2019年度「専修学校による地域産業中核的人材養成事業」

Ruby on Railsを利用した アジャイル型システム開発教材

札幌（北海道）をモデルとした地域創生のためのIT人材育成と企業連携推進事業

2019年度「専修学校による地域産業中核的人材養成事業」

Ruby on Railsを利用した
アジャイル型システム開発教材

札幌（北海道）をモデルとした地域創生のための IT 人材育成と企業連携推進事業

目次

第１章　アジャイル開発 14

1.1 ウォーターフォール型開発 14

1.2 ウォーターフォール・モデルワークショップ 19

1.3 グループディスカッション 22

1.4 アジャイルソフトウェア開発基礎 23

1.5 アジャイルの各種プラクティス 37

1.6 アジャイルのワークショップ　～飛べ！紙飛行機！～ 45

1.7 クループディスカッション 54

1.8 アジャイルでのプロジェクト管理 55

1.9 アジャイル開発を支援するツール 67

第２章　Ruby on Rails 開発環境の準備 80

2.1 Ruby の基本的な構文の復習 80

2.1.1 数字を順番に出力する 80

2.1.2 FizzBuzz 問題 81

2.1.3 メソッドを定義する 83

2.1.4 Integer クラスを拡張する 84

2.1.5 まとめ 84

2.1.6 発展課題 84

2.2 Ruby on Rails：開発環境の構築 85

2.2.1 macOS 用セットアップ(10.9 から 10.14) 85

2.2.2 Windows 用セットアップ(WSL が利用できる Windows 10 64bit) 87

2.2.3 Windows 用セットアップ(Windows 7 や 32bit の Windows の場合等) 90

2.2.4 Linux 用セットアップ(Ubuntu) ... 94
2.2.5 仮想環境(VirtualBox) ... 95
2.2.6 クラウドサービス(AWS Cloud9) ... 99
2.3 Git 基礎 1 ... 103
2.3.1 Git ... 103
2.3.2 GitHub ... 106
2.3.3 Git の利用 ... 108
2.4 Git 基礎 2 ... 115
2.4.1 ワーキングディレクトリと作業ブランチについて ... 115
2.4.2 作業のやりなおしについて ... 115
2.4.3 GitHub Flow とは ... 116
第３章　テスト駆動型開発 ... 122
3.1 Ruby: 自動テスト ... 122
3.1.1 なぜテストを自動化するのか ... 122
3.1.2 テストケースを考える ... 123
3.1.3 テストしやすい設計 ... 123
3.1.4 テストを書いてみる ... 124
3.1.5 仕様書としてのテストコード ... 124
3.1.6 すべてをテストすることはできない ... 125
3.1.7 自動テストとアジャイル開発 ... 126
3.1.8 まとめ ... 126
3.1.9 発展課題 ... 127
3.2 Ruby: 自動テスト（minitest） ... 128

3.2.1 概要 128
3.2.2 テストの書き方 128
3.2.3 テストの実行結果を読む 130
3.2.4 テストケースの実行順 131
3.2.5 さまざまなアサーションメソッド 132
3.2.6 テストケースに影響を及ぼす機能 134
3.2.7 まとめ 134
3.2.8 発展課題 135
3.3 Ruby: 自動テスト（RSpec） 136
3.3.1 概要 136
3.3.2 gem のインストール 136
3.3.3 Array クラスのテストを書く 137
3.3.4 テストケースを失敗させる 138
3.3.5 テストケースを整理する 140
3.3.6 さまざまなマッチャー 141
3.3.7 テストケースに影響を及ぼす機能 143
3.3.8 まとめ 144
3.3.9 発展課題 145
3.4 テスト駆動開発 146
3.4.1 概要 146
3.4.2 実例 147
3.4.3 ふりかえり 156

3.4.4 テスト駆動開発の意義 156
3.4.5 テスト駆動開発の限界 157
3.4.6 まとめ 157
3.4.7 発展課題 158
3.4.8 参考資料 158
3.5[評価課題]自動テストとテスト駆動開発 158
3.5.1 自動テスト 158
3.5.2 テスティングフレームワーク 159
3.5.3 テスト駆動開発 160
第４章　Ruby on Rails 基礎 164
4.1 Ruby on Rails：Web システム概念 164
4.1.1 インターネットの概要 164
4.1.2 HTTP プロトコルと Web サーバの概要 164
4.1.3 Web ブラウザ、HTML,JavaScript の概要 166
4.2 Ruby on Rails:View の機能 168
4.2.1 SASS/SCSS とは 168
4.2.2 SASS と SCSS の違い 168
4.2.3 SCSS の書き方 169
4.2.4 ERB とは 172
4.2.5 ERB の書き方 172
4.2.6 Rails プロジェクトでの SCSS と ERB の使い方 172
4.3 Ruby on Rails：Rails 基礎 1 174
4.3.1 Ruby on Rails について 174

4.3.2 Cloud9 上で Rails アプリケーションを作成......175
4.4 Ruby on Rails：Rails 基礎 2......179
4.4.1 アプリの構造解説......179
4.4.2 Rails の基本......180
4.5 Ruby on Rails：Rails 基礎 3......184
4.6 [評価課題]Rails 基礎......190
4.6.1 Rails のディレクトリ構成......190
4.6.2 StrongParameter......191
第５章　Ruby on Rails デザイン......194
5.1 Ruby on Rails：Gem とは......194
5.1.1 bundler について......194
5.1.2 Gem の構成とコードリーディング......195
5.1.3 よく使われる Gem の紹介......197
5.2 Ruby on Rails：デザインテンプレート......199
5.2.1 Bootstrap インストール......199
5.2.2 デザインの適用......200
5.2.3 国際化対応（I18n）......204
第６章　Ruby on Rails アジャイル開発......214
6.1 Ruby on Rails：データベース設計、多対多の関連付け......214
6.1.1 ActiveRecord とは......214
6.1.2 データベース設計の基礎......214
6.1.3 多対多の関連付け......215
6.1.4 ActiveRecord の代表的なメソッド......215

6.1.5 例題 218

6.1.6 問題 222

6.2 Ruby on Rails: ActiveRecord の応用 223

6.2.1 「N+1 問題」とは 223

6.2.2 「N+1 問題」解消方法 226

6.2.3 サブクエリの作り方 227

6.3 プロダクトバックログを見積もる 230

6.4 スプリントバックログの作成 232

第７章　Ruby on Rails テスト 236

7.1 Ruby on Rails：Rails テスト基礎 １ 236

7.1.1 テスティングツールについて 236

7.1.2 RSpec の環境構築 237

7.2 Ruby on Rails：Rails テスト基礎 2 240

7.2.1 RSpec のテンプレート作成 240

7.2.2 RSpec の実行方法 242

7.3 Ruby on Rails：Rails テスト基礎 3 245

7.3.1 例題：テストの実装 245

7.4 [評価課題]Rails テスト基礎 251

第８章　Ruby on Rails EC **サイト開発 1** 256

8.1 Ruby on Rails：EC サイトの開発 商品一覧 1 256

8.1.1 商品一覧の作成 256

8.1.2 EC サイトの概要とタイムゾーンの設定 256

8.1.3 画面遷移とルーティングの設定 258

8.2 Ruby on Rails：EC サイトの開発 商品一覧 2 260

8.2.1 画面遷移とルーティングの実装 260
8.3 Ruby on Rails：EC サイトの開発 商品一覧 3 262
8.3.1 データベース設計、多対多の関連付け 262
8.3.2 ActiveRecord とは 262
8.3.3 データベース設計の基礎 262
8.3.4 多対多の関連付け 263
8.3.5 ActiveRecord の代表的なメソッド 263
8.3.6 中間テーブルへのデータ登録設定 270
8.4 Ruby on Rails：EC サイトの開発 商品一覧 4 271
8.4.1 中間テーブルのデータ表示と保存の処理 271
8.5 Ruby on Rails: バリデーションとフォームヘルパー 274
8.5.1 バリデーション(validation:検証)とは 274
8.5.3 バリデーション実行時の動作を確認してみる 279
8.5.4 form_with ヘルパーとは 280
8.5.5 form_with ヘルパーの使い方 283
8.6 ログイン認証 284
8.5.1 ユーザー登録機能作成 284
8.6.2 SessionController を作成する 288
8.6.3 ルーティングを定義する 289
8.6.4 ログインフォームを作る 290
8.6.5 入力されたユーザー情報で認証する 292
8.6.6 ログアウトする 293
8.7 ログイン認証とユーザー管理 294

8.7.1 ログイン認証 294
8.7.2 Devise とは 294
8.7.3 Devise のインストール 295
8.7.4 ログイン処理の作成の流れ 295
8.7.5 例題 296
8.7.6 問題 305
8.8 セッション管理 306
8.8.1 セッションストア 306
8.8.2 セッションにアクセスする 307
8.8.3 Flash を使用する 307
8.9 Ruby on Rails：画像アップロード 309
8.9.1 画像アップロード 309
8.9.2 Rails における画像アップロードの概要 309
8.9.3 Active Storage を利用した画像アップロード(Rails5.2 から) 310
8.9.3 画像のリサイズ 313
8.9.4 アップロード機能の自動テスト 314
8.10 [評価課題]EC サイトの開発 1 の課題実施の評価 319
第９章　Ruby on Rails EC **サイト開発２** 324
9.1 Ruby on Rails：EC サイトの開発 注文 324
9.1.1 確認画面 324
9.1.2 例題 324
9.1.3 問題 331
9.1.4 テストの追加 331

9.1.5　まとめ......335

9.2 Ruby on Rails：EC サイトの開発 メール送信......336

9.2.1 メール送信のプロトコルの概要......336

9.2.2 メール送信......336

9.2.3 ActionMailer とは......337

9.2.4 例題......337

9.2.5 問題......342

9.3 [評価課題]EC サイトのメール送信課題......343

第 10 章　Ruby on Rails EC **サイト開発３**......348

10.1 Ruby on Rails：EC サイトの開発 enum/状態遷移 1......348

10.1.1 状態遷移とは......348

10.1.2 例題......349

10.1.3 問題......354

10.2 Ruby on Rails：EC サイトの開発 セッションと複数商品の注文......355

10.2.1 セッション機能について......355

10.2.2 セッションを利用したカート機能の実装......356

10.2.3 複数明細の購入・管理への変更......362

10.2.4 複数明細の管理......363

10.3 [評価課題]セッション管理......364

10.4 Ruby on Rails：gem を使わない検索......365

10.4.1 検索フォームの作成......365

10.4.2 検索処理の作成......369

10.5 Ruby on Rails：EC サイトの開発 検索......372

10.5.1 ransack のしくみ 372
10.5.2 ransack の使い方 372
10.6 [評価課題]EC サイトの検索機能.. 380
10.7 Ruby on Rails：EC サイトの開発 Heroku へのデプロイ 383
10.7.1 Heroku について 383
10.7.2 Heroku へのリリース準備 384
10.7.3 Heroku への公開 385
10.8 Ruby on Rails：EC サイトの開発 ～スプリント＃１ふりかえり～ 392
10.9 Ruby on Rails：まとめ.. 399
10.9.1 Ruby on Rails のコマンド 399
10.9.2 Ruby on Rails のプロジェクトのディレクトリ構成 399
10.9.3 ルーティングの設定方法 401
10.9.4 ActionController.. 402
10.9.5 ActiveRecord.. 404
10.9.6 ビュー.. 405
10.9.7 Heroku へのデプロイ方法.. 406
10.10 [評価課題]Ruby on Rails　まとめ.. 407

第１章　アジャイル開発

第１章　アジャイル開発

1.1 ウォーターフォール型開発

ウォーターフォール型開発

ウォーターフォール・モデル

プロジェクトによって工程の定義に差はあるが、
開発プロジェクトを時系列に、
「要求定義」「外部設計（概要設計）」「内部設計（詳細設計）」
「開発（プログラミング）」「テスト」「運用」などの作業工程
（局面、フェーズ）にトップダウンで分割する。

線表（ガントチャート）を使用してこれらの工程を一度で終わらせる計画を立て進捗管理をする。

原則として前工程が完了しないと次工程に進まない（設計中にプログラミングを開始するなどの並行作業は行わない）事で、
前工程の成果物の品質を確保し、前工程への後戻り（手戻り）を最小限にする。

Wikipedia contributors. "ウォーターフォール・モデル" Wikipedia. Wikipedia, 27 Sep. 2019. Web. 27 Sep. 2019.
～ https://ja.wikipedia.org/wiki/ウォーターフォール・モデル より ～

ウォーターフォール・モデルの各工程

要件定義（要求分析）

システム全体の機能を決定する。要件定義書が作成される。

外部設計（概要設計）

仕様を決める。成果物は外部設計書、画面仕様書、インタフェース仕様書など。

内部設計（詳細設計）

設計する。成果物は内部設計書、データフロー図、ER図など。

開発（プログラミング）

プログラムを作成する。

テスト

作成したプログラムが正常に動作するかテストする。
成果物は、単体テスト報告書、システムテスト報告書。

運用・保守

完成したシステム対する問い合わせ対応や監視、障害対応などを行う。

歴史

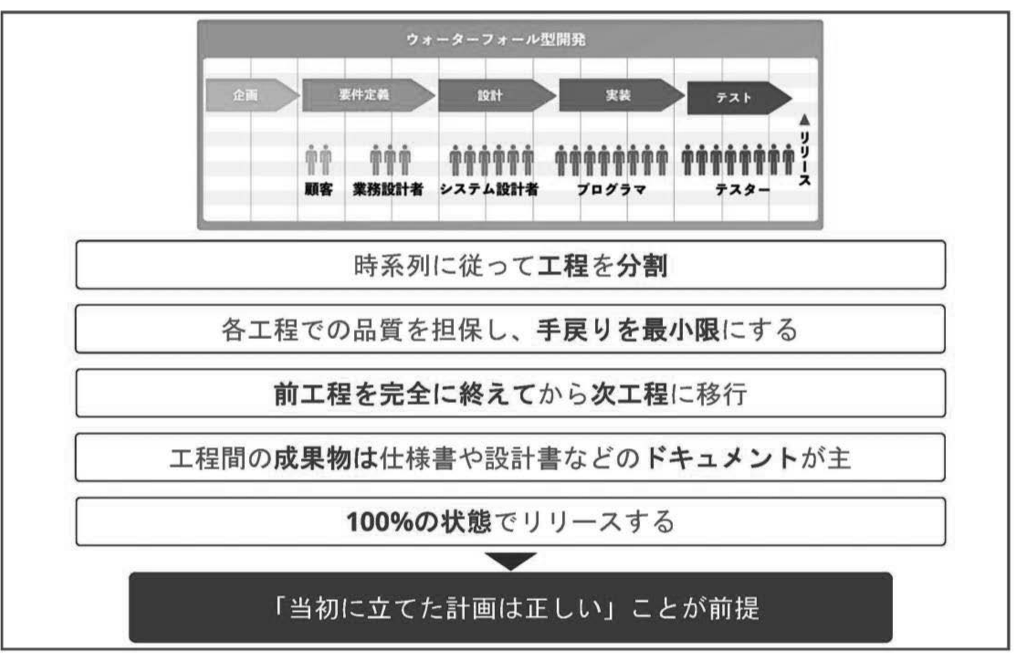
- ソフトウェア開発を、それまでの職人芸的な作成法から
  近代的な工業製品としての作成方法に変える方法として、
  ウォーターフォール・モデルの原形が提唱された　1968

- Royceの論文
  『Managing the Development of Large Software Systems』　1970
  ※本論文では下流工程を予備的に行い、上流工程に戻る「フィードバックループ」が提唱されていた

