11055981222/the-training-wheels-came-off
[36] https://github.com/jnicklas/capybara
[37] https://github.com/bmabey/database_cleaner
[38] http://rubygems.org/gems/launchy

spec/features/users_spec.rb

```
require 'rails_helper'

feature 'User management' do
  # 新しいユーザーを追加する
  scenario "adds a new user" do
    admin = create(:admin)

    visit root_path
    click_link 'Log In'
    fill_in 'Email', with: admin.email
    fill_in 'Password', with: admin.password
    click_button 'Log In'

    visit root_path
    expect{
      click_link 'Users'
      click_link 'New User'
      fill_in 'Email', with: 'newuser@example.com'
      find('#password').fill_in 'Password', with: 'secret123'
      find('#password_confirmation').fill_in 'Password confirmation',
        with: 'secret123'
      click_button 'Create User'
    }.to change(User, :count).by(1)
    expect(current_path).to eq users_path
    expect(page).to have_content 'New user created'
    within 'h1' do
      expect(page).to have_content 'Users'
    end
    expect(page).to have_content 'newuser@example.com'
  end
end
```

このスペックのステップを順番に見ていくと、最初に新しい管理者（他のユーザーを作成できるユーザー）を作っているのがわかるはずです。それからログインフォームを使って管理者としてログインし、実際の管理者が使う Web フォームと同じものを使って新しいユーザーを作成します。これはフィーチャ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 `contacts#index`、`sessions#new`、`users#new`、それに `users#create` です。しかし、結果は同じになります。新しいユーザーが作成され、アプリケーションは全ユーザーの一覧へリダイレクトし、処理の成功を伝えるフラッシュメッセージが表示され、新しいユーザーが一覧に表示されるのです。

また、読者のみなさんは前章でも使っていたテクニックに気付いているかもしれません。`feature` は `describe` と同様にスペックを構造化しています。`scenario` は `it` と同様に example の起点を表しています。そして、第 5 章で見た `expect{}` Proc はここでも同じ役割を担っています。すなわち、ユーザーがこのサイトを訪れて、スクリプトに書かれている操作を完了したときに、何か変化が起きることを私たちは期待（expect）しているのです。

`find('#password')` と `find('#password_confirmation')` はここで何をしているかわかりますか？予想が付いているかもしれませんが、このメソッドはその名の通り、表示されているページ上の要素を見

つけて きます。要素を探す際は引数として渡した値を使います（ActiveRecord の `find` メソッドと間違わないでください）。ここでは CSS による検索、すなわち `<div>` 要素を `id` で検索しています。検索する場合は XPath のロケーションで探すこともできますし、`click_link 'Users'` や `fill_in 'Email'` というふうに、画面に表示されているプレーンテキストを指定して要素を探すことも可能です。しかし、一致があいまいになるとスペックは失敗します。たとえば、次のようなコードを書いたとします。

```
fill_in 'Password', with: 'secret'
fill_in 'Password confirmation', with: 'secret'
```

すると Capybara は *Ambiguous match*（あいまいな一致）というエラーを返します。なぜなら *password* という単語がどちらのラベルにも表示されているからです。このエラーに出くわしたら表示されている HTML をよく調べて、操作したいフィールドを指定する別の方法を探してください。（Capybara 2.0 以前だと、`fill_in` のようなメソッドを使ってもエラーは起きません。）

可能であれば、私はできるだけプレーンテキストを使います。その次に CSS を使います。どちらを使ってもうまくマッチしない場合は、仕方なく XPath ベースのマッチャを使います。詳しい情報については Capybara の README ファイルを参照してください。

`find` メソッドも JavaScript を使ってインターフェースをテストするときに便利です。詳しくは本章でのちほど説明します。

`expect{}` 以降では、一連のテストを実行して期待通りにビューが表示されることを、Capybara を使って確認しています。Capybara の DSL は自然な英文にならないところもありますが、それでも理解しやすい形式です。さらに注目してほしいのは、特定の要素を検索する 場所 を指定するために `within` ブロックを使っている点です。この場合であれば、ユーザー用の `index` ビューにある `<h1>` タグの内部を指定しています。これはパスワードと確認用パスワードのフィールドを特定する場合に、`find()` の代わりとして使える方法です。お望みであれば、もっと凝った方法も使えますが…詳しい話はこのあとすぐ出てきます。

さて、最後のポイントを今から話します。フィーチャ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。一般的にフィーチャ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フィーチャスペックを用意する方が望ましいでしょう。

リクエストからフィーチャへ

2012 年の 11 月、Capybara 2.0 は DSL にいくつかの変更を加えました。その中には `request` の代わりに前述の `feature` という用語を使うことも含まれていました。リクエストスペック（request specs）はまだ使えますが、現在ではパブリック API のテストで使う想定になっています。

リクエストスペックの場所が変わったことに加えて、Capybara 2.0 ではいくつかのエイリアスも導入され、他のフレームワーク（例えば Cucumber）を使った受入テストのようにフィーチャスペックが書けるようになっています。そのエイリアス、つまり前述の `feature` と `scenario` はフィーチャスペック専用に用意されたものです。この他にも、`before` のエイリアスである `background` や、`let`（第 9 章で説明します）のエイリアスである `given` があります。

厳密に言えば、`describe` も `it` もフィーチャスペックの中で 使えます 。とはいえ、最善の結果を求めるのなら、新しい Capybara の DSL を使う方が良いでしょう。この先登場するアドレス帳アプリケーションのスペックもこのように書いていきます。

フィーチャスペックを追加する

アプリケーションに新しいフィーチャスペックを追加する場合は、spec/features ディレクトリ内に新しいファイルを作るのが一番手っ取り早いです。ファイルは次のようなテンプレートで書き始めます。

```
require 'rails_helper'

feature 'my feature' do
  background do
    # セットアップの詳細を追加する
  end

  scenario 'my first test' do
    # example を書く！
  end
end
```

執筆時点では、Rails の scaffold ジェネレータを使ってモデルとそれに関連するコントローラ、ビュー、マイグレーション、スペックを作成すると、それに対応するフィーチャスペックが spec/requests に追加されます。そのファイルは削除するか、spec/features に移動して編集してください。scaffold ジェネレータがこうしたファイルを作らないようにすることもできます。その場合は、*application.rb* ファイルにある RSpec ジェネレータの設定で request_specs: false を指定してください。

フィーチャスペックをデバッグする

すでにお話ししたとおり、フィーチャのシナリオは複数のエクスペクテーションを持っていることが一般的です。しかし、そうすると特定のポイントでシナリオがなぜか失敗し、あなたは頭を抱えるかもしれません。デバッグする場合は通常、Ruby アプリケーションをデバッグするときと同じツールが RSpec 内でも使えます。しかし、最も簡単な方法の一つは *Launchy* を使うことです。Launchy は Capybara の依存関係に含まれており、たった一つの仕事をします。それは何かと言えば、フィーチャスペックの実行中にある時点の HTML を一時ファイルに保存し、デフォルトのブラウザで表示することです。

Launchy をスペックの中で使う場合は、結果を確認したいステップの一つ手前で次のような一行を追加します。

```
save_and_open_page
```

たとえば、本章の冒頭で紹介したフィーチャスペックであれば、次のようにして新規ユーザーフォームの実行結果を確認することができます。

spec/features/users_spec.rb

```
require 'rails_helper'

feature 'User management' do
  # 新しいユーザーを追加する
  scenario "adds a new user" do
    admin = create(:admin)
    sign_in admin

    visit root_path
    expect{
      click_link 'Users'
      click_link 'New User'
      fill_in 'Email', with: 'newuser@example.com'
      find('#password').fill_in 'Password', with: 'secret123'
      find('#password_confirmation').fill_in 'Password confirmation',
        with: 'secret123'
      click_button 'Create User'
    }.to change(User, :count).by(1)

    save_and_open_page

    # 残りのシナリオ
  end
end
```

もちろん、必要がなくなったら save_and_open_page の行を削除してください。

ちょっとしたリファクタリング

次へ進む前に、新規ユーザーを作成するフィーチャスペックをもう一度見てみましょう。リファクタリングできるポイントが少なくとも一箇所はあります。思い出したかもしれませんが、第 7 章で私たちはユーザーのログインをシミュレートするコードを抽出してヘルパーマクロへ移動させました。同じ事はフィーチャスペックでも可能です。

しかし、コントローラスペックと全く同じテクニックを使わないのはなぜでしょうか？それはなぜなら、テストの方法が違うからです。フィーチャレベルのテストでは、実際のユーザーがアプリケーションと やりとりする のと同じ方法でテストします。これはログインするときも同様です！しかし、これはログインのステップがヘルパーに抽出できないという意味ではありません。ではやってみましょう。

spec/support/login_macros.rb

```
module LoginMacros
  # コントローラ用のログインヘルパーは省略...

  def sign_in(user)
    visit root_path
    click_link 'Log In'
    fill_in 'Email', with: user.email
    fill_in 'Password', with: user.password
    click_button 'Log In'
```

```
10    end
11  end
```

そしてこのヘルパーは、こんなふうにフィーチャスペックで使うことができます。

spec/features/users_spec.rb

```
1  feature 'User management' do
2    # 新しいユーザーを追加する
3    scenario "adds a new user" do
4      admin = create(:admin)
5      sign_in admin
6
7      # 残りのステップは省略...
8    end
9  end
```

JavaScript を利用するやりとりも含める

さて、私たちはスペックを使って連絡先追加用のユーザーインターフェースが期待通りに動作することを検証できました。

では次にナビゲーションバーにある About リンクをテストしましょう。表面上はとても基本的なテストに見えますが、実は新しいテストのテクニックが必要になってきます。

スペックはこんな感じになります。

spec/features/about_us_spec.rb

```
1  require 'rails_helper'
2
3  feature "About BigCo modal" do
4    # about のモーダル表示を切り替える
5    scenario "toggles display of the modal about display" do
6      visit root_path
7
8      expect(page).not_to have_content 'About BigCo'
9      expect(page).not_to \
10       have_content 'BigCo produces the finest widgets in all the land'
11
12     click_link 'About Us'
13
14     expect(page).to have_content 'About BigCo'
15     expect(page).to \
16       have_content 'BigCo produces the finest widgets in all the land'
17
18     within '#about_us' do
19       click_button 'Close'
20     end
21
22     expect(page).not_to have_content 'About BigCo'
23     expect(page).not_to \
24       have_content 'BigCo produces the finest widgets in all the land'
```

```
25    end
26  end
```

何も複雑なことはありません…しかし、問題があります。今はフィーチャスペックを実行するときにCapybara 標準の web ドライバを使っています。Rack::Test というこのドライバは JavaScript を実行できないため、JavaScript は無視されます。よって example にある最初のエクスペクテーションは失敗します。なぜなら JavaScript で *application.html.haml* にある `#about_us` div を隠しているからです。JavaScript がなければ Rack::Test にその div が見えてしまい、テストが失敗します。

幸いなことに、Capybara は *Selenium* web ドライバを初めからサポートしています。Selenium を使うと、JavaScript を含む複雑な web アプリ上の操作を Firefox 経由でシミュレートできます。Selenium は軽量な web サーバを通じてテストコードを実行し、そのサーバとブラウザのやりとりを自動化することができるのです。