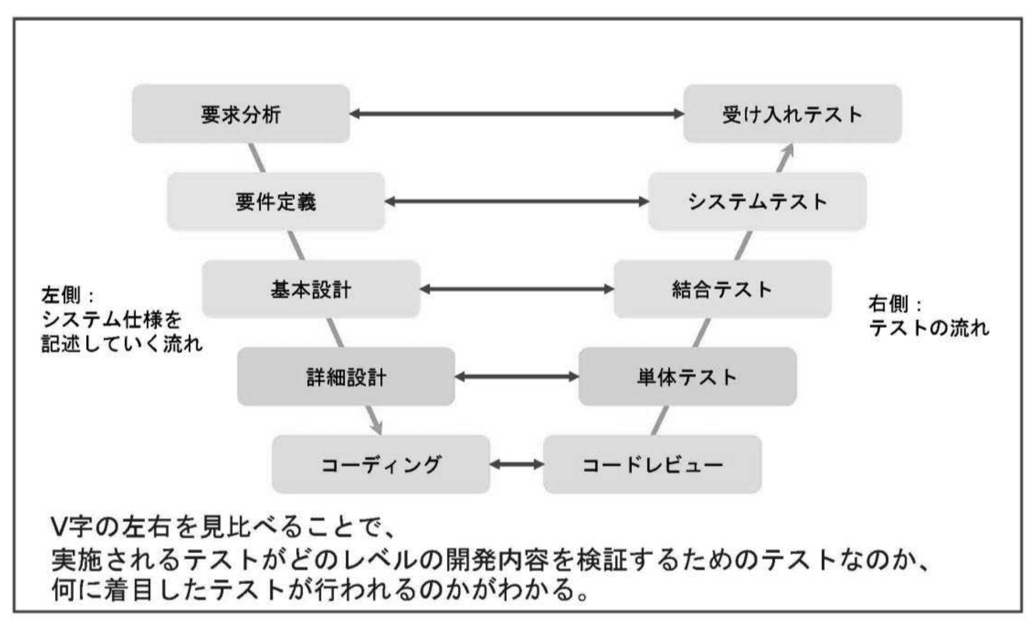
- BellとThayerの論文
  『Software Requirement』　1976

- 防衛用ソフトウェア開発に関する
  米国防総省の規格書「DOD-STD-2167 」　1985
  ※フィードバックの欠落したウォーターフォール・モデルを広めるきっかけになった

ウォーターフォール・モデル



V字モデル



ウォーターフォール・モデルのメリット

- つくるもの（要件）が明確になる
- プロジェクトの全体を把握できる
- 各工程で成果物ができるため、進捗管理しやすい
- 計画的に開発するため、早い段階で品質を作りこめる
- 工程が並行して実行されることがないため、
  無駄な時間を使ってしまうことを防げる

ソフトウェア開発における古典的見積り方法

ファンクションポイント法

ソフトウェアが持つ"機能"に着目し、その数や複雑さによって重み付けした点数を付け、合計点数から開発工数を見積る。

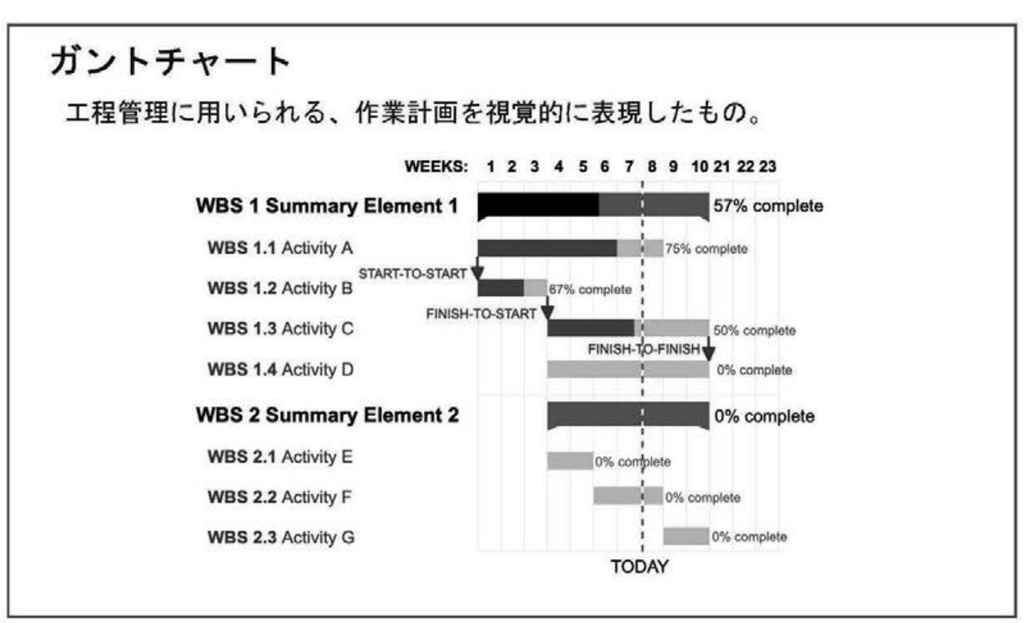
COCOMO

予想されるソースコードの行数にソフトウェアの規模を表す係数をかけて平均的な開発工数を求める。その値にソフトウェア属性やプログラマの能力などからなる15個の特性の係数をかけ合わせることにより工数を算出する。

KKD法

勘(K)と経験(K)と度胸(D)で見積る。過去の事例を元に判断するやり方。

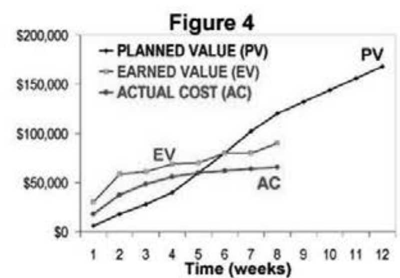
プロジェクト管理に利用されるツール



プロジェクト管理に利用されるツール

EVM（Earned Value Management）

作業の到達度を金銭などの価値に換算したEV（Earned Value : 出来高）という概念で把握する。



EVMで用いられる指標

EV : 出来高
PV : 計画価値
AC : 実コスト
SV : スケジュール差異。EVとPVの差で表される。
CV : コスト差異。EVとACの差で表される。
SPI : スケジュール効率指数。PVに対するEVの比率で表される。
CPI : コスト効率指数。ACに対するEVの比率で表される。

1.2 ウォーターフォール・モデルワークショップ

ウォーターフォール・モデル
ワークショップ
～ 自己紹介してみましょう ～

概要設計

自己紹介のテーマを

２つ以上考えてください

成果物

自己紹介のテーマ２つ

※時間が来たら左隣の人に成果物を渡してください

詳細設計

２つのテーマを詳細化してください

成果物

詳細化されたテーマ

※時間が来たら左隣の人に成果物を渡してください

開発

設定されたテーマに合わせて

自己紹介の内容を考えてください

成果物

・自己紹介文

テスト

グループ内で自己紹介してください

運用

- 自己紹介はうまくできましたか？
- 要求は満たせましたか？

1.3 グループディスカッション

グループディスカッション

グループごとに、ウォーターフォール・モデルの
メリット・デメリットについて
まとめてみましょう

1.4 アジャイルソフトウェア開発基礎

アジャイルソフトウェア
開発基礎

ウォーターフォール・モデルの問題点

よく言われる問題点

- **完成品**を見られるまで**時間がかかる**
- 期待と**実物の差異**
- ビジネス環境の**変化への対応が困難**

実際のプロジェクトでは...

- **計画は遅れがち**で、後工程に**しわ寄せ**
- 計画齟齬を起こさないように次のプロジェクトでは**さらに大きなバッファ**をとるという**悪循環に陥る**ことも...
- **計画通りに推進**することが重要視されるため、チャレンジ精神が失われ、「言われたことしかやらない」**指示待ち状態になる**
- **問題やミスを隠蔽**したり、気づかないフリをする
（ピラミッド型組織にありがちな組織間の壁が**コミュニケーションを阻害**する）

なぜうまくいかないのか？

- 仕様がなかなか決まらない
- 開発の途中で仕様が変更される
- ドキュメントによる仕様伝達で、誤解が発生し、意図されていないモノが作られる
- チャレンジよりも計画に従うことが優先
  - 設計の終わっている製造フェーズでは、より良いアイデアが見つかってもそれを反映できない
  - テストフェーズで見つかった不具合は、その場しのぎのパッチワーク的な修正

もっとうまくやるためにはどうすればよいのか？

- **肝心なところは早期に**作って確認する
- (ニーズを理解している) **ユーザーからのフィードバックを得る**
- 遠い未来のことまで**計画を立てすぎない**
- 変える**必要があればどんどん変える**

アジャイルソフトウェア開発とは

アジャイルソフトウェア開発 (アジャイルソフトウェアかいはつ、英: agile software development) は、ソフトウェア工学において迅速かつ適応的にソフトウェア開発を行う軽量な開発手法群の総称である。

近年、アジャイルソフトウェア開発手法が数多く考案されている。ソフトウェア開発で実際に採用される事例も少しずつではあるが増えつつある。
アジャイルソフトウェア開発手法の例としては、エクストリーム・プログラミング (XP) などがある。

非営利組織 Agile Alliance がアジャイルソフトウェア開発手法を推進している。

Wikipedia contributors. "アジャイルソフトウェア開発." Wikipedia. Wikipedia, 12 Feb. 2017. Web. 12 Feb. 2017.
～ https://ja.wikipedia.org/wiki/アジャイルソフトウェア開発 より ～

アジャイルの歴史

- トヨタ生産方式　1950
- 建築のデザインパターン　197?
- 野中,竹中の論文（The new new development game）　1986
- スクラム　1993
- 組織・プロセスパターン　1994
- GoF デザインパターン　1995
- XP（Extreme Programming）　1999
- アジャイルアライアンス設立　2001

アジャイルソフトウェア開発宣言

アジャイルソフトウェア開発宣言

私たちは、ソフトウェア開発の実践
あるいは実践を手助けをする活動を通じて、
よりよい開発方法を見つけだそうとしている。
この活動を通して、私たちは以下の価値に至った。

プロセスやツールよりも個人と対話を、
包括的なドキュメントよりも動くソフトウェアを、
契約交渉よりも顧客との協調を、
計画に従うことよりも変化への対応を、

価値とする。すなわち、左記のことがらに価値があることを
認めながらも、私たちは右記のことがらにより価値をおく。

Kent Beck　Mike Beedle　Arie van Bennekum　Alistair Cockburn
Ward Cunningham　Martin Fowler　James Grenning　Jim Highsmith
Andrew Hunt　Ron Jeffries　Jon Kern　Brian Marick　Robert C. Martin
Steve Mellor　Ken Schwaber　Jeff Sutherland　Dave Thomas

© 2001, 上記の著者たち
この宣言は、この注意書きも含めた形で全文を含めることを条件に自由にコピーしてよい。

～ http://agilemanifesto.org/iso/ja/manifesto.html より ～

アジャイル宣言の背後にある原則

私たちは以下の原則に従う:

顧客満足を最優先し、
価値のあるソフトウェアを**早く継続的に**提供します。

要求の変更はたとえ開発の後期であっても**歓迎**します。
変化を味方につけることによって、お客様の競争力を引き上げます。

動くソフトウェアを、2-3週間から2-3ヶ月という
できるだけ**短い時間間隔でリリース**します。

ビジネス側の人と開発者は、プロジェクトを通して
日々一緒に働かなければなりません。

意欲に満ちた人々を集めてプロジェクトを構成します。
環境と支援を与え仕事が無事終わるまで彼らを**信頼**します。

情報を伝えるもっとも効率的で効果的な方法は
フェイス・トゥ・フェイスで話をすることです。

動くソフトウェアこそが進捗の最も重要な尺度です。

アジャイル･プロセスは**持続可能**な開発を促進します。
一定のペースを継続的に維持できるようにしなければなりません。

技術的卓越性と優れた設計に対する
不断の注意が機敏さを高めます。

シンプルさ（ムダなく作れる量を最大限にすること）が本質です。

最良のアーキテクチャ・要求・設計は、
自己組織的なチームから生み出されます。

チームがもっと効率を高めることができるかを定期的に振り返り、
それに基づいて自分たちのやり方を最適に調整します。

～ http://agilemanifesto.org/iso/ja/principles.html より ～

アジャイル開発のポイント

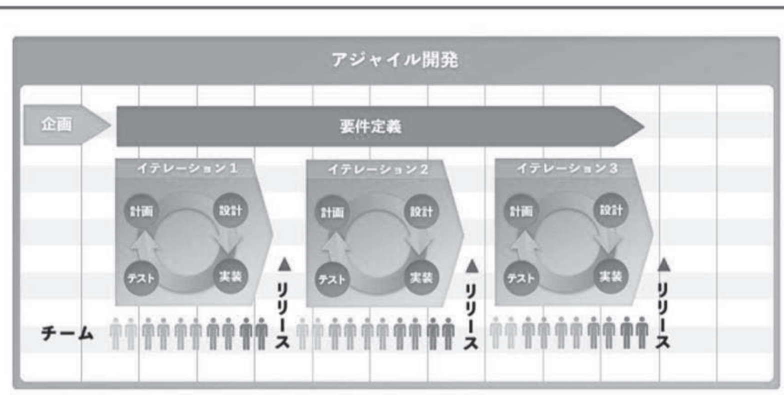
素早く柔軟に（変化を受け入れる）

- 顧客は、早くモノを見たい
- 主要機能だけでも、先に見たい
- 顧客のニーズは変化する
- 必要なのは『課題を解決すること』であり、『計画通りに進めること』ではない

優先順位の高いものから着手

- 顧客が本当に求めていることを認識する
- タスクに優先順位をつける
- 完成したものから随時提供していく

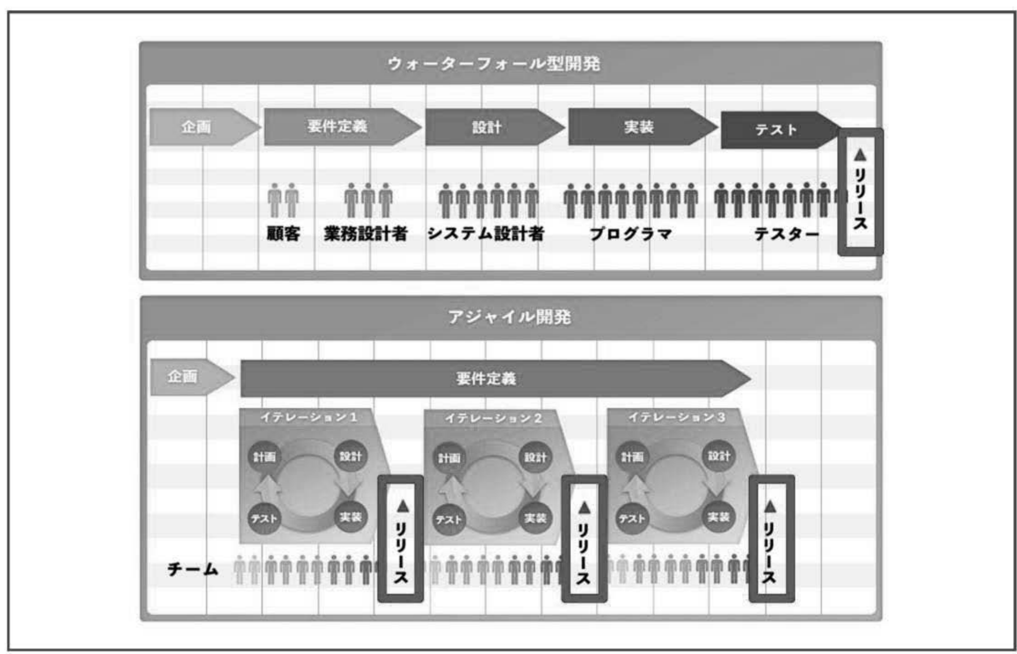
短いスパンでイテレーティブに開発



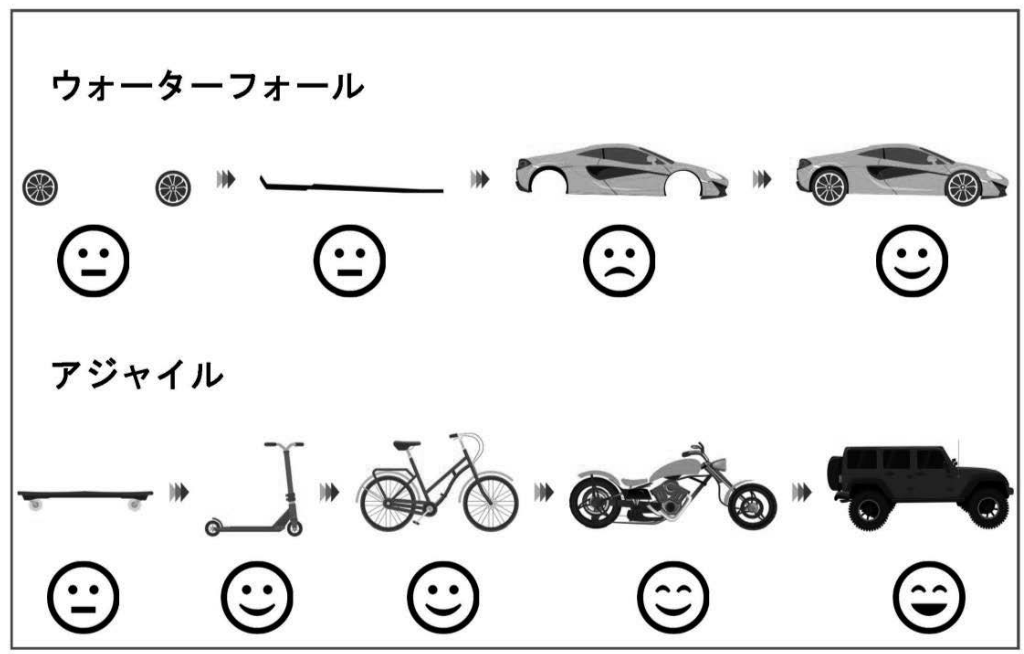
- 要件変更に**柔軟に対応**する
- **リスクを最小限**にできる
- 迅速かつ**継続的なリリース**
- **高い顧客満足**

市況の変化に素早く適応することで、
顧客の競争力を高めビジネスの成功を支援する

開発手法の比較 - 素早く柔軟に -



開発手法の比較 - 優先順位の高いものから着手 -



開発手法の比較 - フィードバックに対する対応 -

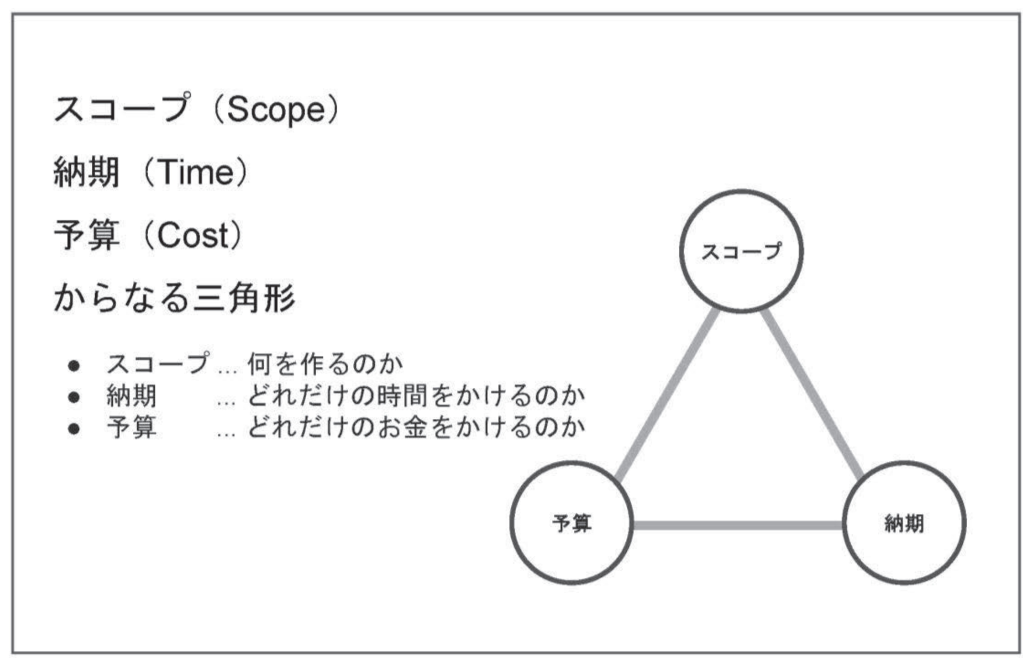
ウォーターフォール型開発

フィードバックが得られるまでに時間がかかる

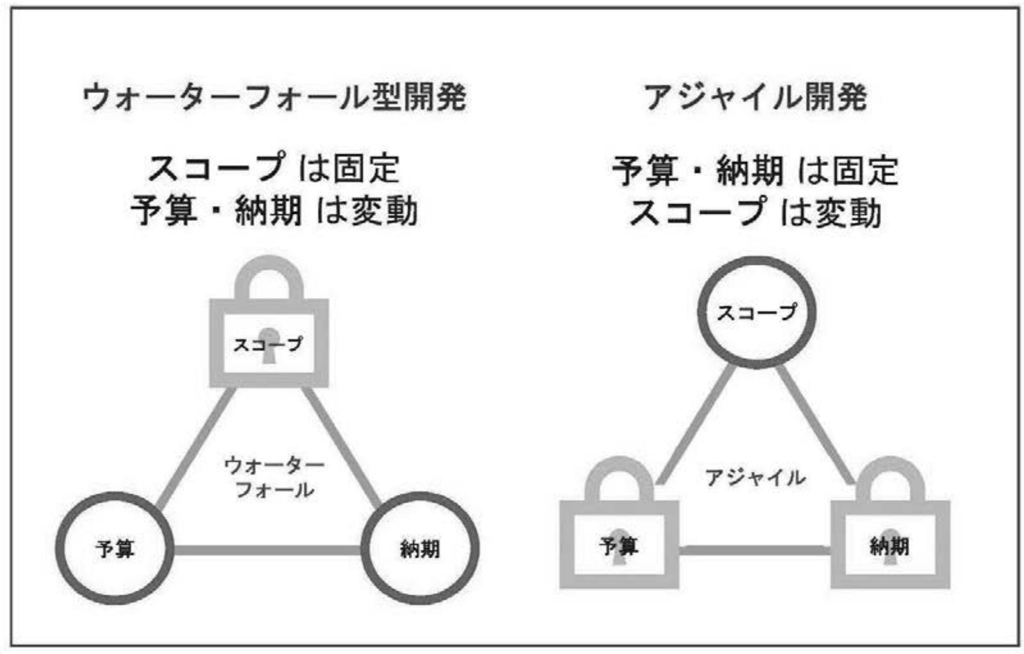
アジャイル開発

要求の変化を受け入れ、素早いフィードバックループで変化に対応

プロジェクト管理における鉄の三角形



開発手法の比較 - プロジェクト管理 -



開発手法の比較 - 成果物と時間の使い方 -

ウォーターフォール型開発

成果物を一つの塊とみなし、それが完成するまでに必要な時間をかける。

アジャイル開発

１～４週間の短い時間のタイムボックスを設け、その中で作れるサイズに成果物を分割する。できた成果物を見て、次のタイムボックスで開発するものを変更できる。

開発手法の比較 - コストと時間の使い方 -

ウォーターフォール型開発

各工程ごとに必要な人員やリソースを割り当てる。工程ごとに別の人が対応することが多い。

アジャイル開発

チームを固定し、同じメンバーで要件定義から設計、開発、テストに至るまでのすべての工程を行う。

開発手法の比較 - ドキュメントとスキル -

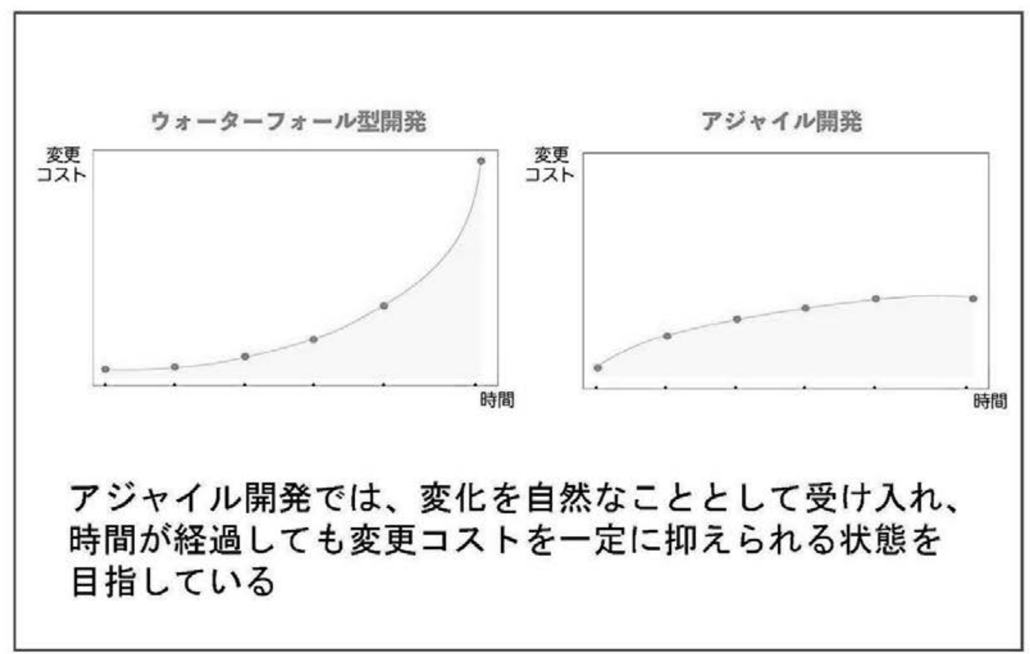
ウォーターフォール型開発

各工程で人が変わることが多く、工程間の情報の引き継ぎのために仕様書や設計書などのドキュメントが重要になる。

アジャイル開発

同じメンバーですべての工程を行うため、前工程で学んだことはチームのスキルやノウハウとして蓄積される。ドキュメントは最小限。

開発手法の比較 - 変更コスト -



代表的なアジャイル開発手法

手法	説明	視点
リーン Lean Software Development	Mary Poppendiekらが提唱している手法（考え方）。 トヨタ生産方式をお手本として、ソフトウェア開発を成功させるための原則集。この原則をもとに、具体的なプラクティスを生み出す。第一原則は「ムダの排除」。	**経営者視点** ・原理/原則
スクラム Scrum	Ken Schwaber, Jeff Sutherlandらが提唱している手法。 ソフトウェア開発のマネジメント面にフォーカスをあて、チームを自律的に動かすための場作りの仕掛け（フレームワーク）を提供している。	**管理者視点** ・原理/原則 ・管理プラクティス
XP Extreme Programming	Kent Beckらが提唱している手法。 「変化ヲ抱擁セヨ」をスローガンとして、ソフトウェア開発技術の複数のベストプラクティスを極端に実施することで、開発サイクルを素早く回す。	**開発者視点** ・原理/原則 ・開発プラクティス ・管理プラクティス

XP(Extreme Programming)

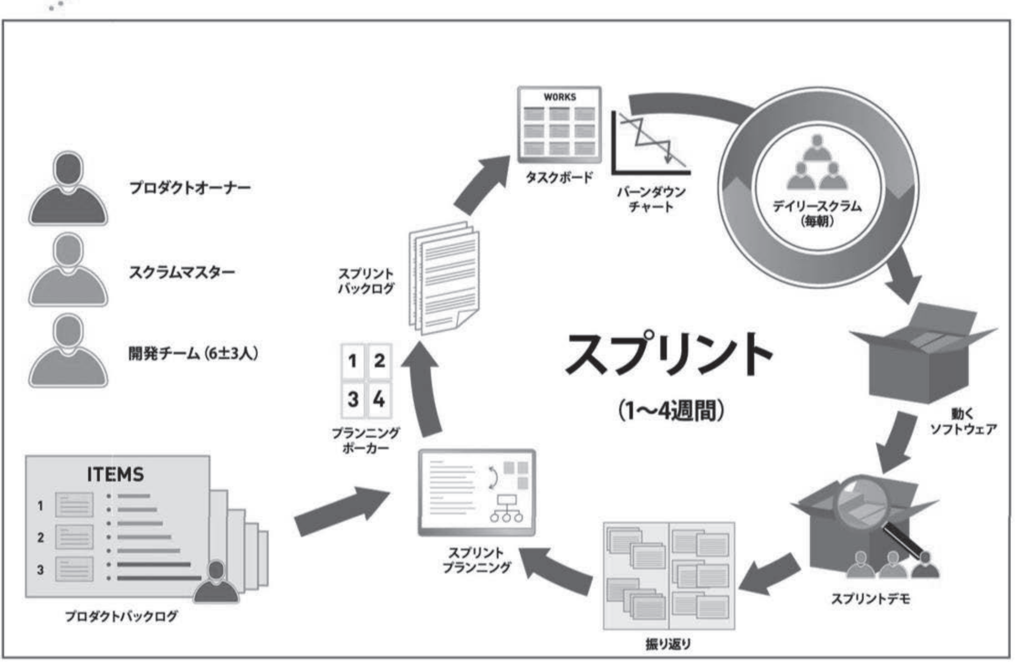
◈ 価値

・コミュニケーション　・シンプル　・フィードバック
・勇気　・尊重

◈ ルール

計画	・ユーザーストーリー　・リリース計画　・小さなリリース ・反復　・イテレーション計画
管理	・開けた作業空間　・継続可能なペース　・朝会　・ベロシティ ・マルチスキル　・ふりかえり
設計	・シンプルさ　・メタファー　・CRC　・スパイク　・YAGNI ・リファクタリング
実装	・顧客同席　・コーディング規約　・TDD　・ペアプログラミング ・順次統合　・頻繁な統合　・継続的統合　・コードの共同所有
テスト	・ユニットテスト　・回帰テスト　・受け入れテスト