残念なことに、Selenium は Ruby 以外の依存関係を追加します。それは Firefox です。Mozilla Foundation はブラウザを頻繁にアップデートするので、場合によってはテストが動かなくなるかもしれません。もしそうなった場合は、`selenium-webdriver` gem[39]を最新版にアップデートしてみてください。

Selenium を使うためには、example に小さな変更を加える必要があります。

spec/features/about_us_spec.rb

```
1  require 'rails_helper'
2
3  feature "About BigCo modal" do
4    # about のモーダル表示を切り替える
5    scenario "toggles display of the modal about display", js: true do
6      # the example ...
7    end
8  end
```

どこが変わったのか注意して見てください。scenario に `js: true` が追加されました。これで Capybara は JavaScript を実行できるドライバ（デフォルトは Selenium）を使うようになります。これだけでおしまいです！スペックをもう一度実行すると、Firefox が起動し、シナリオの各ステップを実行していくので、その様子を見てください。

ただ、この example はちょっと単純すぎるかもしれません。デモを見せる目的で、Selenium を使ってユーザーとやりとりする example をもうちょっと見てみましょう。本章で作成した最初のシナリオは、実行するのに JavaScript は 必要 ありません。ですが、とりあえず JavaScript を有効にしてみて何が起きるのか見てみましょう。

[39] http://rubygems.org/gems/selenium-webdriver

spec/features/users_spec.rb

```
feature 'User management' do
  # 新しいユーザーを追加する
  scenario "adds a new user", js: true do
    # シナリオの各ステップ...
  end
end
```

この場合、テスト内のデータベーストランザクションをうまく扱えるように DatabaseCleaner を設定する必要もあります。*spec/rails_helper.rb* を以下のように変更しましょう。この設定は DatabaseCleaner の README に載っていたものです。

spec/rails_helper.rb

```
RSpec.configure do |config|

  # これより前の設定は省略...

  config.before(:suite) do
    DatabaseCleaner.strategy = :transaction
    DatabaseCleaner.clean_with :truncation
  end

  config.around(:each) do |example|
    DatabaseCleaner.cleaning do
      example.run
    end
  end

  config.after(:each) do
    DatabaseCleaner.clean
  end

  # その他の設定も省略...
```

`DatabaseCleaner.cleaning` はバージョン 1.3.0 から導入されました。

これは何をやっているのでしょうか？最初にやっているのはテストデータをセットアップする際に使う戦略（strategy）の指定です。それぞれのテストを独立させた状態にするため、ここではトランザクション（transaction）を指定しました。ご想像の通り、これはデータベースのトランザクションを使う戦略です。続いて、データを全件削除（clean）するためにテーブルのトランケート（truncation）を指定しています。最後に、トランザクションの開始と終了を実行するタイミングと（これはテストスイート中の各 example を実行する前後です）、全件削除を実行するタイミング（これは各 example を実行した後です）を指定しています。

本書の以前の版では `config.use_transactional_fixtures` に `false` を明示的にセットしていました。しかし、これは RSpec のデフォルト設定と同じであるため、この版では省略しています。

次に、ActiveRecord にモンキーパッチを当てて、スレッドを使うようにしなければなりません。`ActiveRecord::Base` を書き換えるために、次のようなファイルを *spec/support* に追加してください。

spec/support/shared_db_connection.rb

```
class ActiveRecord::Base
  mattr_accessor :shared_connection
  @@shared_connection = nil

  def self.connection
    @@shared_connection || retrieve_connection
  end
end
ActiveRecord::Base.shared_connection = ActiveRecord::Base.connection
```

なぜこのような変更が必要になるのでしょうか？端的に答えるのであれば、Selenium を使った場合、データベーストランザクションの扱い方に違いが出てくるからです。テストを実行するときは Selenium web サーバとテストコードの双方でデータの状態を共有しなければいけません。DatabaseCleaner と上記のパッチがないと、テスト実行後のクリーンアップが正しく行われないために、テストが時折失敗する可能性があります。

以上のような変更を加えると、このフィーチャスペックを Firefox 上で実行できるようになり、テストの信頼性が一段と向上します。

DatabaseCleaner にはここでは紹介しなかった数多くのオプションがあります。テストの要件やデータベースの種類（リレーショナルやドキュメント指向等）、選択した ORM（ActiveRecord や Mongoid、Sequel 等）によっては設定を変える必要があるかもしれません。こうしたオプションの詳しい設定方法については DatabaseCleaner の README[40] を確認してください。

DatabaseCleaner の設定を追加したり変更したりすると、それまでパスしていたテストが失敗し始める可能性があります。その場合は別の設定を試してみましょう。また、設定の変更でテストが失敗したということは、壊れやすいテストが潜んでいるサインなのかもしれません。テストを一度じっくりレビューしてみてください。

Capybara のドライバ

ここまで私たちはフィーチャスペックで二つのドライバを使ってきました。デフォルトのドライバである RackTest は基本的なブラウザ操作のテストで使える選択肢です。RackTest は ヘッドレス（headless） なので、ブラウザの操作はすべてバックグラウンドでシミュレートされます。Selenium はもっと複雑な操作をする場合に使えます。Selenium を使えば JavaScript やリダイレクト（アプリケーション外へのリダイレクトも含みます）が必要になる場合でも大丈夫です。

しかし、Selenium の機能は高く付きます。Firefox が起動し、ブラウザ上でスペックが実行されるのを毎回待つというのは明らかに退屈でしょう。特にテストスイートが大きくなればなるほど、この問題も大きくなります。幸いなことに、JavaScript をサポートするヘッドレスな選択肢も存在します。人気のある Capybara 向けのヘッドレスドライバは二つあります。一つはcapybara-webkit[41]で、もう一つはPoltergeist[42]です。どちらのドライバもライブラリの追加が必要になり、セットアップにも多少時間がかかるかもしれません。しかし、RackTest では対応できないフィーチャスペックが増えてくるのであれば、セットアップの時間に見合った価値が得られるはずです。これらのドライバの詳しいセットアップ方法についてはCapybara の README[43]を参照してください。

[40] https://github.com/DatabaseCleaner/database_cleaner
[41] https://github.com/thoughtbot/capybara-webkit
[42] https://github.com/jonleighton/poltergeist
[43] https://github.com/jnicklas/capybara

JavaScript の完了を待つ

本章の前半で、Capybara の `find` メソッドは JavaScript のテストをする際に便利だと説明しました。`find` はあなたが指定した項目が現れるまで処理を中断します。たとえば、`find` のバリエーションとして以下のようなコードを書くと、ボタンが現れるまで処理が止まります。

```
find_button('Close').click
```

Capybara で設定されているデフォルトのタイムアウト秒数は 2 秒です。次のようにすればタイムアウト秒数を好きなように調整できます。

```
Capybara.default_wait_time = 15
```

この場合、タイムアウト秒数は 15 秒になります。

この設定を *rails_helper.rb* ファイルに書けば、テストスイート全体でこの設定が有効になります。また、テストスイート全体ではなく、example ごとに指定することも可能です。

まとめ

本章は長い章ではありませんでした。特に、コントローラにかけた時間に比べるとかなり短いものです。しかし、本章では数多くの新しいコンセプトが登場しています。そして、これはあなたがここまで学んできたことが基礎になっています。実際、こんなに短く済んだのはこれより前の六つの章であなたがしっかりスキルを身に付けたおかげです。必要であれば何度か見直して、繰り返し練習してください。もし行き詰まったら、ブラウザを起動し、テストで期待していることがブラウザの中で実際に起きているかどうか確認してください。これは別にルール違反ではありません。(Launchy を使って確認するのも便利です。)

現時点であなたは Rails アプリケーションのテストで使用する重要なツールとテクニックに触れてきました。しかし、話を締めくくる前に説明すべき内容がまだいくつか残っています。次章ではだんだん大きくなってきたテストスイートを、できるだけ素早く実行させるテクニックを見ていきます。

演習問題

- フィーチャ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慣れてきたら、より複雑な操作へと進みましょう。
- コントローラスペックのときと同様に、この機会を利用してリファクタリングできそうなコードを探してください。同じ事を言いますが、テストの前にセットアップがたくさん必要になっているのであれば、それはコードベースをきれいにできるサインです。コードをクリーンアップし、フィーチャスペックを再実行してください。そのあともスペックはパスしますか？
- フィーチャの 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？

9. スペックの高速化

第 7 章ではコントローラを一通りリファクタリングして可読性と保守性を向上させました。完了したタスクは具体的に言うと次の三つです。

- shared examples を使って冗長なコードを減らしました。
- よく使う機能をヘルパーマクロに移動しました。
- カスタムマッチャを作り、example 中のエクスペクテーションをシンプルにしました。

テストスイートがほぼ完璧な状態になったところで、もう一度リファクタリングする方法を考えてみましょう。ただし、今回はスピードについてです。

私は スピード 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。本書で扱っている小さなアプリケーションのテストでも徐々にスピードが低下してきています。アプリケーションが大きくなるにつれ、そしてそれに合わせてテストスイートが大きくなるにつれ、確実に遅くなっていくでしょう。放置すれば状況は悪くなる一方です。よって目標としたいのは RSpec の可読性を損なわずに、実行時間を管理しやすい状態に保つことです。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
- Shoulda のカスタムマッチャで実現するシンプルなスペック
- モックとスタブで実現する、さらに効率的なテストデータ
- 遅いスペックを除外するためのタグの使用
- テスト実行の自動化と Rails のプリロード（事前読み込み）
- テスト全体をスピードアップするテクニック

本章の完成後のコードを見たい場合はサンプルソースの *09_speedup* ブランチをチェックアウトしてください。コマンドラインを使う場合は次のようにタイプします。

```
git checkout -b 09_speedup origin/09_speedup
```

本章を読みながら一緒にコードも書いていく場合は、前の章のブランチから始めてください。

```
git checkout -b 08_features origin/08_features
```

その他、詳細については第 1 章を読んでください。

構文を簡潔にするためのオプション

これまで書いてきたスペックに対する批判が一つあるとすれば、それはスペックがあまりにも冗長な点かもしれません。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。そうしたテクニックを組み合わせて使うと 本当に スペックを効率よく書けたり、個別に使った場合でも間延びしたスペックをきれいにできたりします。

let()

ここまで私たちは before :each ブロックを使い、よく使うテストデータをインスタンス変数に割り当ててきました。これを実現するもう一つの方法は let() を使うことです。RSpec ユーザーの多くはこちらを好みます。let() には二つの利点があります。

1. let() は値を キャッシュ します。ただし、インスタンス変数には割り当てません。
2. let() は 遅延評価 されます。すなわち、スペックで呼び出されるまで割り当てられません。

コントローラスペックで let() を使う方法は次のようになります。

spec/controllers/contacts_controller_spec.rb

```
require 'rails_helper'

describe ContactsController do
  let(:contact) do
    create(:contact, firstname: 'Lawrence', lastname: 'Smith')
  end

  # 残りのスペックは省略...
```

こうすると、@contact を通じて連絡先を使うのではなく、単に contact と書くだけでよくなります。たとえば、こんな感じです。

spec/controllers/contacts_controller_spec.rb

```
describe 'GET #show' do
  # contact に要求された連絡先を割り当てること
  it "assigns the requested contact to contact" do
    get :show, id: contact
    expect(:contact).to eq contact
  end

  # :show テンプレートを表示すること
  it "renders the :show template" do
    get :show, id: contact
    expect(response).to render_template :show
  end
end
```

しかし、これは問題の原因になります。具体的には、コントローラの destroy() メソッドが本当にデータベースからデータを消したのかどうかをテストする example で問題が起きます。現在失敗しているのはこのスペックです。

spec/controllers/contacts_controller_spec.rb

```
describe 'DELETE destroy' do
  # 連絡先を削除すること
  it "deletes the contact" do
    expect{
      delete :destroy, id: contact
    }.to change(Contact,:count).by(-1)
  end
end
```

この場合、件数（count）は変わりません。なぜなら expect{} Proc で contact を使うまで example は contact を知らないからです。これを修正する方法は、Proc の前で contact を呼び出すことです。

spec/controllers/contacts_controller_spec.rb

```
describe 'DELETE destroy' do
  # 連絡先を削除すること
  it "deletes the contact" do
    contact
    expect{
      delete :destroy, id: contact
    }.to change(Contact,:count).by(-1)
  end
end
```

この場合、let!()（エクスクラメーションマークに注目！）を使うこともできます。let!() を使うと各 example を実行する前に contact が強制的に割り当てられます。もしくは before ブロックの中で let() で割り当てた値を呼び出すこともできます。しかしこれだと let() を使って解決したかった問題を解決していないかもしれません。

subject{}

subject{} を使うとテスト対象（test subject）のオブジェクトを宣言できます。こうすれば、それ以降の example でそのオブジェクトを暗黙的に再利用できます。実際の使い方はこのあとでお見せします。

it{} と specify{}

it{} と specify{} はシノニム（同義語）の関係にあります。どちらもエクスペクテーションを囲むための単純なブロックです。第 3 章以降、私たちは長い形式で it{} を使ってきました。つまり、こんな example であれば……

```
subject { build(:user, firstname: 'John', lastname: 'Doe') }

it 'returns a full name' do
  should be_named 'John Doe'
end
```

このように変更できます。

```
subject { build(:user, firstname: 'John', lastname: 'Doe') }
specify { should be_named 'John Doe' }
```

こう書いても得られる結果は同じです。ささいな違いかもしれませんが、スペックが大きくなるにつれてワンライナーで書いたときの違いが現れてきます。また、私たちはこれまで expect の構文を使ってきましたが、ワンライナーではまだ should を使っている点にも注目してください。これはわざとです。RSpec の開発者も発言していますが、こういう example では should を使った方が読みやすくなります。

スペックを書いたら声に出して読んでください。そして最も意味が通じる用語を選んでください。どちらの用語を使うか決めるときに厳密なルールはありません。

Shoulda

Shoulda[44]はヘルパーを集めた拡張ライブラリで、テストでよく使う機能を簡単にしてくれます。この gem を一つ追加すれば、3 〜 5 行あったスペックが 1 〜 2 行に減らせるかもしれません。

subject() と it{} と specify{} は、shoulda-matchers gem と一緒に使ったときに真価を発揮します。shoulda-matchers を Gemfile の :test グループに追加すれば、数多くの便利なマッチャを自動的に使えるようになります。使用例を挙げてみましょう。

```
subject{ Contact.new }
specify { should validate_presence_of :firstname }
```

良い感じになりました。読みやすいですし、カバレッジ的にも十分です。自分で定義したカスタムマッチャを使って、さらに効率よく書くこともできます。たとえば、次のようなカスタムマッチャを定義します。

```
1  RSpec::Matchers.define :be_named do |expected|
2    match do |actual|
3      actual.name eq expected
4    end
5
6    description do
7      "return a full name as a string"
8    end
9  end
```

このカスタムマッチャは次のように it{} ブロックの中で簡単に呼び出せます。

```
it { should be_named 'John Doe' }
```

確かにこの例は少しやりすぎかもしれません。しかし、スペックをシンプルにしたいときは、いろんな方法があることを知ってもらいたくて紹介してみました。次のように、シンプルになっても可読性は犠牲になっていません。たとえば次の実行結果を見てください。

[44] http://rubygems.org/gems/shoulda

```
Contact
  should return a full name as a string
  should have 3 phones
  should require firstname to be set
  should require lastname to be set
```

こんな具合です。

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。これは Factory Girl を使ってやろうとしてきたことにちょっと似ています。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よく使う処理が対象になります。

ここで適当な例を二つ挙げてみます。

- 連絡先のモックを作る場合、Factory Girl の `build_stubbed()` メソッドが使えます。このメソッドを使うと、完全にスタブ化されたダミーオブジェクトが生成されます。つまり、このオブジェクトは `firstname` や `lastname` 、`fullname` といった様々なメソッドに応答することができるわけです。ただし、データベースの中にはそのデータはありません。
- Contact モデルに定義されているメソッドをスタブ化する場合は、次のようなコードを書くとスタブが使えるようになります。`allow(Contact).to receive(:order).with('lastname, firstname').and_return([contact])` このケースでは Contact モデルの `order` スコープをオーバーライドしています。ここで渡すのは SQL の order を指定する文字列です（この場合は lastname と firstname）。それからモックに対して返却してほしい値を指定しています（返却されるのは 1 要素の配列で、中にはスペック内で事前に作られたと思われる `contact` が入っています）。

実際のプロジェクトでは、ほとんどの場合RSpec 標準のモックライブラリ[45]か、Mocha[46]のような外部ライブラリが使われていると思います。他にも同じようなモックライブラリ[47]がたくさんあるので、こうしたライブラリが使われることもあります。初心者のために簡単に説明しておくと、微妙なトレードオフはあるにせよ、どのライブラリも同じような機能を持っています。

コントローラスペックの中でモックやスタブが使われている様子を見ると、もっとわかりやすいかもしれません。

[45] http://rubydoc.info/gems/rspec-mocks/frames
[46] http://rubygems.org/gems/mocha
[47] https://www.ruby-toolbox.com/categories/mocking

spec/controllers/contacts_controller_spec.rb

```
describe 'GET #show' do
  let(:contact) { build_stubbed(:contact,
    firstname: 'Lawrence', lastname: 'Smith') }

  before :each do
    allow(contact).to receive(:persisted?).and_return(true)
    allow(Contact).to \
      receive(:order).with('lastname, firstname').and_return([contact])
    allow(Contact).to \
      receive(:find).with(contact.id.to_s).and_return(contact)
    allow(contact).to receive(:save).and_return(true)

    get :show, id: contact
  end

  it "assigns the requested contact to @contact" do
    expect(assigns(:contact)).to eq contact
  end

  it "renders the :show template" do
    expect(response).to render_template :show
  end
end
```

スペックを順番に見ていきましょう。まず `let()` を使ってスタブ化されているモックの連絡先を `contact` に割り当てます。それから Contact モデルと `contact` インスタンスにスタブメソッドを追加します。コントローラは Contact と `contact` の ActiveRecord メソッドをいくつか使うので、このコントローラに使われるメソッドをスタブ化する必要があります。これらのスタブ化されたメソッドは、ActiveRecord からコントローラへ渡してほしい値を返します。最後に、example をまとめるために `it` ブロックを使います。これは今まで本書でやってきたことと同じです。しかし、今回のテストデータは全てモックとスタブから返されます。本物のデータベースや Contact モデルのメソッドを実際に呼び出すわけではありません。

この方法の利点は、example における関心事がこれまでに書いてきたスペックよりも分離されることです。なぜなら、この example の唯一の関心事は対象とするコントローラのメソッドだけであり、モデルやデータベース、その他もろもろには関心がないからです。一方、欠点として挙げられるのは、関心事を分離したことによって、スペックに余計なコードが増えていることです（可読性にも疑問が残ります）。

とはいえ、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。

Guard と Spring を使ったテストの自動実行

スペックの実行は頻繁に行うことが重要ですが、それを忘れると大きな時間のロスにつながりかねません。どこかで問題が起きていることに気付かず、問題となっているコードの上に新しいコードを書き

続けてしまうと、何分もの、いや場合によっては何時間もの貴重な時間を無駄にしてしまうかもしれません。しかしターミナルに切り替えてコマンドラインから `bin/rspec` を実行するのも面倒です（そしてまた、この作業にも時間を削られます）。そこで Guard の出番です！

Guard[48]は指定されたファイルを監視します。そして監視対象のファイルに応じて仕事を実行します。私たちのケースであれば、Guard に望むことは *app* ディレクトリと *spec* ディレクトリ内のファイルを監視し、ファイルが変更された際に適切なテストを実行してもらうことです。たとえば *app/models/contact.rb* を変更すると、*spec/models/contact_spec.rb* を実行する、といった具合です。もしテストが失敗すれば、テストがパスするまで実行を繰り返します。

Guard を使うには、まず Gemfile の `:test` グループと `:development` グループ（第 2 章を参照）に `guard-rspec` を追加してください。`guard-rspec` を追加すると、Guard 自身もインクルードされます。

それからコマンドラインから `Guardfile` を作成します。

```
bundle exec guard init rspec
```

こうすると `Guardfile` が Rails アプリケーションのルートに生成されます。`Guardfile` は Guard の設定ファイルです。そのまま使っても十分便利なのですが、自分の好みに合わせて多少変更したくなるかもしれません。私はよく次のような設定を使います。

- `notification: false`：私はポップアップメッセージよりもターミナルウインドウでスペックが実行されるのを見るのが好きです。
- `all_on_start: false` と `all_on_pass: false`：私は最近この設定を使っています。私は変更内容をコミットする前に全部のテストを実行することが重要だと理解しています。Guard の起動後にスペックを実行させたいときは、単に リターンキー を押します。あるテストがパスしたあとで全部のテストを実行する場合も同様です。私はこうした動作を自分の手で制御するのが好きです。
- ビューの変更時にフィーチャスペックを実行する：私はビュースペック（view specs）を書かないので、このレイヤのテストはフィーチャスペックに任せています。私は基本的にモデルやコントローラを変更してもフィーチャスペックは実行しません。とはいえ何事もそうですが、これは状況によります。

生成された `Guardfile` は本の中で見せるのには向いていません。実際のアプリケーションでどのように使われているのか見たい場合は、サンプルソースにある Guardfile を参照してください。

Guard を起動するには `bundle exec guard` を実行してください。Guard は全テストスイートを実行したあと、忠実に変更を監視し、必要に応じてスペックを実行します。もちろん、他のオプションも設定できます。たとえば私は通常、自分が望んだときにだけ全テストスイートを実行させるのが好きです。これを実現するには次のような設定を Guardfile に追加します。

Guardfile

```
1 guard :rspec, cmd: 'rspec --color --format documentation',
2   all_on_start: false, all_after_pass: false do
```

Guard は単にファイルを監視してスペックを実行するだけのものではありません。他にも Sass や LESS の CSS コンパイル、Cucumber フィーチャの実行、JavaScript の圧縮（minify）、コードメトリクスの収集、開発用サーバの再起動、といったことも実行できます。詳しくは Github にある Guard の全リスト[49]を見てください。

[48] https://github.com/guard/guard
[49] https://github.com/guard

いったんテストが起動すれば、テストの実行は十分速くなります。しかし、現状だとテストを起動させるたびに毎回待ち時間が発生します。この待ち時間の原因は Rails アプリケーションを毎回起動しなければならないせいです。Rails 4.1 から導入された Spring を使うとこの待ち時間を初回起動時だけに限定できます。そして起動後は非常に素早くスペックが実行されます。Guard と組み合わせて使うと、Spring はテストの実行時間を短くする良い解決策の一つになります。テストを個別に再構築する必要はありません。