Scrum



Scrumの理論/価値基準/ルール

◈ **経験的プロセス制御の実現を支える３つの柱**

・透明性　・検査　・適応

◈ **価値基準**

・確約（commitment）・勇気（courage）
・集中（focus）・公開（openness）・尊敬（respect）

◈ **ルール**

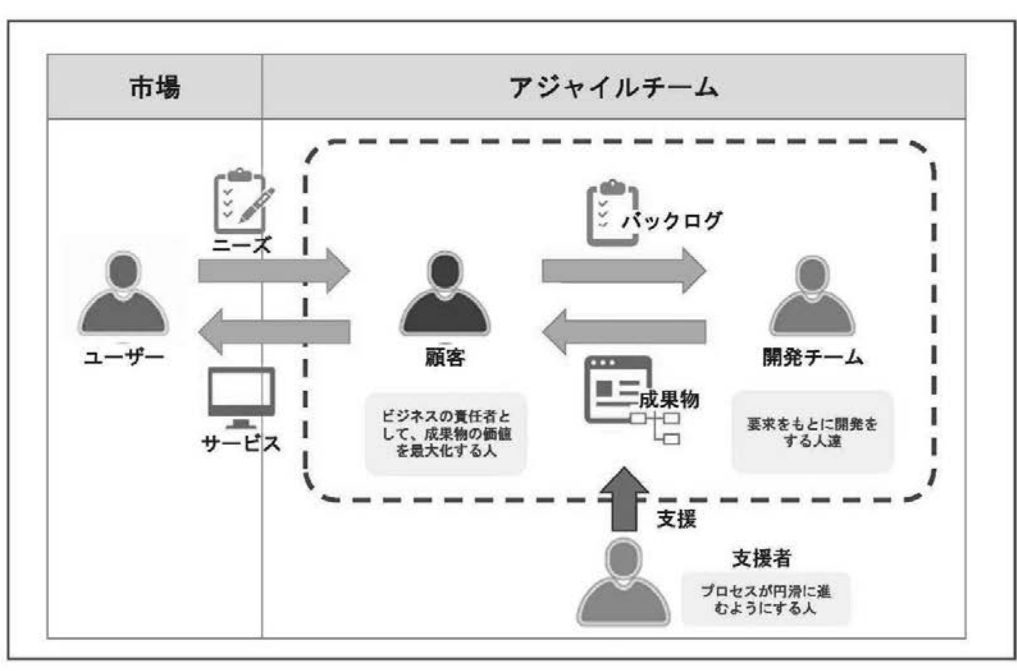
３つのロール

・プロダクトオーナー　・開発チーム　・スクラムマスター

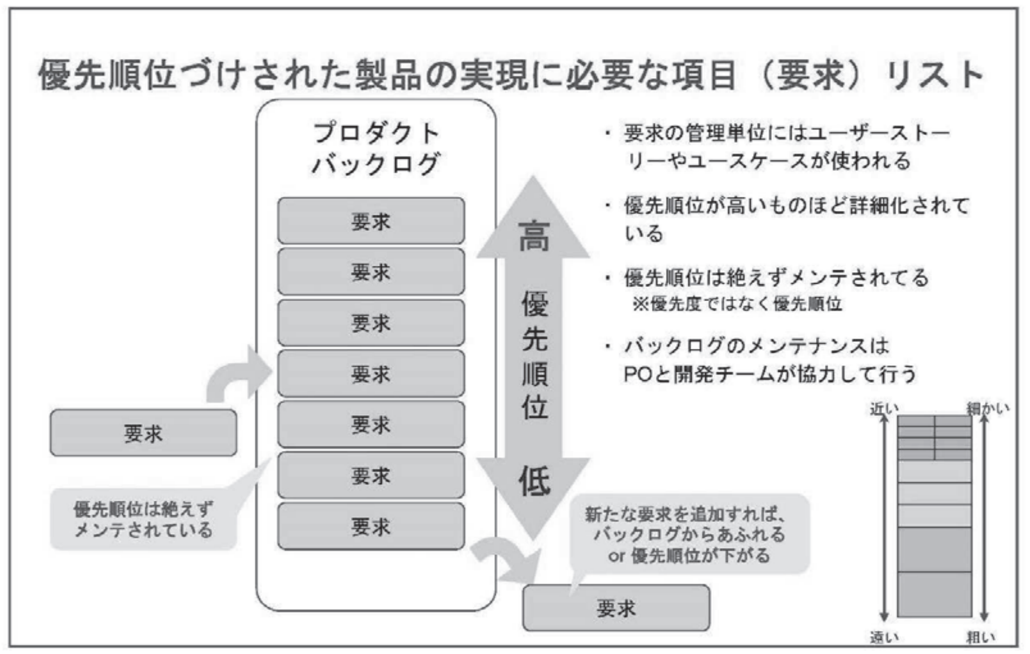
４つのイベント

・スプリントプランニング　・デイリースクラム
・スプリントレビュー　・スプリントレトロスペクティブ

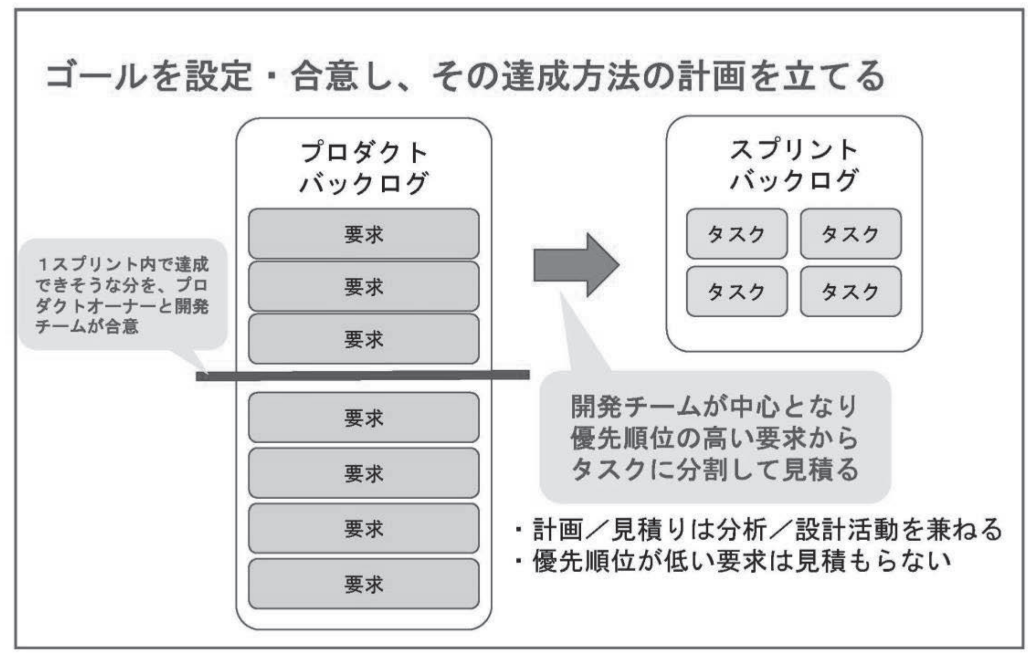
アジャイル開発の主な登場人物



プロダクトバックログ（PBL）



スプリントプランニング・イテレーション計画



デイリースクラム・朝会

スプリントゴールが達成可能か、障害があるか確認する

・質問の例「昨日やったこと、今日やること、
　スプリントゴールを達成するために
　障害となっていること」

・情報共有が目的

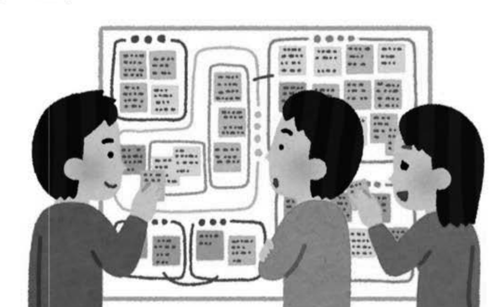
・タスクボードなどの前で行う

・開発チームは全員参加

・スタンドアップで15分以内で実施

・2次会で詳細検討や再計画を実施

・スプリントゴールが達成できるように全員で協力



スプリントの終了

スプリントプランニング

開発

- スプリント中は、チーム外部からのゴールが変わるような変更は受け付けない
- 日々、デイリースクラムを実施する
- 計画したタスクが残っていても時間になったら終了する
- 時間前にスプリントが中止されることもある

スプリントレビュー（デモンストレーション）

- 開発チームはプロダクトオーナーにレビューを行いフィードバックをもらう
- 他の利害関係者にも参加してもらうのが望ましい

スプリントレトロスペクティブ（ふりかえり）

- イテレーションで学んだことを確認する
- 次のスプリントでよりうまくできるようになる為の改善のアイデアを創出する

1.5 アジャイルの各種プラクティス

アジャイルの各種プラクティス

「計画」に関するプラクティス

リリース計画
どのストーリーをどのイテレーションで実現するか、
顧客が主体となって計画を立てる。

小さなリリース
リリースは可能な限り小規模なものにし、
妥当だと判断されれば速やかにリリースする。

イテレーション計画・スプリントプランニング
イテレーション（スプリント）で開発するものについて
開発チームが主体になって見積り、計画を立てる。

ユーザーストーリー
要求仕様を自然言語で簡潔に記述したもの。
『＜役割＞として、＜機能＞ができる。なぜなら＜価値＞だからだ。』
というテンプレートで記述される。

「管理」に関するプラクティス

開けた作業空間
チームに専用の作業場所を与える。

持続可能なペース
知的作業には、週40時間の労働時間が最適である。
心身ともに健全な状態を保つ必要があり、それ以上の労働は適切ではない。

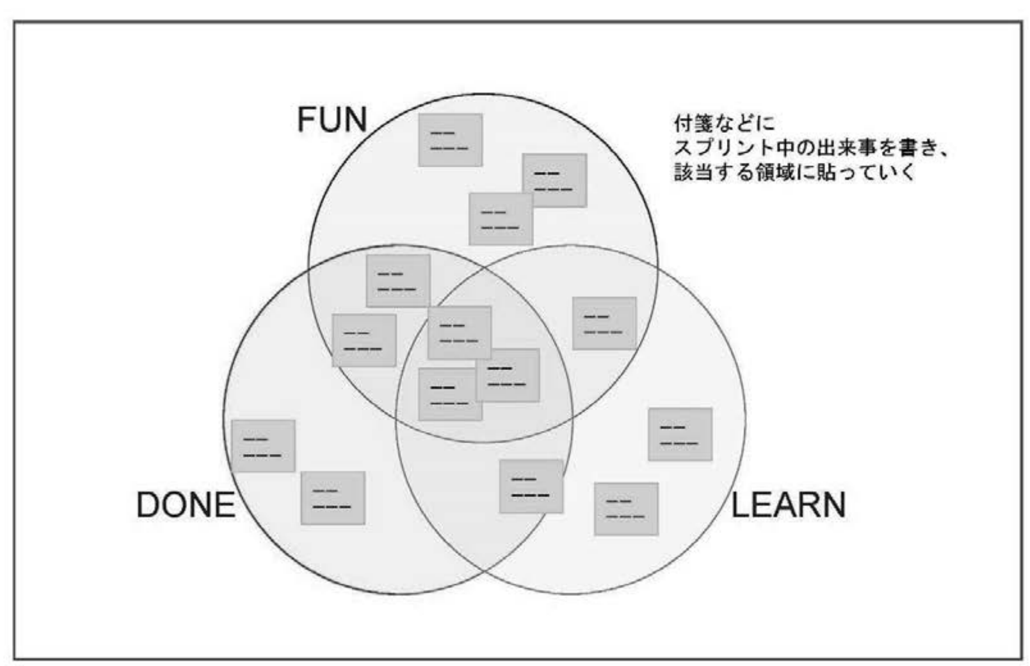
朝会・デイリースクラム
開発チームは日々、情報共有しながら開発をすすめる。
必要があれば再計画を行う。

ベロシティ
決められた期間の中で開発チームが生み出せる成果物の量のこと。

マルチスキル
深刻な知識の損失とボトルネックを解消するために、多能工を目指す。

ふりかえり
次により良いものを作るために定期的にふりかえり、継続的に改善する。

Fun! Done! Learn!



ふりかえり

失敗から学び成長するために、**定期的にふりかえる**

◇ ふりかえりの代表的な手法
- KPT（Keep, Problem, Try）
- YWT（やったこと, わかったこと, 次にやること）
- Fun/Done/Learn

など

Keep 「今やっていて、これからも続けたいこと」	Try 「試したいこと」
Problem 「問題だと認識していること」	

怒、哀、喜(Mad Sad Glad)

Mad Sad Gladは、前のスプリントでのチームの感情に関する議論を構成し、チームの士気と仕事の満足度を改善する機会を特定する手法です。
フラストレーションやイライラ（怒り）、失望（悲しみ）、そして幸せや誇り（うれしさ）を参加者全員と共有します。

Mad … あなたを夢中にさせているものをリストアップしてください。
あなたの最高のパフォーマンスを妨げるものは何ですか？
Sad … あなたを失望させた、または改善したいと思ったものは何ですか？
Glad… このプロジェクトについて考えるとき、何があなたを幸せにしますか？
一番楽しんでいる要素は何ですか？

その他のふりかえり手法

ブレインストーミング

タイムライン

フィッシュボーン

Five Whys

満足ヒストグラム

555(Triple Nickels)

など、多くの手法があります。
※上記はほんの一例です。

許可を求めるな、謝罪せよ

『許可を求める＝許可されていないことはやらない』
→受動的。チャレンジしない／できない。

うまくいくこともあれば、うまくいかないこともある。
試行錯誤によってのみ学びが得られる。
とにかくやってみて、もしうまくいかなければ
別の手段を考案する。

目的を解決する手段は無数にある。
「なぜそれが必要なのか？」を考え、
本当に必要なことだけをやる

「設計」に関するプラクティス

シンプルさ
設計を単純に保つ。テスト可能、理解可能、閲覧可能、説明可能、という４つの主観的な品質で評価するとよい。

メタファー
システムがどのように組み合わさってできているかの理解を促すもの。アーキテクチャーと呼ばれているものに類似している。

リファクタリング
完成済みのコードでも、随時、改善処置を行う。この際、外部から見た動作は変えずに、内部構造をより見通し良く優れたものになるようにする。

YAGNI
今必要とされている機能だけのシンプルな設計・実装を行う。将来的に使うかもしれない、という機能の設計は行わない。

ふりかえり
次により良いものを作るために定期的にふりかえり、継続的に改善する。

「実装」に関するプラクティス

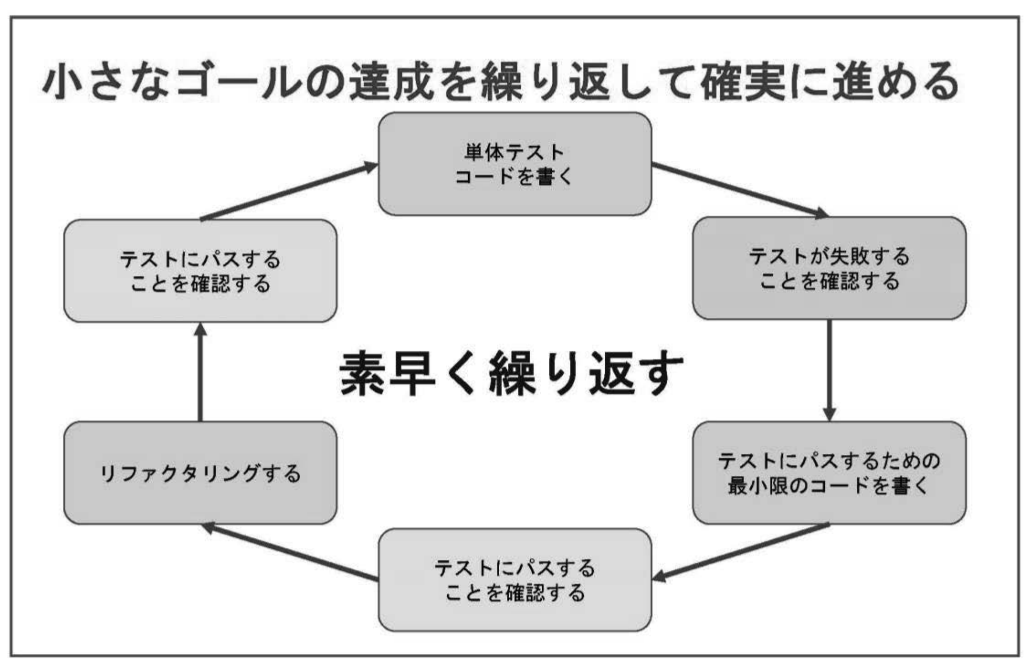
顧客同席
常に顧客とコミュニケーションをとれるようにしておく。あらゆる規模のプロジェクトで、顧客はフルタイムのコミットが必要。

コーディング規約
コーディング標準をチームで定義し、一貫性のあるコードにする。

テスト駆動開発（TDD）
実装よりも先にテストを作成する。実装は、そのテストをパスするようにシンプルに作成する。テストはテストコードとしてプログラミングし自動化することで、変更コストを抑制することができる。

ペアプログラミング
プログラミングを、二人一組で行う。一人がコードを書き、もう一人はそれをチェックしながら、仕様を確認したり全体構造を考えたりするなど、ナビゲートを行う。二人は、この役割を定期的に交代しながら仕事を進める。

テスト駆動開発



「実装」に関するプラクティス

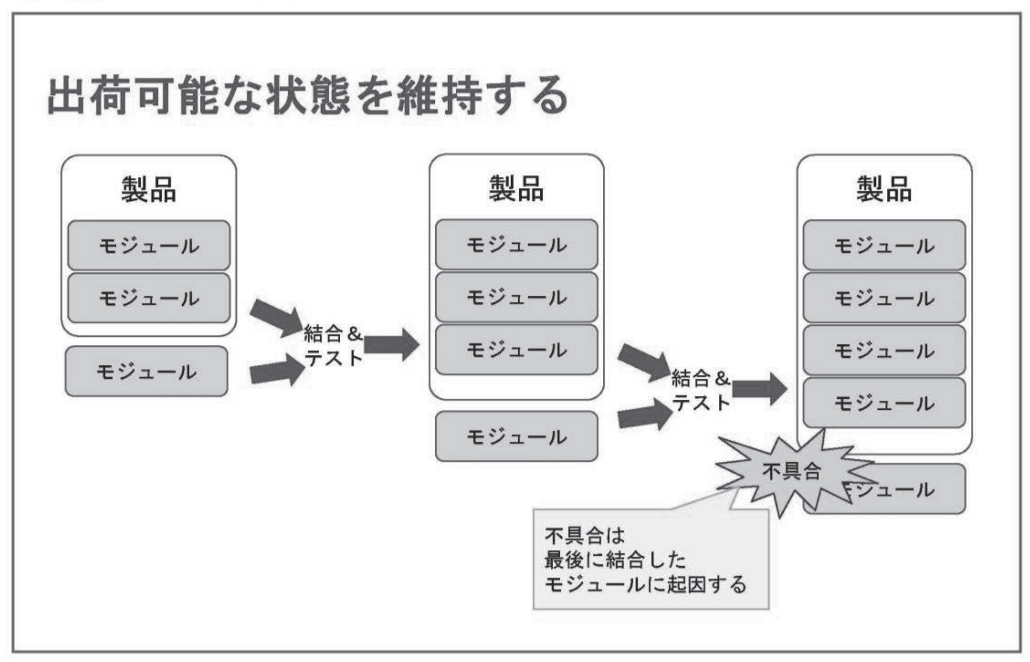
順次統合
一度に複数の機能を統合しない。

頻繁な統合
可能な限り数時間ごとにコードをリポジトリに統合する。

継続的インテグレーション
単体テストをパスするコードが追加されるたび、全てのコードがテストされ、全体として正常な状態を維持していく。

コードの共同所有
チームのすべてのメンバは、いつでもどのコードでも変更できる。

継続的インテグレーション



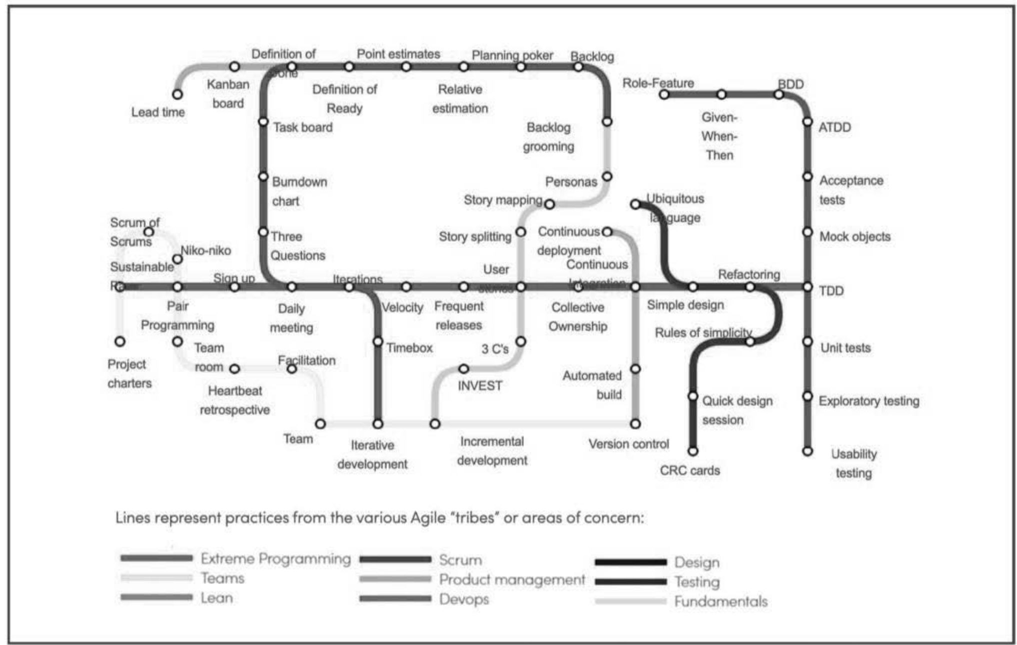
「テスト」に関するプラクティス

ユニットテスト
すべてのコードにはユニットテストが必要。リリース前に、すべての
コードはすべてのユニットテストにパスしなければならない。

回帰テスト
バグが発見された場合、バグの再発を防ぐためにテストが作成される。

受け入れテスト
イテレーションごとに顧客の立場からテストを行い、
ストーリーが実現できているか、望むシステムになっているか確認する。
受け入れテストは頻繁に実行され、テストのスコアは公開される。

アジャイルプラクティスマップ



https://www.agilealliance.org/agile101/subway-map-to-agile-practices/ より引用

1.6 アジャイルのワークショップ　～飛べ！紙飛行機！～

アジャイル ワークショップ
～ 飛べ！紙飛行機！ ～

目的

他のチームより
できるだけ多くの
飛行機を飛ばす

ルール

- 同じ人が2回以上連続で紙を折ることはできません。
  1回紙を折ったら次の人に渡してください。
- 全長15cm、全幅8cm以内のサイズにしてください。
- 機体の先端は尖っててはいけません。
  （2cm程度の幅を持つようにしてください）
- 各機体は1回だけテスト飛行場で飛行試験を受けられます。
- 3m以上飛行した機体のみ納品できます。
- 制限時間は5分間です。
- 「見積り→実施→ふりかえり」を4サイクル繰り返します。

作りかけの飛行機はサイクル終了時に都度破棄します。

見積り

チームで制限時間内に何機納品
できるか相談してください。

（相談時間１分）

実施

できるだけ多くの飛行機を
作ってください。

（制限時間５分）

成果発表

完成品は何機できましたか？
チームごとに発表しましょう。

ふりかえり

次のサイクルに向けて
チームで改善点を
話し合ってください。

（制限時間２分）

条件変更（３サイクル目）

一つだけルールを変更できます。
どのルールを変更するか
1分で決めてください。

決まったら変更内容と
その理由も合わせて
発表しましょう。

見積り

チームで制限時間内に何機納品
できるか相談してください。

（相談時間１分）

実施

できるだけ多くの飛行機を
作ってください。

（制限時間５分）

成果発表

完成品は何機できましたか？
チームごとに発表しましょう。

条件変更（４サイクル目）

ルールは**最初のとおり**に戻ります。

ふりかえり

次のサイクルに向けて
チームで改善点を
話し合ってください。

（制限時間２分）

見積り

チームで制限時間内に何機納品
できるか相談してください。

（相談時間１分）

実施

できるだけ多くの飛行機を
作ってください。

（制限時間５分）

成果発表

完成品は何機できましたか？
チームごとに発表しましょう。

総合結果発表とふりかえり

- 4サイクル実施した結果はどうなりましたか？
- プロセスは改善できましたか？
- より大きな成果を出すためには
  どうすればいいと思いますか？
- 見積り/実施/ふりかえり、それぞれのフェーズ
  をアジャイル開発に置き換えると？

1.7 グループディスカッション

グループディスカッション

1.8 アジャイルでのプロジェクト管理

アジャイルでのプロジェクト管理

計画づくりの難しさ

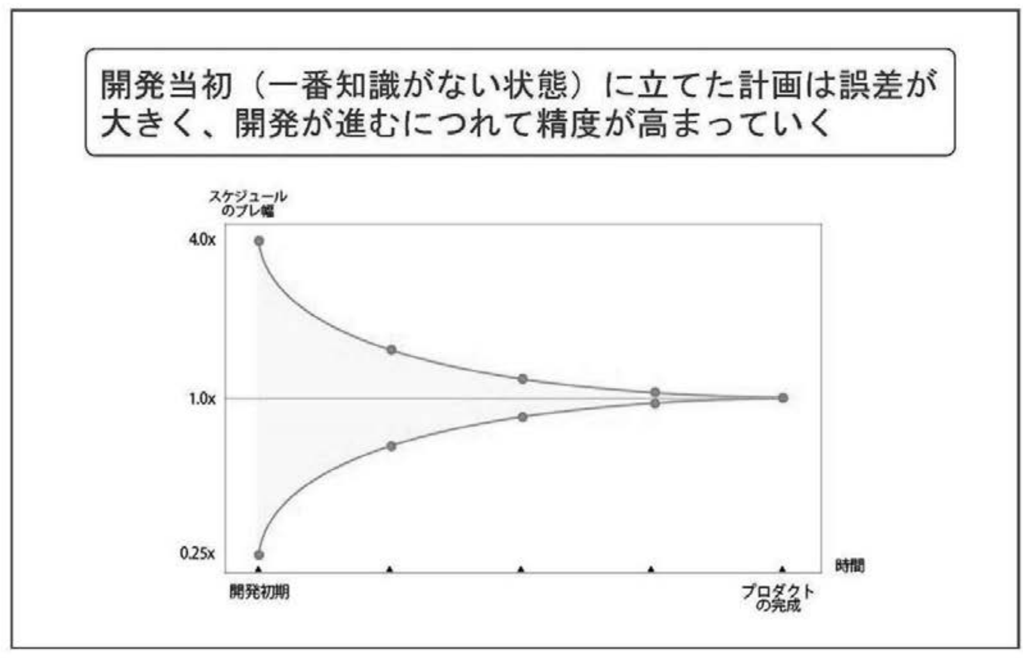
プロダクトを取り巻く状況によって
プロダクトへの要件は常に変化しており、
開発当初に全ての事象を予測することは非常に困難

計画を狂わせる要因

- 途中で要件や仕様が変わる
- 開発開始後に新たな課題が見つかる

- より良い実装手段が見つかった
- メンバーのスキルアップ
- などなど

マコネルの不確実性コーン



アジャイルな計画づくり

状況の変化に対応するためのプラクティス

- インセプションデッキ
- プロダクトバックログ
- スプリントバックログ
- ストーリーポイントによる見積もり
- カンバン
- バーンダウンチャート
- ベロシティ計測

インセプションデッキ

インセプションデッキ

Whyを明らかにする

1. **われわれはなぜここにいるのか**
   何の為にチームを組み、顧客は誰で、プロジェクトの始まった理由等を再確認する
1. **エレベータピッチ**
   30秒以内に2センテンスでプロジェクトの本質を定義すると？
2. **パッケージデザイン**
   製品の広告をきめる。キャッチコピーと楽しい画像！
3. **やることやらないことリスト**
   やること、やらないこと、未定なことをはっきりさせる
4. **プロジェクトコミュニティ**
   このプロダクトに関わる人全てを洗い出す

グループごとに、ウォーターフォールモデルと
アジャイル開発の手法の違いや
それぞれのメリット・デメリットについて
まとめてみましょう

Howを明らかにする

6. 技術的な解決案

概要レベルでのアーキテクチャ設計図を書こう

7. 夜も眠れない問題

プロダクトを開発するにあたって、予想される世にも恐ろしい問題は？

8. 期間を見極める

どのくらいの期間が必要なプロジェクトだろうか？

9. トレードオフスライダー

機能、予算、時間、品質のどれを優先させるのか？
（その他　使い勝手、デザイン…）

10.なにがどれだけ必要か

最終的に必要なスタッフ、お金、時間をまとめる。

プロダクトバックログ

プロダクトに必要な機能・要件・要望・修正事項をリスト化したもの

- 顧客に提供できる価値の大きさやコストなどを勘案して**一列に順序づけ**
- ステークホルダーと調整して決定する
- 開発チームは**順位の高いものから着手**する
- **ユーザーストーリー**の形式で書かれることが多い