RSpec と Spring を一緒に動かすためにはちょっとした追加のセットアップが必要になります。最初に Spring に `bin/rspec` コマンドのサポートを追加しましょう。これには `spring-commands-rspec` gem を利用します。

Gemfile

```
group :development, :test do
  gem 'spring-commands-rspec', '~> 1.0.2'
  # ...
end
```

次に、`bin/rspec` コマンドを binstub として実行できるようにします。

```
$ bundle exec spring binstub rspec
$ bin/spring stop
```

こうすれば Guard ファイルの中でも Spring 経由で RSpec を読み込むことができます。

Guardfile

```
1 guard :rspec, cmd: 'spring rspec --color --format documentation',
2   all_on_start: false, all_after_pass: false do
```

タグ

Guard を使うかどうかにかかわらず、RSpec のタグ（tags）機能[50]を使えば、実行するスペックをうまく使い分けることができます。タグを使うには、対象の example にタグを付けます。

```
# 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
```

こうすればコマンドラインから `focus` タグが付いたスペックだけを実行させることができます。

```
$ bin/rspec --tag focus
```

タグ機能が便利だと思ったら、`run_all_when_everything_filtered` を設定すると良いかもしれません。この設定を追加すると、指定されたタグにマッチする example が 1 件もないときに 全ての spec が実行されるようになります。

[50] https://www.relishapp.com/rspec/rspec-core/v/2-4/docs/command-line/tag-option

spec/rails_helper.rb

```
RSpec.configure.do |config|
  config.run_all_when_everything_filtered = true
end
```

特定のタグが付いた example だけを実行する（または実行しない）ように RSpec を設定することもできます。以下はその例です。

spec/rails_helper.rb

```
RSpec.configure do |config|
  config.filter_run focus: true
  config.filter_run_excluding slow: true
end
```

これは Guard を使う場合に特に便利です。なぜなら、rails_helper.rb 内で特定のタグを有効、もしくは無効にすると Guard が自動的にリロードし、そのあとまた使い続けられるからです。私はこの機能をそれほど頻繁に使うわけではありませんが、場合によっては大変便利です。

高速化させる他の解決策

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 残したいと思うなら、そのスペックに skip のマークを付けてください。

```
1 it "loads a lot of data" do
2   skip "no longer necessary"
3   # スペックのコードが続く。ただし実行はされない
4 end
```

以前のバージョンの 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 RSpec 3 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。

Rails を取り外す

これまでに加えてきた変更は全て、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。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、さらにもう一歩高速化させることができます。

この話題は本書のスコープを少し超えてしまいます。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[51]とDestroy All Software[52]を視聴することをお薦めします。

[51] http://confreaks.com/videos/641-gogaruco2011-fast-rails-tests
[52] https://www.destroyallsoftware.com/screencasts

まとめ

本章では少し難しい話題について見てきました。これまで私はテストのやり方を変えるテクニックについては説明してきませんでした。しかし、これであなたは選択肢の一つとしてそのテクニックを身につけました。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。あなたはテストデータを読み込み、それを活用する様々な方法も選ぶことができます。たとえば、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、それを実行するテクニックについてもいくつか身につけました。あなたは順調に RSpec を使いこなせるようになってきています。

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で `let()` 、`subject{}` 、`it{}` 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。)
- `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 のソースを調べて、Shoulda が RSpec ユーザーに提供している全マッチャについて学習してください。
- 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？

10. その他のテスト

現時点で私たちはアドレス帳アプリケーションをほとんど網羅するテストを作りました。モデルとコントローラのテストは終わっています。それだけでなく、フィーチャスペックでもビューと一緒にテストしました。これぐらい簡単なアプリケーションなら、基礎テクニックだけで十分カバーできるでしょう。しかし、多くの Rails アプリケーションは（おそらくあなたのアプリケーションも）こんなに単純ではありません。あなたのアプリケーションはユーザーにメールを送信したり、ファイルアップロードを処理するかもしれません。もしくは API 経由で外部の web サービスと通信するかもしれません。もしかすると自分が API の提供者になっているかもしれませんね。あるいは特定の日付や時間に実行される機能もあるかもしれません。こうした機能もテストできるんです！

本章では次のような内容を説明します。

- メール送信をテストする方法
- ファイルアップロードのテスト
- スペックの中で時間を扱う方法
- 外部の web サービスを利用するテスト
- 自分のアプリケーションが提供する API のテスト
- rake タスクのテスト

本章のサンプルコードはありません。試しにこのアドレス帳アプリケーションに機能を追加しようとしましたが、わざとらしい使用例しか思いつきませんでした。とはいえ、本書の将来のリリースではいくつかのサンプルコードが登場するかもしれません。

メール送信をテストする

メーラー（mailer）の動きをテストするのは比較的簡単です。ある gem を使うこともできますし、通常の Ruby と RSpec を使って自分で処理することもできます。

ある gem とはEmail Spec[53]のことです。この gem を導入すると、メールの送信先や件名、ヘッダ、本文をテストする便利なカスタムマッチャが使えるようになります。この gem を Gemfile の `:test` グループに追加して `bundle install` を実行したら、あとは *spec/rails_helper.rb* にいくつかの設定を追加するだけです。

spec/rails_helper.rb

```
1 require "email_spec"
2 config.include(EmailSpec::Helpers)
3 config.include(EmailSpec::Matchers)
```

この設定を加えれば、次のようなエクスペクテーションが使えるようになります。

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

```
# メール送信を実行するためのセットアップが完了していること...

expect(open_last_email).to be_delivered_from sender.email
expect(open_last_email).to have_reply_to sender.email
expect(open_last_email).to be_delivered_to recipient.email
expect(open_last_email).to have_subject message.subject
expect(open_last_email).to have_body_text message.message
```

上のサンプルコードで使われている open_last_email は一番最後に送信されたメールを開き、その属性を取得できるヘルパーメソッドです。ライブラリのドキュメント[54]に書かれているとおり、新しいメールオブジェクトを作成し、それを直接テストすることもできます。

```
email = MessageMailer.create_friend_request("aaron@everydayrails.com")
expect(email).to deliver_to("aaron@everydayrails.com")
expect(email).to have_subject "Friend Request"
```

ご覧の通り、Email Spec のカスタムマッチャは大変便利で読みやすいです。カスタムマッチャの全リストを確認する場合は、Email Spec のドキュメントを読んでください。また、そのドキュメントを読むときは、Email Spec で使えるヘルパーの一覧も確認してください。私が特に好んで使うのは open_last_email で、統合テスト(フィーチャ)レベルのテストでよく使っています。たいていの場合、Email Spec はモデルレベルやコントローラレベルでも便利に使えます。アプリケーションの中で一番役に立ちそうな場所で使ってください。

もしアプリケーションに gem を追加したくないのであれば、同じことを RSpec と Ruby で実現できます。必要になるのは Rails の ActionMailer に関するちょっとした知識です。メッセージは送信されるたびに ActionMailer::Base.deliveries に追加されていきます。なので、ActionMailer::Base.deliveries.last を見れば最後に送信したメッセージの内容がわかります。これを理解していれば、フィーチャスペックの中でメールに関するエクスペクテーションを追加することができます。たとえば、メッセージの To ヘッダにユーザのメールアドレスが含まれていることを検証する場合は次のようになります。

```
expect(ActionMailer::Base.deliveries.last.to).to include user.email
```

こういった記述は第 7 章で学んだ知識を使って改善できます。ヘルパーマクロを作ってきれいにしてみましょう。

spec/support/mailer_macros.rb

```
1  module MailerMacros
2    def last_email
3      ActionMailer::Base.deliveries.last
4    end
5
6    def reset_email
7      ActionMailer::Base.deliveries = []
8    end
9  end
```

こうすればスペックがもっと読みやすくなります。

[54] http://rubydoc.info/gems/email_spec/1.5.0/frames

```
expect(last_email.to).to include user.email
```

さらに私は deliveries 配列をリセットするためのヘルパーメソッドも用意しました。MailerMacros を rails_helper の中でインクルードし、スペックを実行する前に毎回 reset_email を実行するようにしてみましょう。

spec/rails_helper.rb

```
RSpec.configure do |config|
  config.include MailerMacros
  config.before(:each) { reset_email }
end
```

この方法はメーラーの単体テストを書くときにも活用できます。たとえば次の例を見てください。

spec/mailers/user_mailer_spec.rb

```
1  RSpec.describe UserMailer, :type => :mailer do
2    describe "friend reqeust message" do
3      let(:user) { FactoryGirl.create(:user)
4      let(:mail) { UserMailer.friend_request(user, friend) }
5
6      # 友達申請をユーザに送信すること
7      it "sends user a friend request" do
8        expect(mail.subject).to eq "New friend request from #{friend.name}"
9        expect(mail.to).to eq [user.email]
10       expect(mail.from).to eq ["from@example.com"]
11       expect(mail.body.encoded).to match edit_friendship_path(user, friend)
12     end
13   end
14 end
```

ご覧の通り、メールメッセージのほとんどの部分をテストすることができました。メールメッセージの構造を深く理解するために、ActionMailer を解説している Rails Guide[55] と API ドキュメント[56] を読むことをお薦めします。そうすればメールメッセージの何をテストすべきかがわかるはずです。

もし、メーラースペックでルーティング名に関するエラーが発生した場合は、*test.rb* ファイルで ActionMailer の default_url_options を設定しているかどうかを確認してください。

config/environments/test.db

```
config.action_mailer.default_url_options = {
  :host => "localhost:3000"
}
```

このセクションで紹介したマクロは Railscasts のお気に入りのエピソードである、"How I Test"[57] を参考にして作成しました。このエピソードはとりわけ私のお気に入りで、TDD を理解するのに非常に役立ちました。それまでに私が見たり読んだりしてきたどの情報源よりも優れていたエピソードです。公開されてから数年が経ち、構文が変わっている部分もありますが、自分のソフトウェアをテストしたいと思っている人はこのエピソードを視聴することを強くお薦めします。

[55] http://guides.rubyonrails.org/action_mailer_basics.html
[56] http://api.rubyonrails.org/classes/ActionMailer/Base.html
[57] http://railscasts.com/episodes/275-how-i-test

ファイルアップロードをテストする

ファイルアップロードの動作確認は、私が長い間行き詰まっていたポイントです。たとえば、どうやってダミーファイルをスペックに取り込めば良いのでしょうか？処理されたファイルはどこに保存されるのでしょうか？Rails には *fixtures* ディレクトリからファイルをアップロードする方法がありますが、これはうまくいくときといかないときがありました。なので、代わりに私は次のような単純な方法をよく使います。小さなダミーファイル（できれば本番データに近いもの）を *spec/factories* ディレクトリに置いてください。そうすれば次のようにしてファクトリからファイルを参照できます。

```
FactoryGirl.define do
  factory :user do
    # その他のユーザ属性

    factory :user_with_avatar do
      avatar { File.new("#{Rails.root}/spec/factories/avatar.png") }
    end
  end
end
```

もしあなたのモデルで添付ファイルが 必須になっている なら、ファクトリで作ったテストデータにファイルが保存されていることを確認した方が良いかもしれません。

さらに重要なことは、スペックの中で明示的にファイルにアクセスできることです。たとえば、フィーチャの example で使う場合はこんな感じです。

```
# 新しいユーザーを作成できること
it "creates a new user" do
  visit new_user_url
  fill_in 'Username', with: 'aaron'
  fill_in 'Password', with: 'secret'
  attach_file 'Avatar',
    "#{Rails.root}/spec/factories/avatar.png"
  click_button 'Sign up'
  expect(User.last.avatar_file_name).to eq 'avatar.png'
end
```

上で作ったファクトリを使うと、このようにコントローラレベルでテストすることもできます。

```
it "uploads an avatar" do
  post :create, user: attributes_for(:user_with_avatar)
  expect(assigns(:user).avatar_file_name).to eq 'avatar.png'
end
```

あなたがもし Paperclip や Carrierwave といったポピュラーなファイルアップロードライブラリを使っているなら、テスト用の便利メソッドが標準で用意されているはずです。使用している gem のドキュメントを参照し、使い方を調べてみてください。

時間をテストする

もし特定の時間や日にちに依存する実行結果を確認しなければならない場合、あなたならどうしますか？たとえば、1 月 1 日にだけ「明けましておめでとう」をサイトの訪問者に伝えたいときなどです。Timecop[58]を使うと時間を止めることができるので、そういうテストも書けるようになります。面倒な Ruby の日付操作は必要ありません。Timecop は Gemfile に追加するだけで、すぐ使えるようになります。次のような架空のフィーチャスペックで使い方を見てみましょう。

```
1  # 1 月 1 日に「明けましておめでとう」を訪問者に伝えること
2  it "wishes the visitor a Happy New Year on January 1" do
3    Timecop.travel Time.parse("January 1")
4    visit root_url
5    expect(page).to have_content "Happy New Year!"
6    Timecop.return
7  end
```

このサンプルコードでは `Timecop.return` を呼び出していることに注意してください。これによってシステム日時がリセットされ、RSpec は実行にかかった時間を正しく表示できるようになります。

Timecop は締め切り日を設定するような場合にも便利です。たとえば、あなたのアプリケーションでは期日まで納税の申告をしなければならないかもしれません。

```
1  # 4 月 15 日を過ぎると納税者は申告できなくなること
2  it "doesn't allow taxpayers to file after April 15" do
3    Timecop.travel Time.parse("April 16")
4    visit pay_taxes_path
5    expect(page).to have_content "Sorry, you're too late!"
6    Timecop.return
7  end
```

または

```
1  # 4 月 15 日までは納税者は申告ができること
2  it "gives taxpayers up until the 15th to file" do
3    Timecop.travel Time.parse("April 15")
4    visit pay_taxes_path
5    expect(page).to have_content "There's still time to file, but hurry!"
6    Timecop.return
7  end
```

私はよくこんなふうに Timecop を使いますが、よくある使い方がもう一つあります。それはテストの実行中に時間を 止める ことです。たとえば、あなたは Rails 標準のタイムスタンプ機能が 本当に 正しく動作しているか確認したいかもしれません。そんなときはモデルスペックで次のようにします。

[58] http://rubygems.org/gems/timecop

```
# 現在時刻でモデルの作成日時が記録されること
it "stamps the model's created at with the current time" do
  Timecop.freeze
  user = create(:user)
  expect(user.created_at).to eq Time.now
  Timecop.return
end
```

Timecop.freeze を使わなければ、データが保存された日時とスペックが値をチェックする日時が秒単位でずれるので、このテストは失敗します。

Web サービスをテストする

最近では Rails アプリケーションが外部の web サービスを利用することは一般的です。これまでに説明した内容を理解していれば、こういった機能をテストすることも可能です。しかし、実際にやってみた人はいくつかの問題に気付いているはずです。第一に、こういったテストはローカルリソースにだけアクセスするテストに比べると、はるかに遅くなりがちです。なぜならテストはネットワーク越しのリクエストがリモートサーバに送られ、処理され、レスポンスが返るまで待ち続けなくてはならないからです。第二に、サービスの呼び出しにレートリミットが設定されている場合、テストスイートを頻繁に実行するとあっという間にリミットに達してしまいます。

VCR[59] 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。

実際の使用例を見てみましょう。これは私の最近のプロジェクトで使ったコードです。このコードはGeocoder[60] gem を使って、入力された住所から緯度と経度を取得しています。このとき、Google Maps API へのアクセスが発生します。以下がそのコードです。

```
class Address < ActiveRecord::Base
  geocoded_by :address
  before_save :geocode

  def address
    "#{street}, #{city} #{state}"
  end
end
```

そしてこれが私の書いたモデルスペックです。コードの 7 行目でカセットを定義している点に注目してください。

[59] https://github.com/vcr/vcr
[60] http://www.rubygeocoder.com

```
require 'rails_helper'

RSpec.describe Address, :type => :model do
  describe 'geocoding' do
    # 新しい住所の緯度と経度を取得すること
    it 'geocodes a new address' do
      VCR.use_cassette('allen_fieldhouse') do
        address = FactoryGirl.create(:address,
          street: '1651 Naismith Drive',
          city:   'Lawrence',
          state:  'KS'
        )
        expect(address.latitude).to eq 38.9541438
        expect(address.longitude).to eq -95.2527379
      end
    end
  end
end
```

VCR はフィーチャスペックでも使用できます。次に紹介する使用例は実際の住所の UI です。ここでは let ブロックの中（6 行目）でカセットを使用しています。これでテストデータを作成するときにネットワーク呼び出しをモック化することができます。さらに二つ目のカセット（27 行目）を使って geocoding サービスに対する 2 回目の呼び出しを処理しています。このときはデータの更新時に緯度と経度の処理を行っています。

```
require 'rails_helper'

feature 'Addresses' do
  let(:user) { FactoryGirl.create(:user) }
  let(:address) {
    VCR.use_cassette('busch_stadium') do
      FactoryGirl.create(:address,
        user: user,
        street: '700 Clark Avenue',
        city: 'St. Louis',
        state: 'MO',
        zip: '63102'
      )
    end
  }

  # ユーザが住所を編集する
  scenario 'user edits a address' do
    old_address = address
    sign_in_as(user)
    visit user_path(user)

    within '#addresses' do
      click_link 'Edit'
    end
```

```
26
27      VCR.use_cassette('allen_fieldhouse') do
28        fill_in 'Street', with: '1651 Naismith Drive'
29        fill_in 'City', with: 'Lawrence'
30        select 'KS', from: 'State'
31        click_button 'Update Address'
32      end
33
34      expect(current_path).to eq user_path(user)
35      expect(page).to have_content 'Successfully updated address.'
36      expect(page).to have_content '1651 Naismith Drive'
37      expect(page).not_to have_content old_address.street
38      # などなど
39    end
40  end
```

VCR はセットアップが簡単で、ドキュメントも豊富です。Gemfile に gem を追加し、*spec/rails_helper.rb* でインクルードすれば、VCR で記録する準備が整います。

自分のアプリケーションが提供する API をテストする

これまでに私たちは外部のサービスから情報を取得する方法を見てきました。しかし、あなたのアプリケーションがモバイルアプリケーションや JavaScript ベースの web フロントエンド、もしくはサードパーティのアドオンからリクエストを受け取る立場だったらどうしますか？おそらくアプリケーションの API を用意する必要があるでしょう。そして、あなたは（さらに API の利用者も）API の信頼性を保ちたいと思っているので、テストを書きたくなるはずです。

堅牢でプログラマが使いやすいパブリック API を開発する方法は本書のスコープを超えてしまいます。しかし、テストに関してはそうではありません。幸いなことに、もしあなたがコントローラスペックとフィーチャスペックの章を読んできたのであれば、あなたは API をテストする基本的なテクニックを身につけているはずです。

まず最初に、このようなテストはどこに置くべきでしょうか？第 5 章で述べたように、JSON（または XML）の出力結果はコントローラスペックで直接テストできます。自分のアプリケーションでしか使わない単純なメソッドであれば、これで十分かもしれません。一方、本格的な API であれば、第 8 章で説明したフィーチャスペックのような結合テストが必要です。とはいえ、フィーチャスペックとはいくつか異なる点もあります。RSpec の場合、このような API 関連のテストは *spec/requests* ディレクトリに置くのが最も適切です。これまでに書いてきたフィーチャスペックとは区別する必要があります。さらに、API 用のスペックでは Capybara は使いません。Capybara はブラウザの操作をシミュレートするためのツールであり、プログラム上の操作をシミュレートするものではありません。その代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シンプルなメソッドを使います。すなわち、HTTP メソッドである `get` 、`post` 、`delete` 、それに `patch` です。

というわけでサンプルコードを二つ用意しました。一つ目は連絡先を取得する API 呼び出しのコード例です。おそらく、次のようなコードになると思います。

spec/requests/api/v1/contacts_spec.rb

```ruby
require 'rails_helper'

describe 'Contacts API', type: :request do
  # 指定された連絡先を返却すること
  it 'sends an individual contact' do
    contact = FactoryGirl.create(:contact)

    get "/api/contacts/#{contact.id}", nil, \
      { 'HTTP_ACCEPT' => 'application/vnd.contacts.v1' }

    expect(response).to have_http_status(:success)

    json = JSON.parse(response.body)
    expect(json['firstname']).to eq contact.firstname
    expect(json['lastname']).to eq contact.lastname
    expect(json['email']).to eq contact.email
    # などなど
  end
end
```

ではこのコードを順番に見ていきましょう。まず、テストで使用する新しい連絡先を永続化します。次に、get メソッドを使って API を呼び出し、その連絡先を要求します。このサンプルコードでは具体的なエンドポイントに対する GET リクエストを定義しています。パラメータはありません（nil です）。そして API バージョンは HTTP_ACCEPT ヘッダで指定しています。もし URL に API バージョンを含めるのであれば、この行は次のように変わるはずです。

```ruby
get "/api/v1/contacts/#{contact.id}"
```

API の実行結果は response オブジェクトとしてスペックに返却されます。そして、このオブジェクトを使って実行結果を検証していきます。このテストは結合テストなので、一つのスペック内で複数の内容を検証してもかまいません。少なくとも以下の 2 点は必ず検証すべきでしょう。まず、API が正しいステータスコードを返していることを確認します。ここでは have_http_status マッチャを使って、API が「成功」を意味するステータスコード 200 を返すことを検証しています。次に API から返却された JSON（respomse.body ）をパースし、正しいデータが格納されていることを検証しています。

では二つ目のサンプルコードを見てみましょう。このコードでは PUT リクエスト経由で連絡先を更新する API を使っています。

spec/requests/api/v1/contacts_spec.rb

```ruby
require 'rails_helper'

# 前述の example...

describe 'PUT /api/contacts/:id' do
  # 連絡先を更新すること
  it 'updates a contact' do
    contact = FactoryGirl.create(:contact)
    new_attributes = {
      contact: {
```

```
11          firstname: 'Aaron',
12          lastname: 'Sumner',
13          email: 'aaron@everydayrails.com',
14        }
15      }
16
17      put "/api/contacts/#{contact.id}", new_attributes, \
18        {'HTTP_ACCEPT' => 'application/vnd.contacts.v1' }
19
20      expect(response).to be_success
21      json = JSON.parse(response.body)
22      expect(json['id']).to eq contact.id
23      expect(json['firstname']).to eq 'Aaron'
24      expect(json['lastname']).to eq 'Sumner'
25      expect(json['email']).to eq 'aaron@everydayrails.com'
26    end
27  end
```