ユーザーストーリー

ユーザー視点に立って開発を進めるための手法

ユーザー（顧客）が抱える問題の解決についての議論であり、何を作るかについての共通理解を築くためのもの。

完璧な記述よりも、コミュニケーションによって認識を合わせる。

良いユーザーストーリーを書くために

システムの利用者に焦点をあてる

ストーリーの記述ではユーザーロールを意識する

ユーザーストーリーをもとに議論する

ユーザーストーリーはチームとステークホルダー間の
議論を活性化させるための道具

ユーザーストーリーを書くのはチーム全体の仕事

ユーザーストーリーを書くのに全員が協力する
ユーザーストーリーをより良くするために
定期的に**バックロググルーミングミーティング**を行う

ストーリーテンプレート

[ユーザの種類や役割]として
[達成したいゴール]をしたい
（なぜなら[理由]だからだ）

誰の　ためのストーリーで
何を　したいのか
（なぜ　そうしたいのか）

例：
[メールの受信者]として
[受信した電子メールをキーワード検索]したい。
なぜなら、[キーワードが含まれる電子メールを素早く見つけたい]からだ。

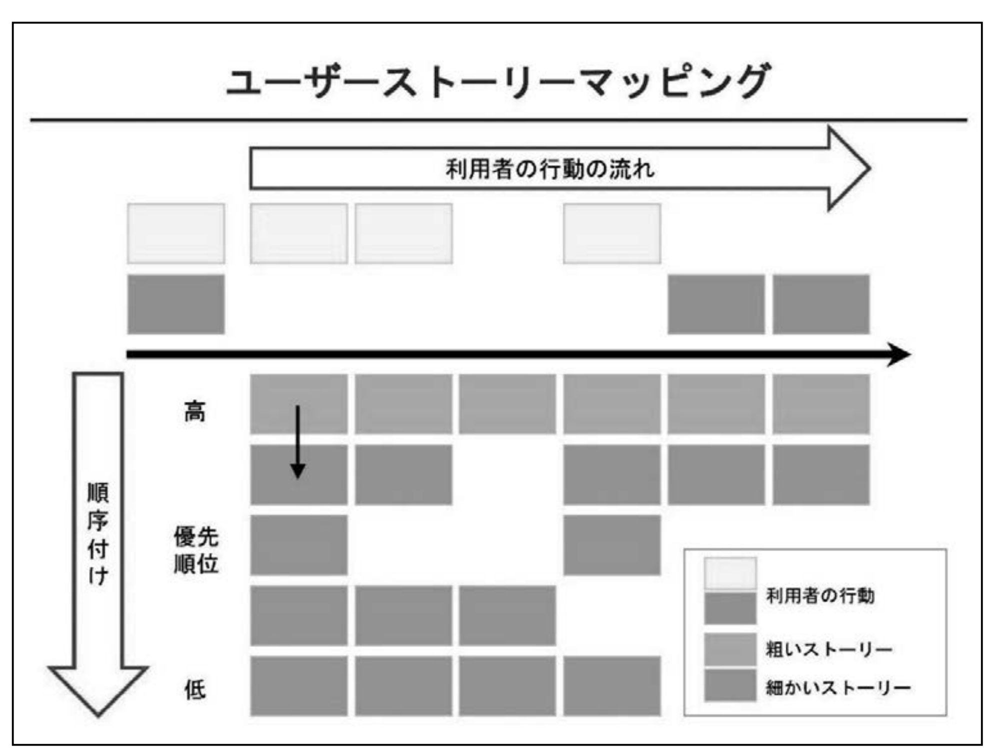
INVEST

良いユーザーストーリーに備わっている 6つの要素

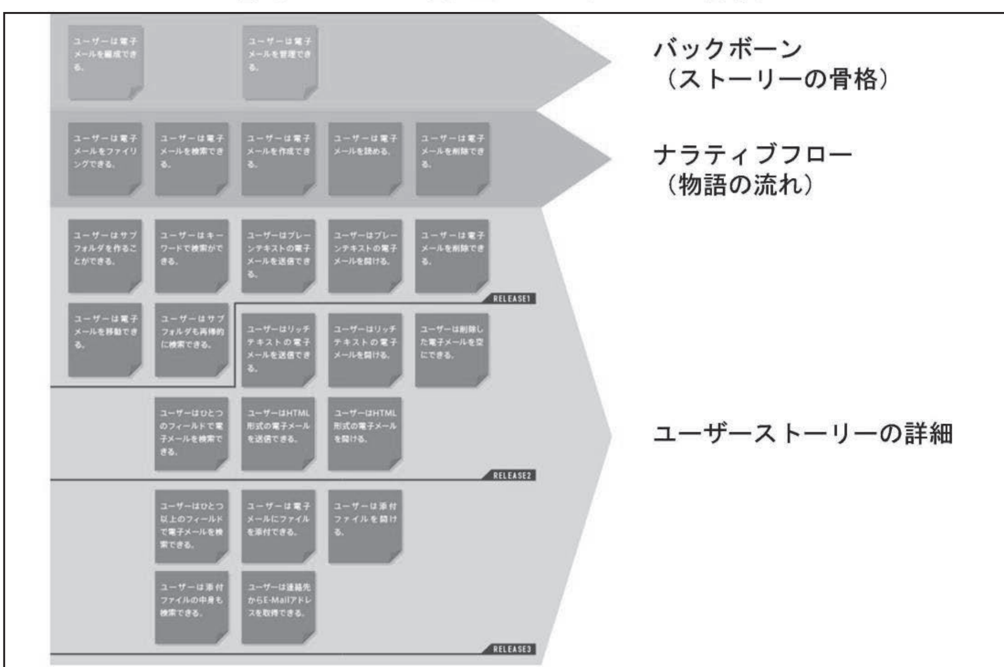
Independent	独立している	… ストーリーが独立していることで、必要に応じてスコープを柔軟に調整できる
Negotiable	交渉の余地がある	… 交渉の余地があれば、予算の範囲に収まるように、ある程度の融通が利くようにできる
Valuable	価値がある	… ユーザーが理解できるシンプルな言葉で表現する。対価を払ってもいいと思える物＝価値
Estimable	見積り可能である	… ストーリーを実現するにかかる時間を見積もれるだけの十分な情報がある。
Small	小さい	… ストーリーを実現するのに要する時間が長すぎない程度に、小さい（適切な）サイズに分割されている。
Testable	テストできる	… ユーザーストーリーのテストができれば、作業範囲と仕事の完了基準が明確になる。

ユーザーストーリーマッピング

書き出したユーザーストーリーを
ストーリーの流れや優先順位を軸に並べ替え、
ユーザーワークフローの全体像を可視化したもの



電子メール管理システムの場合



ストーリーポイントによる相対見積り

Q1. Aのビルは何メートルですか？

Q2. AのビルはBのビルの何倍ですか？

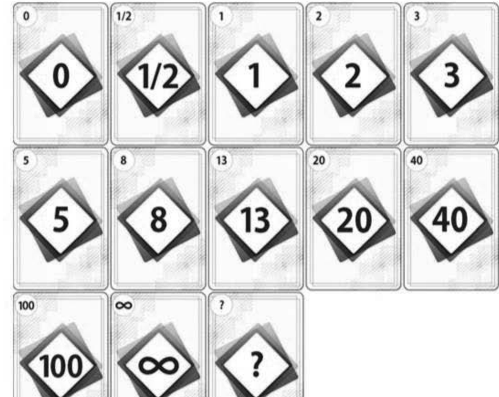
ストーリーポイントによる相対見積り

Q1. Aのビルは何メートルですか？
→絶対的な値による見積りは外れやすく
時間もかかる

Q2. AのビルはBのビルの何倍ですか？
→相対的な見積りは簡単で正確

プランニングポーカー

0, 1, 2, 3, 5, 8, 13, …の数字が書かれたカードを使用して素早く見積もる



- 時間をかけすぎない
- みんなで見積もる
- 認識を合わせる
- 大きな数字の小さな差は気にしない

カンバン（タスクボード）

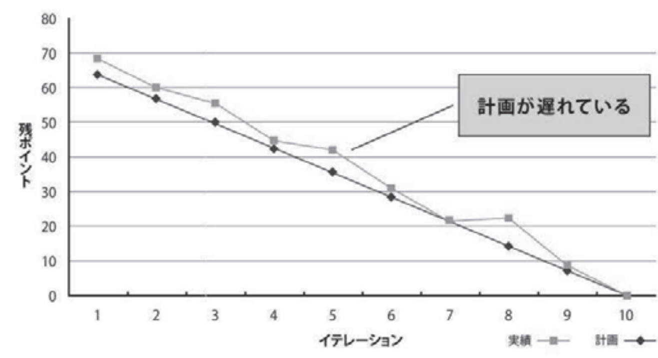
バックログを付箋などに書き出して作業を
見える化するためのツール

ToDo	Doing	Done

カードには、期日・残作業時間・作業責任者・進捗インジケータなどを補足情報として書く場合もある。

バーンダウンチャート

ポイントや残作業時間、残タスク数を使って
バーンダウンチャートを描くことでプロジェクト
の進捗を見える化する



計画と実績が合わない場合は、プロジェクトの期間かスコープ（実現するバックログ）を調整する

ベロシティ計測

ベロシティ＝スプリントで完成したバックログのポイントの合計

ベロシティを計測することで、
残りのバックログを消化するためにあと何スプリント必要かがわかります。

スプリントのベロシティにはバラつきがあるため
通常は数回分のスプリントの平均ベロシティを使います。

1.9 アジャイル開発を支援するツール

アジャイルソフトウェア開発
を支援するツール

アジャイルソフトウェア開発を支援するツール

情報共有

・Slack ・Wiki ・壁

カンバン・タスク管理

・Trello ・Redmine ・付箋

継続的インテグレーション

・Jenkins ・CircleCI

アナログツール vs デジタルツール

- 全員が同じ場所で作業しているなら
  自由度の高いアナログツールがオススメ
- 色々メトリクスを取りたくなったら
  デジタルに移行
- 開けないと中に何があるかわからない
  情報冷蔵庫にならないように注意が必要
- ツールに運用を合わせるのではなく、
  運用にマッチするツールを探す

情報共有 - Slack -

Slackとは

2013年8月のリリース後、IT業界に爆発的に
人気となっているチャットサービス。

slack の効果

- だれでもいつでも、平等に発言できる環境
- 好きなときに返事をする（自由）
- オープンなコミュニケーション（見える化）
- 楽しむ心

チーム力、生産性向上に繋がる

情報共有 - Wiki -

Wikiとは
独自の文法を使用し、Web上から簡単に
内容を置き換えることができるWebシステム。

フリー百科事典として有名なウィキペディアにも

Wikiシステムが使用されています。

（https://ja.wikipedia.org/）

～イメージ https://ja.wikipedia.org/より 抜粋～

Wikiの特徴・効果

- だれでもいつでも、平等に情報を書き込める環境
- ソースコードの共同所有ならぬ、ドキュメントの共同所有
- ドキュメントの代わり

効率的、効果的に情報共有

進捗管理の代表的な方法

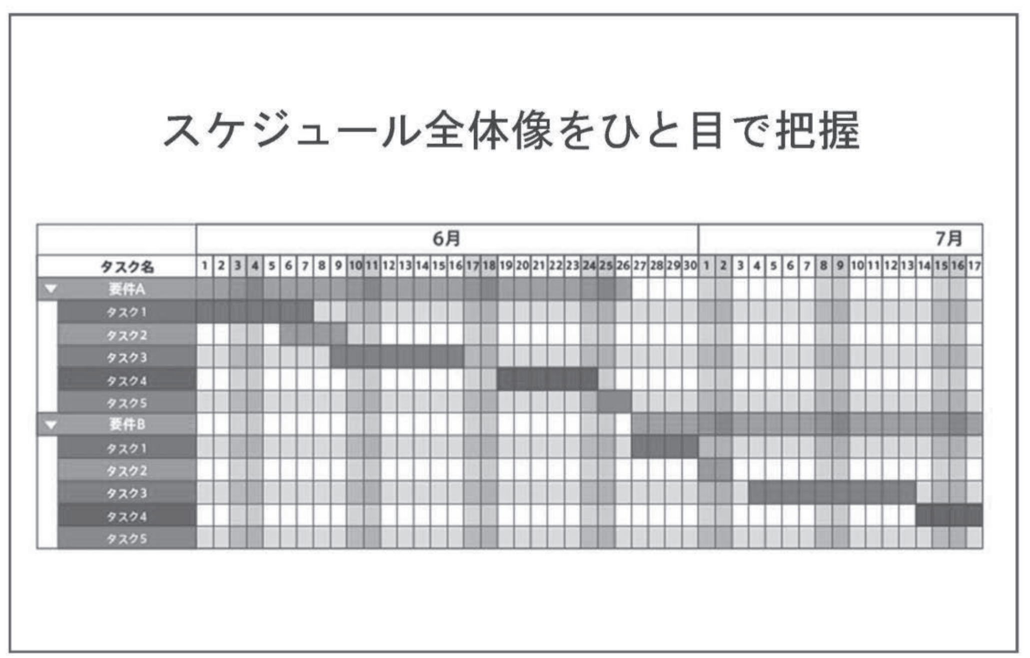
❑ ガントチャート

プロジェクトの作業の流れや進捗を項目に分けて表にしたもの。スケジュールの全体像をひと目で把握できる。

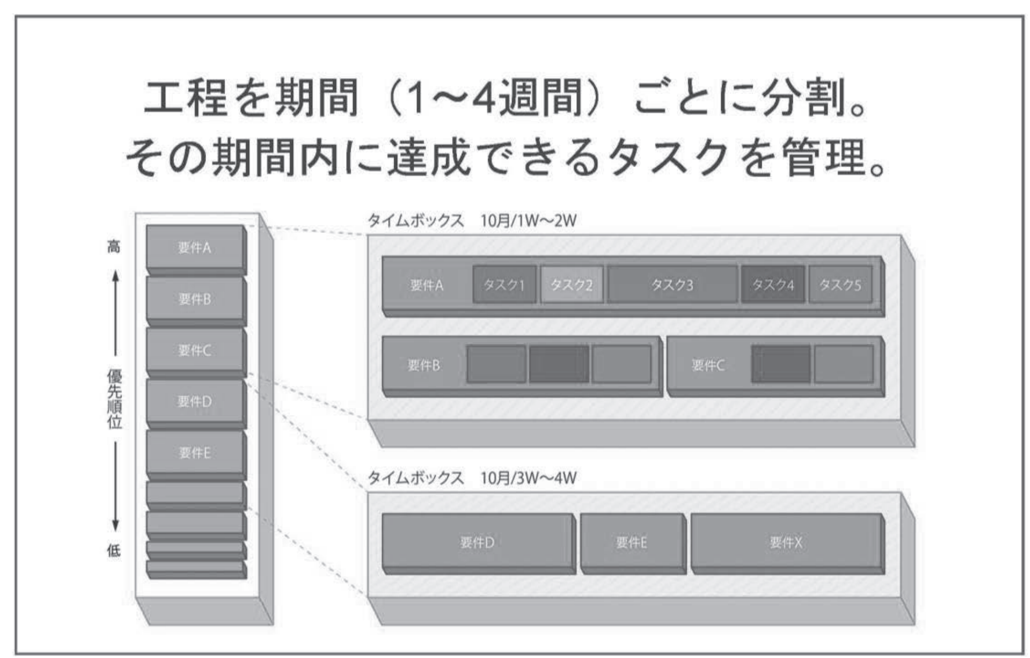
❑ タイムボックス

ゴールまでの工程を固定された一定期間毎に区切ったもので、アジャイルソフトウェア開発では「スプリント」や「イテレーション」と呼ばれる。
期間内に達成できるタスクを各タイムボックスに入れて管理する。

ガントチャートによる管理

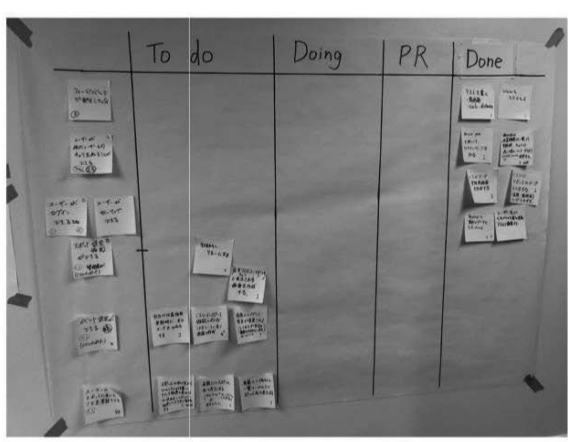


タイムボックスによる管理



タスク管理 - 付箋によるカンバン -

１タスクを付箋１枚で書き出し、Todo(未着手), Doing(着手中), Done(完了)のようなステータスごとにレーンをつくって貼りだすことで仕事を見える化する。