このコードは第 5 章で見たコントローラスペックによく似ています。ここでは異なるエンドポイントにアクセスし、ちょっと違った方法で実行結果を確認しているだけです。

もちろん、データベースを変更するような実際の API では認証が必要になるはずです。たとえば、シークレットトークンをヘッダかパラメータに含めたりすると思います。繰り返しますが、ここでのゴールは API 設計の原則を説明することではありません。そういう話はいったん抜きにしましょう。今回紹介したやり方はどちらも非常に簡単でしたね。ここで説明したかった一番大事なことを覚えておいてください。それはつまり、コントローラスペックとフィーチャスペックの知識を組み合わせれば、API のテストで必要なスキルはすでに身に付いているということです！

マクロを使うと `response.body` をパースする際の重複もなくせるかもしれません。マクロの作成は第 7 章で紹介済みです。これはみなさんの練習問題として残しておくので、ぜひトライしてみてください。

rake タスクをテストする

ある程度長い間 Rails を開発している人なら、一回ぐらいはアプリケーション用に Rake のコマンドラインユーティリティを書いたことがあると思います。私が Rake タスクをよく使うのは、レガシーなデータを転送したり、定期的な処理を実行したりする場合です。特にレガシーデータの転送はかなり面倒な処理になりがちです。なので、他のコードと同様に私はテストを書いて予期せぬ問題が起きないことを確認しています。

私の経験上、一番良いと思われるやり方は、Rake タスクに書いたコードを抽出してクラスやモジュールに移し、それからそのメソッドをタスクの中で呼び出すことです。たとえば次のような Rake タスクを想像してみてください。その Rake タスクはレガシーな `Person` クラスから本書で使っている `Conatct` クラスに情報を移動させます。これを手続き的なアプローチで書くとしたら次のようになります。

```
namespace :legacy do
  # Persons を Contacts に移し替える
  desc "Move Persons to Contacts"
  task person: :environment do
    Person.all.each do |person|
      Contact.create!(
        firstname:  person.firstname,
        lastname:   person.lastname,
        email:      person.email
      )
    end
  end
end
```

このような場合、私なら *lib/legacy.rb* に Legacy クラスを作り、タスクの中身をクラスメソッドへ移します。

```
class Legacy
  def self.move_people
    Person.all.each do |person|
      Contact.create!(
        firstname:  person.firstname,
        lastname:   person.lastname,
        email:      person.email
      )
    end
  end
end
```

そして元の Rake タスクを変更します。

```
namespace :legacy do
  # Persons を Contacts に移し替える
  desc "Move Persons to Contacts"
  task person: :environment do
    Legacy.move_people
  end
end
```

こうすれば簡単にこのタスクをテストできます。Legacy クラスをテストするだけで済むからです。アプリケーション側の構造に合わせて、まず *spec/lib* ディレクトリを作成し、そこへ *legacy_spec.rb* を追加しましょう。それからテストを書いてください。

```
require 'rails_helper'

describe Legacy do
  # person から contact を作成すること
  it 'creates a contact from a person'
  # etc.
end
```

本書で見てきたテクニックを使えば、他のコードと同じように Rake 関連のコードもテストすることができます。

まとめ

メールやファイルアップロード、タイムスタンプ、web サービス、API、ユーティリティタスクといった機能は、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。
- 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、時間に依存する機能はありますか？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。私たちはアドレス帳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 を練習するには考え方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。さあ、このサンプルアプリケーションで実践してみましょう！

本章の完成後のコードを見たい場合はサンプルソースの *11_tdd* ブランチをチェックアウトしてください。コマンドラインを使う場合は次のようにタイプします。

```
git checkout -b 11_tdd origin/11_tdd
```

本章を読みながら一緒にコードも書いていく場合は、第 9 章のブランチから始めてください。

```
git checkout -b 09_speedup origin/09_speedup
```

その他、詳細については第 1 章を読んでください。

フィーチャを定義する

本書ではずっと架空の会社のために連絡先管理アプリケーションを開発してきました。さて、その会社が今度はニュースリリースを投稿できる場所が欲しいと言ってきています。彼らのリクエストはアプリケーションのメニューバーにニュースリリース用のリンクを付けることです。演習問題はシンプルな方が良いので、仕様はたとえば次のようにしましょう。アカウントを持っているユーザーなら誰でもニュースリリースを投稿できます。そして、ゲストユーザーは投稿後すぐにそのニュースリリースが閲覧できます。

この仕様は二つの基本的なシナリオとして表現できます。

- ユーザーとして、私はニュースリリースを追加したい。その目的は当社の偉大さを世界に知らしめるためである。
- サイト訪問者（ゲスト）として、私はニュースリリースを読みたい。その目的はこの会社をもっと詳しく知るためである。

本章では一つ目のシナリオに取り組みます。二つ目のシナリオは演習問題としてあなた自身が取り組んでください。もちろん、これ以外のシナリオもいくつか挙げることができます。たとえば、「ニュースリリースの更新や削除」や「編集ワークフローの開発（公開前に管理者の承認が必須となる）」等です。これらのシナリオもあなたのために残しておきます。きっとあなたは練習問題がもっと欲しいと思っているでしょうから。

最初のストーリーを思い浮かべながら、さあ始めましょう。

まず Guard を起動してください。それから、必要であればリターンキーを押し、全部のスペックを実行してください。フィーチャの追加前から壊れているスペックはありませんか？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、新しいフィーチャスペックを作って作業のアウトラインを決めていきます。エディタを起動して、次のような新しいファイルを追加してください。

spec/features/news_releases_spec.rb

```
require 'rails_helper'

feature "News releases" do
  # ユーザーとして
  context "as a user" do
    # ニュースリリースを追加する
    scenario "adds a news release"
  end

  # ゲストとして
  context "as a guest" do
    # ニュースリリースを読む
    scenario "reads a news release"
  end
end
```

ファイルを保存してください。すると Guard が自動的にこのスペックを実行するはずです。あなたが予想した通り、今のところ RSpec は次のようなフィードバックを返すだけです。

```
2 examples, 0 failures, 2 pending
```

では最初のシナリオにステップを追加しましょう。

spec/features/news_releases_spec.rb

```
require 'rails_helper'

feature "News releases" do
  # ユーザーとして
  context "as a user" do
    # ニュースリリースを追加する
    scenario "adds a news release" do
      user = create(:user)
      sign_in(user)
      visit root_path
      click_link "News"

      expect(page).not_to have_content "BigCo switches to Rails"
      click_link "Add News Release"

      fill_in "Date", with: "2013-07-29"
      fill_in "Title", with: "BigCo switches to Rails"
      fill_in "Body",
        with: "BigCo has released a new website built with open source."
      click_button "Create News release"
```

```
21
22        expect(current_path).to eq news_releases_path
23        expect(page).to have_content "Successfully created news release."
24        expect(page).to have_content "2013-07-29: BigCo switches to Rails"
25      end
26    end
27
28    # ...
29  end
```

レッドからグリーンへ

ターミナルに戻って表示をチェックしてください。Guard はスペックが変更されたことに気付いて、そのスペックを自動的に実行したはずです。そしてテストは失敗してしまいました！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) News releases as a user adds a news release
   Failure/Error: click_link "News"
   Capybara::ElementNotFound:
     Unable to find link "News"
```

ではこの問題を修正して、先へ進みましょう。アプリケーションのレイアウトテンプレートを開き、見つからないと言われたリンクをナビゲーションバーに追加してください。

app/views/layouts/application.html.erb

```
<li><%= link_to "News", news_releases_path %></li>
```

ここで私たちは設計を二つ決めたことになります。一つは、ゲストがクリックするリンクは *News* と表示されることです。もう一つは、ニュースリリースの一覧を表示するための *news_releases_path* が必要になることです。こうした決めごとは通常、前もって決められます。そして、その内容を Capybara を使った自動テストに変換するのはあなたの仕事になります。

ターミナルに戻ってください。そして…いや、待ってください。ファイルが変更されたのに、どうしてフィーチャスペックは自動的に実行されなかったのでしょうか？その理由は Guard がそのように設定されていないからです。Guard は今変更したレイアウトファイルを監視していないので、ファイルが変更されてもスペックを実行しません。ここで考えられるのは次のような選択肢です。

- テンプレートを監視し、スペックが実行されるように `Guardfile` を編集する
- リターンキーを押し、遅いスペックも含めて すべての スペックを Guard に実行させる
- RSpec のタグ機能を利用して、*focus* タグが *true* になっているスペックのみを実行する

私は最後の選択肢が一番良いと思います。対象とするテストのレベルをあれこれ変更する場合は、タグを使うと便利です。実際、本章でもこれからこの機能を使っていきます。必要であれば第 9 章を確認して *spec/rails_helper.rb* に `filter_run` の設定を一行加えてください。

spec/rails_helper.rb

```
RSpec.configure do |config|
  # ...
  config.filter_run focus: true
end
```

それから focus タグを作業中のスペックに追加します。

spec/features/news_releases_spec.rb

```
feature "News releases", focus: true do
  # ...
```

新しい設定を追加したときは Guard がリロードされるまでしばらく待ってください。リロードが終わると作業中のシナリオだけが監視されるようになります。この場合、フィーチャファイルに focus タグを付けて保存すると、このスペックが実行されます。しかし、guard(main)> プロンプト内でリターンキーを押すと、今注目しているスペック だけ を実行させることができます。他のコンポーネントのスペックは実行されません。なお、この設定はフィーチャが完成したら元に戻します。

何らかの理由で Guard が RSpec の設定を自動的にリロードしない場合は、Guard のプロンプト内で reload と入力してください。

それでは続けましょう。ターミナルに戻ると、まだレッドのままです。しかし、こんどは別の理由で失敗しています。

```
1) News releases as a user adds a news release
   Failure/Error: sign_in(user)
   ActionView::Template::Error:
     undefined local variable or method `news_releases_path' for
     #<#<Class:0x007fb90d506b08>:0x007fb90d50e498>
```

news_releases_path というルーティングをまだ定義していないので、RSpec の実行中に Rails に怒られました。ここでまた決めなければいけません。さあ、私たちは…

- アプリケーションにこのルーティングを明示的に追加すべきでしょうか？これはスペックをパスさせるために一番簡単な方法です。
- Rails の scaffold 機能を使ってルーティングを追加すべきでしょうか？こうすると、使わないかもしれない他のコードも一緒に作成されます。

このケースなら私は二番目の選択肢を選びます。なぜなら、このフィーチャを最終的に完成させるためには、ニュースリリースの一覧を表示するだけでなく、ニュースリリースの表示や追加、編集、削除も必要になるとわかっているからです。scaffold を使えばすべてのアクションに対して堅牢なコードが作成され、そこから作業を始められます。そしてお約束のコードを手書きする時間が減って、アプリケーションを改善する時間に回せます。

必要に応じて新しいターミナルを開いてアプリケーションへ移動し、scaffold を作成してください。

```
$ bin/rails g scaffold news_release title released_on:date body:text
```

作成されたファイルの中には使うファイルと使わないファイルがありますが、とりあえず実行結果を確認してください。

```
invoke    rspec
create      spec/models/news_release_spec.rb
invoke      factory_girl
create        spec/factories/news_releases.rb
...
invoke    rspec
create      spec/controllers/news_releases_controller_spec.rb
```

第 2 章で scaffold ジェネレータを設定したことを思い出してください。scaffold を作成すると、モデルスペックとコントローラスペック、それにこのあとの開発で使うファクトリも一緒に作成されます。

Guard に戻り、リターンキーを押してスペックを実行してください。Rails 4.1 はマイグレーションをまだ実行していないことを検知するはずです。なので、マイグレーションを実行しましょう。