タスク管理 - Trello -

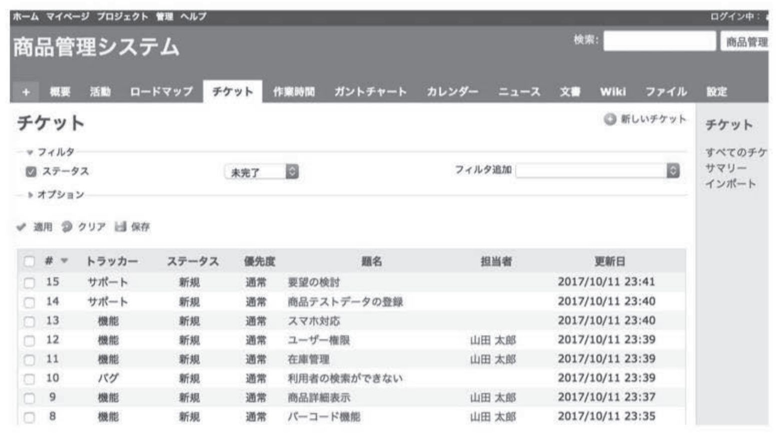
Trelloとは

Web上でタスクをカードとして管理し、
カンバンのようにフェーズごとにリストを作成することでタスクを見える化することができるWebサービス。

タスク管理 - Redmine -

Redmineとは

Redmineは、タスク管理や情報管理を行うためのオープンソースソフトウェアのプロジェクト管理ツール。



Redmineの特徴

一般的にタスク管理にはExcelが良く使われています。Excelは誰でもすぐに使えて便利な反面、チームで使うようになると問題もあります。

Excelの問題点：
・タスク情報をリアルタイムで共有することが困難。
・どれが最新のファイルかわからない。
・一つひとつのタスクの履歴を把握しにくい。

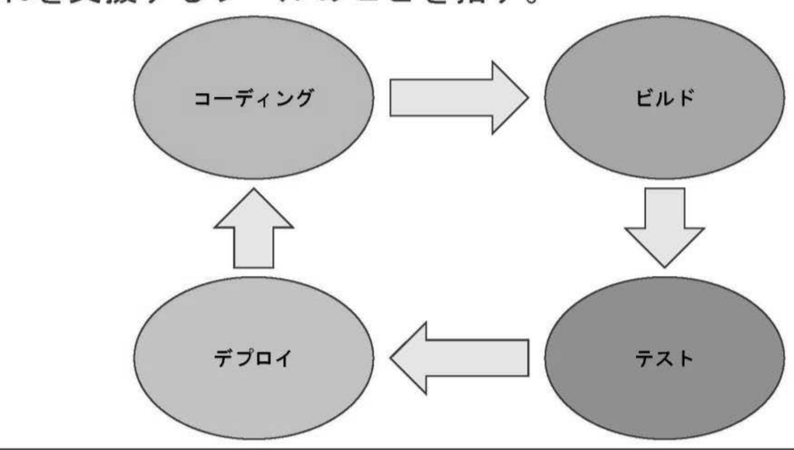
Redmineの利点：
・Web上で管理するためメンバー間でリアルに情報共有できる。
・タスクの履歴が把握しやすい。
・各チームメンバーが更新できるためマネージャーの負担が減る
などのことにより、Excelの課題を解決することができます。

参考: 「はじめる！REDMINE」
http://www.slideshare.net/g_maeda/redmine-2015-54346755

継続的インテグレーションとは

継続的インテグレーション(Continuous Integration / CI)

「ビルドやテストを自動的に繰り返し実行可能とすること」や、
それを支援するツールのことを指す。



継続的インテグレーション - Jenkins -

Jenkins

Javaで書かれたオープンソースのCIツール。
主にJava向けのビルドツールとして開発がされてきたが、
シェルスクリプトやバッチファイルが実行できるため、
汎用性が高くJavaをベースとしないアプリケーションにも
適用可能である。

CIツールをオンプレミス(自社運用)する場合、OSS製品では
Jenkinsを採用する事例が大多数を占める。

継続的インテグレーション - Jenkins -

メリット

- OSSなので、無償で利用可能
- オンプレミスで手軽に構築が可能
- プラグイン(拡張機能)が豊富である

デメリット

- 基本的に自社運用となるので、メンテナンスは自前でやらなければならない
- プロジェクト数が増え、大量のビルドが必要になると、パフォーマンスがボトルネックになりやすい

継続的インテグレーション - CircleCI -

CircleCI

CircleCI社が提供するクラウド型のCIツール。

GithubなどのSCM(ソースコード管理)と連携し、リポジトリにコミットが行われると、自動的にクラウド上に仮想環境を立ち上げ、ビルドを実行する。

ビルドプロセスは設定ファイルを変更することで細かく動作を設定可能で、容易に独自のカスタマイズが可能である。

また、Railsなど主要な言語、FWは事前に大まかな設定がなされており、ほとんど設定ファイルを記載することなくすぐにCIを始めることが可能である。

継続的インテグレーション - CircleCI -

メリット

- クラウドサービスなので、自社でのメンテナンスフリー
- 主要FWには、大まかな設定がなされているので、すぐに利用が開始できる
- ビルド結果のSlack通知など、他サービスとの連携が容易である

デメリット

- 設定ファイルの記法や、処理プロセスの変更には、CircleCIの独自実装部分を深く理解する必要がある
- 無料枠はあるが、基本的に有償のサービスである

第２章　Ruby on Rails 開発環境の準備

第２章　Ruby on Rails 開発環境の準備

2.1 Ruby の基本的な構文の復習

この授業では Ruby on Rails による Web サービスの作り方を解説します。その準備として、プログラミング言語 Ruby の基本的な構文を復習しておきましょう。

2.1.1 数字を順番に出力する

「1 から 100 までの数字を 1 行ずつ出力する」というプログラムをいろいろな方法で書いてみましょう。

①　loop メソッドを使う

loop メソッドを使うと処理を繰り返すことができます。

```
counter = 1
loop do
   puts counter
   break if counter >= 100
   counter += 1
end
```

まず counter という変数を定義しています。puts メソッドで出力してから counter に 1 を加算するという処理を loop メソッドで繰り返し、counter の値が 100 以上になったら break で繰り返しから脱出しています。
loop はメソッドなので、do 〜　end でブロック引数を渡しています。

②　while を使う

while を使うと、「ある条件が成り立っている間だけ繰り返す」という処理を定型的に書くことができます。

```
counter = 1
while counter <= 100
  puts counter
  counter += 1
end
```

while は loop メソッドと違って制御構造なので、do〜end のブロックは使えません。
while の繰り返しも break で脱出することができるので、次のように書くこともできます。

```
counter = 1
while true
  puts counter
```

```
  break if counter >= 100
  counter += 1
end
```

while の代わりに until を使うと、「ある条件が成り立つまで繰り返す」という処理を書くことができます。ここまでに出てきた while を使ったコードと同じ動作をするコードを until で書いてみましょう。

③ Range オブジェクトと each メソッドを使う

1..100 と書くことで、1 から 100 までの範囲を表す Range クラスのインスタンスを作ることができます。

```
(1..100).each do |n|
  puts n
end
```

Range クラスには each というインスタンスメソッドが定義されています。このメソッドを使うと、範囲内の要素を順番に取り出すことができます。 do〜end でブロック引数を渡しています。

繰り返しの脱出処理や counter 変数をコードから消すことができて、よりすっきりとした実装になりました。

④ Numeric オブジェクトのメソッドを使う

数値を表現する Numeric クラスにも、繰り返しを表現するための便利なメソッドが用意されています。以下はその一例です。

```
1.step(100) do |n|
  puts n
end

100.times do |n|
  puts n + 1
end
```

2.1.2 FizzBuzz 問題

ここまでに実装した「1 から 100 までの数字を 1 行ずつ出力する」というプログラムに、以下の動作を追加してみましょう。

- 数字が 3 で割り切れる場合には、数字の代わりに"Fizz"を出力する
- 数字が 5 で割り切れる場合には、数字の代わりに"Buzz"を出力する
- 数字が 3 と 5 の両方で割り切れる場合には、数字の代わりに"FizzBuzz"を出力する

これは「FizzBuzz 問題」という名前でよく知られている問題です。
正しく実装すると以下のような文字列が出力されます。

```
1
2
Fizz
4
Buzz
Fizz
7
8
Fizz
Buzz
11
Fizz
13
14
FizzBuzz
16
17
...
```

ここからは次の実装をベースにして進めることにします。

```
1.step(100) do |n|
  puts n
end
```

① if を使う

if を使って単純に実装すると以下のようになります。

```
1.step(100) do |n|
  if n % 15 == 0
    puts 'FizzBuzz'
  elsif n % 3 == 0
    puts 'Fizz'
  elsif n % 5 == 0
    puts 'Buzz'
  else
    puts n
  end
end
```

elsif, else を使って複数の条件を表現しています。
以下のような順番で条件分岐を作ってしまうとうまく動きません。理由を考えてみましょう。

```
  if n % 3 == 0
    puts 'Fizz'
  elsif n % 5 == 0
    puts 'Buzz'
  elsif n % 15 == 0
    puts 'FizzBuzz'
  else
    puts n
  end
```

② if と配列を使う

配列を使うと以下のように書くことができます。

```
1.step(100) do |n|
  words = []
  words << 'Fizz' if n % 3 == 0
  words << 'Buzz' if n % 5 == 0
  if words.empty?
    puts n
  else
    puts words.join
  end
end
```

words オブジェクトに対してどんなメソッドが呼ばれているでしょうか？　また、そのメソッドにはどのような引数が渡され、どんなクラスのインスタンスが返り値として戻されているでしょうか？

2.1.3 メソッドを定義する

与えられた数に対して、数字、Fizz, Buzz, FizzBuzz のどれかを返すメソッドを定義してみましょう。

```
def fizzbuzz(number)
  words = ''
  words << 'Fizz' if number % 3 == 0
  words << 'Buzz' if number % 5 == 0
  words.empty? ? number : words
end

1.upto(100) { |n| puts fizzbuzz(n) }
```

メソッドの内容や繰り返し処理はこれまでに出てきたコードとほぼ同じですが、細かい違いがあります。見比べてみてください。
以下のような引数を受け取るメソッドをそれぞれ書いてみましょう。

```
# 'Fizz', 'Buzz'の代わりの値をキーワード引数で指定できるメソッド
fizzbuzz(15, three: 'Fizzzzz', five: 'Buzzzzz')
# => FizzzzzBuzzzzz

# 任意の数の引数を受け取るメソッド
fizzbuzz(3, 5, 15, 17)
# => ['Fizz', 'Buzz', 'FizzBuzz', 17]

# 上記の 2 つを合わせたメソッド
fizzbuzz(3, 5, 15, 17, three: 'Fizzzzz', five: 'Buzzzzz')
# => ['Fizzzzz', 'Buzzzzz', 'FizzzzzBuzzzzz', 17]
```

2.1.4 Integer クラスを拡張する

最後に、整数を表す Integer クラスに fizzbuzz メソッドを組み込む例を示します。

```
class Integer
  def fizzbuzz
    words = ''
    words << 'Fizz' if self % 3 == 0
    words << 'Buzz' if self % 5 == 0
    words.empty? ? self : words
  end
end

15.fizzbuzz #=> FizzBuzz
```

2.1.5 まとめ

ここでは Ruby on Rails を学ぶ前の準備として、Ruby の以下の構文を復習しました。

- 変数
- loop, while, メソッドを使った繰り返し
- 繰り返しの脱出
- メソッドの呼び出し
- if を使った条件分岐
- メソッドの定義
- 既存のクラスの拡張

2.1.6 発展課題

- 1 から 100 までの奇数をいろいろな方法で出力してみましょう。
- case を使って FizzBuzz プログラムを書いてみましょう。
- 「数字が 7 で割り切れる場合には、数字の代わりに"Jazz"を出力する」というルールを追加した FizzBuzz プログラムを書いてみましょう。
- Enumerable のメソッドをうまく組み合わせて、数値計算の演算子（+, -, *, /, %）を使わずに FizzBuzz プログラムを書いてみましょう。

2.2 Ruby on Rails：開発環境の構築

ここでは、Ruby on Rails の開発環境を構築する手順を説明します。

Ruby on Rails は、その名前にあるように Ruby や Rails が必要です。また、その Ruby や Rails の動作に必要な環境やソフトウェアのインストールも必要です。
環境構築は、オペレーティングシステム(OS)によって、インストールの手順が異なります。ここでは、以下の 7 種類の構築手順を説明します。

① macOS 用セットアップ(10.9 から 10.14)
② Windows 用セットアップ(WSL が利用できる Windows 10 64bit)
③ Windows 用セットアップ(Windows 7 や 32bit の Windows の場合等)
④ Linux 用セットアップ(Ubuntu 18.04 LTS)
⑤ 仮想環境(VirtualBox)
⑥ クラウドサービス(AWS Cloud9)

どのオペレーティングシステムを利用する場合でも、インターネットへ接続が必要です。その際、インストール手順によっては、ギガバイト単位でファイルをダウンロードすることもありますので、ご注意ください。

ダウンロードするファイルは、一般的にセキュリティの問題やバグで修正があればバージョン番号が上がります。以下の手順に示されているファイルのバージョンと比べて、メンテナンスバージョン(たとえば、v1.2.3 でいうと 3 のところ)が大きくなっていることがあります。その場合は、新しいものをダウンロードしてください。

また、メジャーバージョン(たとえば、v1.2.3 でいうところの 1 のところ)やマイナーバージョンが異なる場合は、機能の追加や廃止等の理由から期待している動作をしない可能性もありますので、ご注意ください。

2.2.1 macOS 用セットアップ(10.9 から 10.14)

① オペレーティング・システムのバージョンを調べます

Apple メニューをクリックして About This Mac を選択します。
開いたウィンドウに使用しているオペレーティングシステムのバージョンが表示されます。

Version 10.9 から Version 10.14 であることを確認します。Dock から、Launchpad、その他、ターミナルをクリックして、ターミナル画面を開きます。
以下、ターミナル画面での作業が続きます。表示されている$の後ろからコマンドを入力します。

② C 言語のコンパイラ等が必要なので、Xcode の Command line tools をインストールします

```
xcode-select –install
```

③ パッケージ管理システムの Homebrew をインストールします

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

④ Ruby のバージョン管理システムの rbenv をインストールします

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

⑤ rbenv を使って Ruby をインストールします

```
rbenv install 2.6.5
```

もし、OpenSSL に関するエラーが表示された場合は、以下のコマンドを実行して、再試行してください。

```
brew install curl-ca-bundle
cp /usr/local/opt/curl-ca-bundle/share/ca-bundle.crt ruby -ropenssl -e
'puts OpenSSL::X509::DEFAULT_CERT_FILE'
パッケージ管理システムの yarn をインストールします
brew install yarn
```

⑥ 手順 5 でインストールした Ruby を常用の Ruby として設定します

```
rbenv global 2.6.5
```

⑦ Rails をインストールをします

```
gem install rails -v 5.1.3 --no-document
```

⑧ 動作確認をします

```
rails new sample
cd sample
rails g scaffold book
rails db:migrate
rails server
```

ブラウザの URL 欄に http://localhost:3000/books と入力して、画面が表示されれば成功です。
動作が確認できたら、Control キーと C キーを同時に押して、Rails server を停止しましょう。

⑨　コードエディタをインストールします

コードエディタの一例です。
すでにお気に入りのコードエディタをインストールされている場合は、それらをお使いください。

- ATOM
- Visual Studio Code
- Sublime Text

2.2.2 Windows 用セットアップ(WSL が利用できる Windows 10 64bit)

Windows 10 で利用できる、WSL と Ubuntu のアプリで環境を構築します。
以下の手順では、Windows の管理者のパスワードが必要です。

①　Windows Subsystem for Linux をインストールします

画面左下のスタートアイコンをクリックします。
歯車の設定アイコンをクリックします。
アプリをクリック、**プログラムと機能**をクリックします。
Windows の機能の有効化または無効化をクリックします。
パスワードの入力を求められたら管理者のアカウント名を確認して、パスワードを入力します。
Windows の機能ダイアログが表示されるので、**Windows Subsystem for Linux** をチェックします。
OK ボタンをクリックします。
インストールにしばらく時間がかかります。
再起動が必要であれば再起動します。