```
$ bin/rake db:migrate
```

そしてまた別の失敗です。しかし、信じてもらえないかもしれませんが、私たちは確実に前へ進んでいます。

```
Failures:

  1) News releases as a user adds a news release
     Failure/Error: click_link "Add News Release"
     Capybara::ElementNotFound:
       Unable to find link "Add News Release"
     # ./spec/features/news_releases_spec.rb:12:in \
       `block (3 levels) in <top (required)>'
     # -e:1:in `<main>'
```

シナリオを見ていきながら、失敗した場所を確認しましょう。失敗したのはナビゲーションバーにある *News* リンクをクリックしたときよりも あと です。さらに、表示されたページ（*news_releases_path* で表示されたページです）にニュースリリースのタイトル（まだ追加していません）が含まれていないことを RSpec が確認した あと に失敗しています。この新しい失敗の原因はニュースリリース用の *index* テンプレートのどこかにあると考えて間違いないでしょう。このファイルを開いてみるとやはり、ニュースリリース作成用のリンクが scaffold で作られたときのままになっています。これはスペックの中で書いた文言と異なります。ではこの問題を修正しましょう。ついでに Bootstrap 用のスタイルも追加します。

app/views/news_releases/index.html.erb

```
<%= link_to 'Add News Release', new_news_release_path,
  class: 'btn btn-primary' %>
```

次のエラーでは *Date* のテキストフィールドが見つからないと表示されています。こちらも簡単に直せます。

app/views/news_releases/_form.html.erb

```
<%= f.label :released_on, 'Date' %><br>
<%= f.text_field :released_on %>
```

さあこれでフォームに関するスペックはすべてパスしたようです。次のエラーを見てみると、ニュースリリースの投稿が終わったあとのリダイレクト先を適切に設定する必要があるようです。

```
1) News releases as a user adds a news release
   Failure/Error: expect(current_path).to eq news_releases_path

     expected: "/news_releases"
          got: "/news_releases/1"

     (compared using ==)
   # ./spec/features/news_releases_spec.rb:18:in `block (3 levels)
     in <top (required)>'
```

これは私たちが望んでいるパスと scaffold が設定したデフォルトのパスが異なるせいです。これはコントローラを修正すれば簡単に修正できます。

app/controllers/news_releases_controller.rb

```
def create
  @news_release = NewsRelease.new(news_release_params)

  respond_to do |format|
    if @news_release.save
      format.html { redirect_to news_releases_url,
          notice: 'News release was successfully created.' }
  # ...
```

そろそろ終わりに近づいてきました。次に失敗しているステップに対処するため、:notice シンボルに渡している値を *Successfully created news release.* に変更してください。

app/controllers/news_releases_controller.rb

```
def create
  @news_release = NewsRelease.new(news_release_params)

  respond_to do |format|
    if @news_release.save
      format.html { redirect_to news_releases_url,
          notice: 'Successfully created news release.' }
  # ...
```

あと 1 ステップでこのシナリオのテストが完了します。ここもパスさせましょう!

```
Failures:

  1) News releases as a user adds a news release
     Failure/Error: expect(page).to have_content \
       "2013-07-29: BigCo switches to Rails"
     # ...
     # ./spec/features/news_releases_spec.rb:20:in
       `block (3 levels) in <top (required)>'
```

これをパスさせる一番簡単な方法はビューのテンプレートにコードをちょっと追加することです。すぐあとでリファクタリングの方法を考えるので、それまではいったんこの実装で済ませます。

app/views/news_releases/index.html.erb

```
<h1>News releases</h1>

<ul>
  <% @news_releases.each do |news_release| %>
    <li>
      <%= link_to "#{news_release.released_on.strftime('%Y-%m-%d')}:
        #{news_release.title}", news_release %>
    </li>
  <% end %>
</ul>

<p>
  <%= link_to 'Add News Release', new_news_release_path,
    class: 'btn btn-primary' %>
</p>
```

さあこれで最初のシナリオがすべてパスしました!

よくがんばりましたね。ですが、これでおしまい、ではありません。きれいにできる部分がまだ残っているからです。私たちは レッド - グリーン - リファクタ の リファクタ ステージにやってきました。リファクタリングもやろうと思えばとことんやれるのですが、ここでは比較的シンプルなリファクタリングにとどめます。本格的なリファクタリングは本書のスコープを超えてしまうからです。

少なくとも一箇所は修正できます。前述の通り、さっき追加した個々のニュースリリースを表示するリンクがとても不格好です。簡単にリファクタリングする方法はこれを NewsRelease モデルに移動させることです。もちろん、最初に NewsRelease モデルの簡単なテストを書かなければなりません。

spec/models/news_release_spec.rb

```
require 'rails_helper'

describe NewsRelease, type: :model, focus: true do
  # フォーマットされた日付とタイトルの文字列を返すこと
  it "returns the formatted date and title as a string" do
    news_release = NewsRelease.new(
      released_on: '2013-07-31',
      title: 'BigCo hires new CEO')
    expect(news_release.title_with_date).to \
      eq '2013-07-31: BigCo hires new CEO'
  end
end
```

それからモデルを変更してこのテストをパスさせます。

app/models/news_release.rb

```
class NewsRelease < ActiveRecord::Base
  def title_with_date
    "#{released_on.strftime('%Y-%m-%d')}: #{title}"
  end
end
```

続いてビューを変更し、この新しいメソッドを使うようにします。

app/views/news_releases/index.html.erb

```
<%= link_to news_release.title_with_date, news_release %>
```

そしてテストはグリーンのままになっています。これがリファクタリングステップのキーポイントです。つまり、どんな変更をを加えたとしても、テストはパスし続けなければなりません。リファクタリングするときは、フィーチャスペックからモデルに移動したり、コントローラに移動したり、はたまたライブラリに移動したり、いろんな場所を行き来することになると思います。どうなるかはあなたのアプリケーション次第です。そのコードが最も意味を持つ場所を探してください。

NewsRelease モデル用のスペックを開いているので、ついでにこのモデルのその他の要件についても少し考えてみましょう。バリデーションの要件がちょっと気になってきました。具体的に言うと、リリース日やタイトル、本文のないニュースリリースはほとんど役に立ちません。ではエクスペクテーションをいくつか追加してみましょう。ここでは Shoudla（第 9 章を参照）が提供するカスタムマッチャを使っています。

spec/models/news_release_spec.rb

```
it { should validate_presence_of :released_on }
it { should validate_presence_of :title }
it { should validate_presence_of :body }
```

NewsRelease モデルにバリデーションを追加してこのテストをパスさせてください。続きはまだあります。

そういえば、まだ注目していない新しい機能がもう一つありました。それは認証機能です。ユーザーがログインしてニュースリリースを追加することは問題ありません。しかし、それがゲストだったらどうでしょうか？当たり前ですが、テストユーザーを作ってログインしている 2 行をシナリオからコメントアウトしても、テストはまだパスします。これはよくありません。

当然、ニュースリリースコントローラに認証フィルタを追加する必要が出てきます。

入念にコントローラの全メソッドをテストするのではなく、ここでは重要なポイントだけにフォーカスしましょう。確認したい点はゲストが `new` と `create` のメソッドにアクセスできないことです。そこでちょっと前に scaffold が作成したコントローラスペックに着目しましょう。scaffold は余分なコードをたくさん生成しますが、それを全部消して次のように書き換えます。

spec/controllers/news_releases_controller_spec.rb

```
require 'rails_helper'

describe NewsReleasesController, focus: true do
  describe 'GET #new' do
    # ログインを要求すること
    it "requires login" do
      get :new
      expect(response).to require_login
    end
  end

  describe "POST #create" do
    # ログインを要求すること
    it "requires login" do
      post :create, news_release: attributes_for(:news_release)
      expect(response).to require_login
    end
  end
end
```

require_login は第 7 章で作成したカスタムマッチャです。

次に、二つ目のエクスペクテーションがパスするようにファクトリを追加します。

spec/factories/news_releases.rb

```
require 'faker'

FactoryGirl.define do
  factory :news_release do
    title "Test news release"
    released_on 1.day.ago
    body { Faker::Lorem.paragraph }
  end
end
```

スペックは失敗します。

```
Failures:

  1) NewsReleasesController GET #new requires login
     Failure/Error: expect(response).to require_login
       expected to require login to access the method
     # ./spec/controllers/news_releases_controller_spec.rb:7:in
       `block (3 levels) in <top (required)>'
     # -e:1:in `<main>'

  2) NewsReleasesController POST #create requires login
     Failure/Error: expect(response).to require_login
       expected to require login to access the method
     # ./spec/controllers/news_releases_controller_spec.rb:14:in
       `block (3 levels) in <top (required)>'
     # -e:1:in `<main>'
```

そこでゲストがアクセスできないように認証フィルタを追加し、スペックをパスさせます。

app/controllers/news_releases_controller.rb

```
class NewsReleasesController < ApplicationController
  before_action :authenticate, except: [:index, :show]
  # ...
```

クリーンアップ

テストはすべてパスしています。新しいフィーチャも実装できました。ですが、本章を締めくくる前に、クリーンアップすべきコードが少し残っています。このケースでは scaffold を使うと最初に決めたので、さしあたって使うことのない余分なコードがたくさん追加されています。こうしたコードは削除した方が良いでしょう。

追加したり変更したりした ファイルの中には、scaffold で作られてから全く変更していないものものあります。たとえば次のようなファイルです。

- *app/assets/javascripts/news_releases.js.coffee*
- *app/assets/stylesheets/news_releases.css.scss*
- *app/helpers/news_releases_helper.rb*

これらのファイルは使われていないので削除しましょう。必要になったらいつでもまた追加できます。

最初から上記のファイルを作成したくなければ、*config/application.rb* 内でジェネレータの設定をするとファイルの作成をスキップできます。

```
config.generators do |g|
  g.assets false
  g.helper false
  g.javascripts false
  g.stylesheets false
  # etc.
end
```

覚えている方も多いと思いますが、第 2 章でも RSpec 関連のジェネレータを同じように設定しています。

最後に、今回追加した新機能が既存の機能を壊していないことを確認する必要があります。新しく作ったスペックから `focus` タグを外してください。対象となるファイルは *spec/features/news_releases_spec.rb* 、*spec/models/news_release_spec.rb* 、*spec/controllers/news_releases_controller_spec.rb* の三つです。それから、*spec/rails_helper* のタグフィルタも削除してください。

spec/rails_helper.rb

```
# この行を削除するか、もしくは最低限コメントアウトしてください
# config.filter_run focus: true
```

Guard が起動しているターミナル画面に戻ってください。それから `<return>` キーを押して全部のテストを実行しましょう。大丈夫そうですね！全部のスペックがパスしています。ただし、ゲストにニュースリリースを読んでもらうようにするスペックだけは保留中になっています。そしてこのスペックはこれからあなたが対応します。そうですよね？

もう一点。テストスイートの中では全部うまくいっているように見えたとしても、コミット前に（さらにデプロイ前は必ず）ブラウザを使った動作確認を忘れずに実施してください。また、可能であれば他の人にも変更点を確認してもらってください。なぜなら、そういう確認をしないと気付かないような問題もよくあるからです。実際、次のシナリオでちょっと目立つ問題が発生しています。ニュースリリース一覧を開いてみてください。*Add News Release* ボタンがゲストにもログイン中のユーザーにも同じように見えてしまっています！次のシナリオを対応する際、たぶんあなたは `expect(page).not_to have_content 'Add News Release'`（ページに'Add News Release' と表示してほしくない）と考えるでしょう（ヒント）。

というのは冗談です。GitHub にある本章のソースをチェックアウトすると、あなたのために私が書いておいたシナリオが見つかると思います。これをパスさせてください。

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、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

- もしあなたがここまでサンプルコードを一緒にやってきているのなら、次のシナリオに進んで実際にやってみてください。保留中のシナリオに私が書いておいたステップがあるので、それをパスさせてください！必要があれば他のレベルのスペックを作ることも忘れないでください。つまり、コントローラレベル、もしくはモデルレベルでテストした方が簡単になりそうな追加のテストケースはないでしょうか？
- さらに練習するため、あなたが実装できそうな他の一般的な機能も考えてください。たとえば、ニュースリリースの編集と削除はどうですか？

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、コントローラレベルのスペックなど、こうした内容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外す（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（先にテストなしでスパイクを作る場合も含みます）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。

できるだけフィーチャスペックから書き始めてください

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、フィーチャ、と進んでいくのではなく、まずフィーチャスペックから 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。

フィーチャ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。よくできたフィーチャ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、フィーチャ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丸一日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（特にフィーチャスペックやモック、スタブ）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。

そんな人々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 8 章で見たような JavaScript を実行するフィーチャ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、使い捨ての Rails アプリケーションはこうした目的に打って付けです。新しいアプリ(たとえばブログアプリや To-do リストなど)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 To-do リストを改造して、ボタンをクリックしたらプロジェクトのタスクをプロジェクトオーナーに送信するようにしてください。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。

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 Girl、Capybara、それに DatabaseCleaner。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: http://facebook.com/everydayrails
- GitHub: https://github.com/everydayrails/rails-4-1-rspec-3-0/issues
- 日本語版に関するフィードバック: https://leanpub.com/everydayrailsrspec-jp/feedback

それから、Everyday Rails blog(http://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。

Aaron

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、次に挙げる情報源すべて私がこれまで読んできたものです。どの情報源もあなたが 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。

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 Relish のドキュメントを読んでみましょう。最新の RSpec からバージョン 2.13 までのドキュメントを読むことができます。https://www.relishapp.com/rspec

Better Specs

Better Specs には本当に素晴らし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。(訳注: 日本語の翻訳ページもあります)http://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。http://beta.pluralsight.com/courses/rspec-the-right-way

The RSpec Book

RSpec の元開発リーダーである David Chelimsky によって書かれたこの本は、RSpec とそれを取り巻く関連ツール全般について説明しています。ただし私としては、あなたが RSpec の基礎を一通り理解したあとか、Rails 以外の場所で RSpec を使いたいと思ったときに読むことをお薦めします。使われているコードはちょっと古くなっていますが、基本的な考え方は同じです。http://pragprog.com/book/achbd/the-rspec-book / 日本語版: http://books.shoeisha.co.jp/book/b94964.html

Railscasts

Railscasts を知らないという Rails 開発者は誰一人としていないでしょう。Ryan Bates が提供しているこのスクリーンキャストはどれも一級品です。Ryan はテストに関するエピソードもたくさん作っています。RSpec にフォーカスしたエピソードもたくさんありますし、大きな演習問題の一部として RSpec が登場している場合もあります。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

Code School

Code School の *Testing with RSpec* は動画とハンズオンタイプのチュートリアルを組み合わせた教育コースになっています。このコースには RSpec の設定方法や、フックとタグ、モックとスタブ、カスタムマッチャ等を学習するためのコンテンツが含まれています。Rails がデフォルトで提供するテストフレームワークを見てみたい方は、*Rails Testing for Zombies* をチェックしてください。http://www.codeschool.com/courses/

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。http://groups.google.com/group/rspec

Rails のテスト

Rails 4 Test Prescriptions: Keeping Your Application Healthy

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。Rails 4 と RSpec 3 に対応した改訂版もリリースされています。https://pragprog.com/book/nrtest2/rails-4-test-prescriptions

Rubyconf 2012 で発表された *Testing Should Be Fun* という Noel の講演も非常に素晴らしいです。あなたのテストスイートが遅くなったり管理しにくくなってきたら、いや、むしろそうなってしまう前にこの講演を見て問題を回避してください。ビデオは http://confreaks.com/videos/1306-rubyconf2012-testing-should-be-fun に、スライドは https://speakerdeck.com/noelrap/testing-should-be-fun にあります。

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。http://ruby.railstutorial.org / 日本語版: http://railstutorial.jp

Rails によるアジャイル Web アプリケーション開発

Sam Ruby（それに Dave Thomas と David Heinemeier-Hansson）によって書かれた Rails によるアジャイル Web アプリケーション開発（原題: *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。初版を読み返すと、テストはおまけのように扱われていましたが、最近の版では開発プロセスにテストが組みこまれており、ずいぶん良くなりました。http://pragprog.com/book/rails4/agile-web-development-with-rails / 日本語版: http://estore.ohmsha.co.jp/titles/978427406866P

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。http://learn-rails.com

[61] https://www.railstutorial.org/book/static_pages#aside-when_to_test

訳者あとがき

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！

秋元利春

Rails & RSpec の学習に今から取り組む方にはぜひ読んでほしいと思える一冊に仕上がりました。翻訳チームのメンバーも本書で RSpec を学んでおり、非常に良い学習体験を与えてくれる一冊だと思います。そんな本書の日本語版翻訳に携わることができて本当に嬉しいです。Aaron さん素晴らしい本をありがとう。そしてこの本を手に取ってくれたあなたに感謝します。あなたに楽しい Ruby Life が訪れますように。

翻訳作業は想像していた以上に大変な作業でした。翻訳を終えてから過去の技術書を振りかえると、読みやすく伝わりやすい文体で私達の元へ届ける努力をしてくれていた翻訳者の方々の苦労がわかり、彼らには本当に頭が上がらないなと感じました。

とはいえ、確かに大変な作業ではありましたが、得られるものも多かったです。技術的な学習はもちろん、より深く文章の意図やニュアンス、文体のリズムに想いを馳せるきっかけになりました。本当に貴重な経験になりました。

正式版リリース準備に追われて勉強会の最中に翻訳作業にいそしむ私を温かく見守ってくれただけでなく、翻訳に関する相談にも乗ってくれた神戸.rb[62]メンバーのみんな、そして原文解釈に悩んだ際に助けてくれた西脇.rb[63]のマイケルに感謝します。

この翻訳チームの皆さんと一緒に翻訳作業ができて良い経験になりました。翻訳作業自体も大変でしたが、それに付随する諸々のタスクに追われた時にはチームメンバーでサポートしながら問題なく翻訳作業を進められました。伊藤淳一さん、魚振江さんには本当に感謝しています。

最後に。本業や翻訳、コミュニティ活動で忙しくなる私を精神面、体調面で支えてくれている妻と愛猫に感謝します。いつもありがとう。

[62] https://www.facebook.com/nshgrb
[63] http://nishiwaki-higashinadarb.doorkeeper.jp/

魚振江

今回、日本語版の翻訳チームに参加できて、そして無事にリリースできてすごく良かったです。これまでの道のりを振り返ってみると、私にとってこの翻訳プロジェクトはなかなか大変なものでした。

まず、私は本書の中国語版をすでに読んでいました。読んだ当時はまだ日本語版がなかったので、本書を日本語で翻訳したいとぼんやりと考えていました。その後、いつも拝読している伊藤さんのブログ[64]で英語版の紹介がたびたび出てきて、比較的大きな反響を得ているのを目にしました。そこで実際に翻訳してみようという気持ちになり、2013 年 10 月上旬頃、著者の Aaron さんに連絡して翻訳を開始しました。しかし、早くも第 1 章から長文和訳に苦戦し始め、一人では翻訳作業を思うように進められませんでした。2013 年 11 月頃、伊藤さんに声を掛けて頂き、伊藤さんがプロジェクトをリードする形で翻訳を進めてきました。また、12 月から Aki さんも参加してもらい、いろいろと助けていただきました。

このように振り返ってみると、伊藤さんと Aki さんの助けなしでは日本語版の翻訳は全くできなかったと思います。お二人には大変感謝しています。そして、私たちの手で本書の翻訳を完了させたことが、少しでも読者のみなさんのお役に立てば嬉しいです。

[64] http://blog.jnito.com/

日本語版の謝辞

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[65]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[66]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[67]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[68]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。

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[69]、西川茂伸さん[70]、遠藤大介さん[71]にも非常に助けられました。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[72]のイギリス人プログラマ、マイケル(P. Michael Holland)[73]はほとんど 4 人目の翻訳者と呼んでも差し支えないぐらいの活躍をしてもらいました。もしマイケル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

---

[65] https://twitter.com/dentomo
[66] https://twitter.com/daipresents
[67] https://leanpub.com
[68] https://twitter.com/thc_o0
[69] https://twitter.com/joker1007
[70] https://twitter.com/shishi4tw
[71] https://twitter.com/ruzia
[72] http://nishiwaki-higashinadarb.doorkeeper.jp/
[73] 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 gem 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。http://everydayrails.com/

著者について

Aaron Sumner は Django の国の中心部に住んでいる Ruby 開発者です（訳注: Django は Python で実装された web フレームワーク。筆者の住むカンザス州で生まれた）。

彼は 1990 年代の半ばから 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（本気です）、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）、カレッジバスケットボール（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、カンザスの田舎に住んでいます。

Aaron の個人ブログは http://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[74]に勤務する Rails プログラマ。Ruby コミュニティ西脇.rb[75]の主催者として神戸近辺で Ruby 勉強会も開催している。個人ブログ give IT a try[76]で技術情報や日常のよもやま話等も発信中。Twitter アカウントは@jnchito[77]。

秋元利春

神戸・京都で活動するフリーランス Rails プログラマ。Ruby への恩返しも兼ねてコミュニティ活動に勤しむ。神戸で神戸.rb[78]や Kobe Rubyist Meetup[79]を主催。RailsGirls Kyoto[80]の organizer, coach も務める。Twitter アカウントは@spring_aki[81]。

魚振江

Rails プログラマ。Yokohama.rb[82]と東京近くの Ruby/Rails コミュニティで活動しています。Twitter アカウントは@blueplanet42[83]。

---

[74] http://www.sonicgarden.jp
[75] http://nishiwaki-higashinadarb.doorkeeper.jp/
[76] http://blog.jnito.com
[77] https://twitter.com/jnchito
[78] http://koberb.doorkeeper.jp/
[79] http://koberubyistmeetup.doorkeeper.jp/
[80] http://railsgirls.com/kyoto
[81] https://twitter.com/spring_aki
[82] http://yokohamarb.doorkeeper.jp/
[83] https://twitter.com/blueplanet42

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[84]は iStockphoto の投稿者であるHabman_18[85]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。

[84] http://www.istockphoto.com/stock-photo-1607117l-old-truck-in-early-morning-light.php?st=1e7555f

[85] http://www.istockphoto.com/user_view.php?id=4151137

変更履歴

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）。