③　Microsoft Store から Ubuntu をインストールします。

画面左下のスタートアイコンをクリックします。
Microsoft Store をクリックします。
ルーペの検索アイコンをクリックします。
キーワードを入力する枠に ubuntu と入力して検索します。
Ubuntu と表示のあるアプリ(18.04 や 16.04 ではありません)をクリックします。
入手ボタンをクリックします。
複数のデバイスで使用すると確認される場合がありますが、ここでは必要ありません
をクリックします。
インストールにしばらく時間がかかります。
起動ボタンをクリックします。
準備にしばらく時間がかかります。

```
Enter new UNIX username:
```

と表示されたら、任意のユーザー名を入力します。

```
Enter new UNIX password:
```

と表示されたら、任意のパスワードを入力します。
今使っている Windows アカウントのパスワードや管理者のパスワードではありません。

```
Tetype new UNIX password:
```

と表示されたら、同じパスワードを入力します。
２つのパスワードが一致すると

```
xxxx@yyyy: $
```

と表示されます。xxxx には、先ほどのユーザー名、yyyy は Windows のコンピュータ名
が表示されます。
ユーザー名と、パスワードは忘れないようにしましょう。
念のため、日本時間に合わせます。

```
sudo dpkg-reconfigure tzdata
```

補足

もし、コマンドをテキストから貼り付ける場合は、マウスの右ボタン(右利きの場合)
をクリックすると貼り付けることができます。

パスワードには、UNIX のパスワードを入力します。

Asia と Tokyo を選択します。ただし OK を選択するためにはマウスは使えないので、TAB キーを数回押して移動して選択してください。

```
Date
```

と入力して、Asia/Tokyo と表示されれば設定できています。

④　UNIX 環境を最新化します

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

⑤　Ruby のバージョン管理システムの rbenv をインストールします

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

⑥　rbenv を使って Ruby をインストールします

```
rbenv install 2.6.5
```

⑦　手順 5 でインストールした Ruby を常用の Ruby として設定します

```
rbenv global 2.6.5
```

⑧　Rails をインストールをします

```
gem install rails -v 5.1.3 --no-document
```

⑨　動作確認をします

```
rails new sample
cd sample
rails g scaffold book
rails db:migrate
rails server
```

ブラウザの URL 欄に http://localhost:3000/books と入力して、画面が表示されれば成功です。
動作が確認できたら、CTRL キーと C キーを同時に押して、Rails server を停止しましょう。

補足

UNIX 環境で、
explorer.exe .
とすることで、作業しているディレクトリ(カレントディレクトリ)を Windows の
エクスプローラで参照できます。 この場所を Windows 上にインストールしたコードエディタで参照することでファイルを編集できます。

⑩　コードエディタをインストールします。

コードエディタの一例です。エディタは Windows 用をダウンロードして、インストールします。 すでにお気に入りのコードエディタをインストールされている場合は、それらをお使いください。

- ATOM
- Visual Studio Code
- Sublime Text

2.2.3 Windows 用セットアップ(Windows 7 や 32bit の Windows の場合等)

プログラム開発では、同じファイル名で、拡張子だけ違うファイルを取り扱うことが多々ありますので、エクスプローラの設定で、フォルダーオプションの表示から、登録されている拡張子は表示しないのチェックを外すことをお勧めします。

①　Ruby のインストーラをダウンロードします。

以下の URL を開きます。
https://rubyinstaller.org/downloads/
英語でページが表示されますが、WITH DEVKIT の中から
64bit の場合は、Ruby+Devkit 2.6.5-1 (x64)をクリックしてダウンロードします。
32bit の場合は、Ruby+Devkit 2.6.5-1 (x86)をクリックしてダウンロードします。
ダウンロードにしばらく時間がかかります。

② Ruby のインストール

ダウンロードフォルダからダウンロードしたファイル(rubyinstaller-devkit-2.6.5-1-x64.exe 等)をダブルクリックして実行します。
実行の確認、使用ライセンスの確認は、それぞれ実行と I accept the License を選択して進めます。
インストール先の場所(パス)等を確認されます。
インストール先は、C:¥Ruby26-x64(そのまま)
Add Ruby executables to your PATH にチェック(そのまま)
Associate .rb and .rbw files with this Ruby installation にチェック(そのまま)
Use UTF-8 as default external encoding.にチェック
Install ボタンをクリックします。
Select Components の画面が表示されますので、2 つがチェックされた状態で
Next ボタンをクリックします。

インストールが成功すると、Completing the Ruby 2.6.5-1-x64 with MSYS2 Setup Wizard と表示されますので、Finish ボタンをクリックします(インストールは続きます)。
コマンドプロンプトが立ち上がって MSYS2 のインストールに進みますのでデフォルトの選択肢(何も入力せずにエンターキー)を選びます。

必要なファイルをインターネットからダウンロードして、インストールがされるのでしばらく時間がかかります。
再度、1,2,3 から選択するように表示されますのでデフォルトの選択肢(何も入力せずにエンターキー)を選びます。(コマンドプロンプトの画面が閉じられます)

③ Rails で使用している外部プログラム(JavaScript ライブラリ等)をインストールします

以下の URL を開きます。
https://nodejs.org/ja/download/
64bit の場合は、Windows Installer(.msi)の 64-bit をクリックしてダウンロードします。
32bit の場合は、Windows Installer(.msi)の 32-bit をクリックしてダウンロードします。
ダウンロードフォルダから、ダウンロードしたファイルをダブルクリックして実行します。
実行の確認、使用ライセンスの確認は、それぞれ実行と I accept the License を選択して進めます。

以下の URL を開きます。
https://yarnpkg.com/ja/docs/install#windows-stable

OS が Windows、バージョンが安定板(1.19.1)になっていることを確認してインストーラをダウンロードするボタンをクリックします。
ダウンロードフォルダから、ダウンロードしたファイルをダブルクリックして実行します。

④　Rails をインストールします

画面左下のスタートアイコンから、すべてのプログラム、アクセサリ、コマンドプロンプトをクリックして、コマンドプロンプトの画面を作成します。

```
gem install rails -v 5.1.3 --no-document
```

⑤　データベース関連をインストールします

以下の URL を開いて、SQLite のダウンロードとインストールをします。

https://www.sqlite.org/download.html

英語で表示されますが、Precompiled Binaries for Windows の中から
64bit の場合、sqlite-dll-win64-x64-3300100.zip をクリックしてダウンロードします。
32bit の場合、sqlite-dll-win32-x86-3300100.zip をクリックしてダウンロードします。
64bit、32bit ともに、sqlite-tools-win32-x86-3300100.zip もダウンロードします。
ダウンロードフォルダからダウンロードした 2 つの圧縮ファイルをダブルクリックして展開します。

```
sqlite3.def
sqlite3.dll
```

```
sqldiff.exe
sqlite3.exe
sqlite3_analyzer.exe
```

2 つと 3 つのファイルが展開されますので、今回は、5 つのファイルを先ほど Ruby をインストールした以下のフォルダの bin の中にコピーします(64bit の場合)。

```
C:¥Ruby26-x64¥bin
```

Rails とデータベースを接続するアダプタをソースコードからコンパイルしてインストールします。

```
gem install sqlite3 -v '1.4.1' --source 'https://rubygems.org/'
```

補足

以下のようなエラーが表示された場合も、この手順で gem とデータベース SQLite をインストールします。

エラーメッセージ(例)

An error occurred while installing sqlite3 (1.4.1), and Bundler cannot continue.

Make sure that gem install sqlite3 -v '1.4.1' --source 'https://rubygems.org/'

⑥ 動作確認をします

```
rails new sample
cd sample
rails g scaffold book
rails db:migrate
rails server
```

ブラウザの URL 欄に http://localhost:3000/books と入力して、画面が表示されれば成功です。
動作が確認できたら、CTRL キーと C キーを同時に押して、Rails server を停止しましょう。

補足

以下のようなエラー表示がされた場合は
'x64_mingw' is not a valid platform
　Gemfile を修正します。
gem 'tzinfo-data', platforms: [:mingw, :mswin, :x64_mingw]
の行を
gem 'tzinfo-data', platforms: [:mingw, :mswin]
となるように修正、保存して、コマンド
bundle update
を実行します。

⑦ コードエディタをインストールします

コードエディタの一例です。 すでにお気に入りのコードエディタをインストールされている場合は、それらをお使いください。

- ATOM
- Visual Studio Code
- Sublime Text

2.2.4 Linux 用セットアップ(Ubuntu)

① Ruby のインストールに必要なコマンド等をインストールします

```
sudo apt update
sudo apt upgrade -y
sudo apt install git curl net-tools gcc make -y
```

② Ruby on Rails の環境に必要なライブラリ等をインストールします

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

Ubuntu 16.04LTS で libssl1.0-dev が見つからないというエラーが表示された場合、libssl-dev と修正して再試行してください。

③ Ruby のバージョン管理システムの rbenv をインストールします

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

④ rbenv を使って Ruby をインストールします

```
rbenv install 2.6.5
```

⑤ 手順 4 でインストールした Ruby を常用の Ruby として設定します

```
rbenv global 2.6.5
```

⑥ Rails をインストールをします

```
gem install rails -v 5.1.3 --no-document
```

⑦ 動作確認をします

```
rails new sample
cd sample
rails g scaffold book
rails db:migrate
rails server
```

ブラウザの URL 欄に http://localhost:3000/books と入力して、画面が表示されれば成功です。
動作が確認できたら、CTRL キーと C キーを同時に押して、Rails server を停止しましょう。

⑧ コードエディタをインストールします。

コードエディタの一例です。 すでにお気に入りのコードエディタをインストールされている場合は、それらをお使いください。

- ATOM
- Visual Studio Code
- Sublime Text

2.2.5 仮想環境(VirtualBox)

コンピュータの中に、別のコンピュータの環境を構築できる仮想環境ソフトウェアとして VirtualBox や VMware、Hyper-V 等がありますが、ここでは VirtualBox を使用します。
また、仮想環境内に構築する開発環境は Ubuntu 18.04 LTS(32bit の場合は Ubuntu 16.04 LTS)を使用します。
仮想環境はウィンドウやマウスを使用する GUI の環境で構築しますので、使用するコンピュータが搭載しているメモリは 4GB 以上、ハードディスクや SSD 等の補助記憶装置には空き領域が 25GB 以上あることを確認してください。

① ダウンロードします

以下の URL を開きます。

https://www.virtualbox.org/wiki/Downloads

使用しているコンピュータ(ホスト)の OS によって Windows hosts、OS X hosts をクリックします。

ホストが 32bit の場合は、Version 5.2 の最新を以下の URL からダウンロードします。

https://www.virtualbox.org/wiki/Download_Old_Builds_5_2

② インストールします

ダウンロードフォルダからダウンロードしたファイルをダブルクリックして実行します。

インストールの途中に、構成の確認がありますが、変更せず Next ボタンをクリックして進めます。

また、仮想のネットワークアダプタを追加する確認画面が表示されますが、インストールを選択して進めます。

③ 仮想環境の中に構築する OS のインストーラをダウンロードします

64bit の場合は、以下の URL から Ubuntu Desktop の 18.04 LTS をクリックしてダウンロードします。

https://ubuntu.com/#download

32bit の場合は、以下の URL から 16.04 LTS の 32-bit PC(i386) desktop image をダウンロードします。

http://releases.ubuntu.com/xenial/

④ 仮想環境の構築

VirtualBox マネージャが表示されていなければ、画面左下のスタートアイコン、**すべてのプログラム**、Oracle VM VirtualBox、Orale VM VirtualBox をクリックして起動します。

新規アイコンをクリックします。

仮想環境の構成を決定します。32bit の場合は適宜調整してください。

```
名前[(例:Ruby on Rails)]
タイプ[Linux]
バージョン[Ubuntu(64-bit)]
メモリ[1024]MB (ホストのメモリに余裕があれば 2048MB 程度が良いです)
仮想ハードディスク[仮想ハードディスクを作成する]
仮想ハードディスクの種類[VDI/可変サイズ]
仮想ハードディスクのサイズ[10.00GB] (ホストのディスク空き容量に余裕があれば 20.00GB 程度が良いです)
[作成]ボタンをクリックして設定完了
```

手順 3 でダウンロードした OS のイメージを作成した仮想環境に割り当てます。

[ストレージ]から、[IDE セカンダリマスター]の右横に表示されている
[光学ドライブ]空
をクリックして、[ディスクイメージを選択...]をクリックします。
ダウンロードフォルダから、手順 3 でダウンロードしたファイルを選択します。

起動アイコンをクリックして仮想環境を起動します。
言語から日本語をクリックします。
Ubuntu をインストールボタン をクリックします。
キーボードの選択は、コンピュータに合うキーボードを選択し、アットマーク(@)や、ダブルクオート(")の文字を入力し、正しく表示されることを確認して
続けるボタンをクリックします。
通常のインストールで
ディスクを削除して Ubuntu をインストール を選択して**インストール**、**続ける**をクリックします。

注意

ここでは、仮想環境の中で作業をしているので、ディスクとは仮想環境の中の仮想ディスクのことです。

場所(タイムゾーン)の選択で Tokyo を選択します。
利用するアカウント、コンピュータ名、パスワードを入力して進めます。
アカウントと、パスワードは忘れないようにしましょう。
インストールにしばらく時間がかかります。
インストール完了のメッセージが表示されたら今すぐ再起動するボタンをクリックします。

```
Please remove the installation medium, then press ENTER:
```

と表示されるのでエンターキーを押します。

⑤ 仮想環境へのログインとゲスト環境用追加ソフトウェアのインストール

インストールのときに入力したアカウントを選択、パスワードを入力してログインします。
仮想環境の中で Ubuntu の新機能を説明するウィンドウが表示されたら、画面上部の Ubuntu へようこそから終了をクリックします。Ubuntu 16.04 LTS の場合は、キー操作の画面が表示されますので、その画面を終了します。

仮想環境の中の左下から、アプリケーションを表示するアイコンをクリックし、検索ワードに term と入力します。
Ubuntu 16.04 LTS の場合は、仮想環境の中の左上から、コンピュータを検索アイコンをクリックし、検索ワードに term と入力します。
端末が表示されるので、クリックします。
日本語環境の構築をします(参考：https://www.ubuntulinux.jp/japanese)。

Ubuntu 18.04 LTS の場合:

```
wget -q https://www.ubuntulinux.jp/ubuntu-ja-archive-keyring.gpg -O- |
   sudo apt-key add -
wget -q https://www.ubuntulinux.jp/ubuntu-jp-ppa-keyring.gpg -O- | sudo
   apt-key add -
sudo wget https://www.ubuntulinux.jp/sources.list.d/bionic.list -O
   /etc/apt/sources.list.d/ubuntu-ja.list
```

Ubuntu 16.04 LTS の場合:

```
wget -q https://www.ubuntulinux.jp/ubuntu-ja-archive-keyring.gpg -O- |
   sudo apt-key add -
wget -q https://www.ubuntulinux.jp/ubuntu-jp-ppa-keyring.gpg -O- | sudo
   apt-key add -
sudo wget https://www.ubuntulinux.jp/sources.list.d/xenial.list -O
   /etc/apt/sources.list.d/ubuntu-ja.list
```

共通:

```
sudo apt update
sudo apt upgrade -y
sudo apt install gcc make perl -y
sudo apt install ubuntu-defaults-ja
```

右上の電源アイコンをクリックし、さらに電源アイコンをクリック、再起動をクリックします。
再度、ログインします。
VirtualBox のメニューからデバイス、Guest Additions CD イメージの挿入...をクリックして、実行するを選択、パスワードを入力します。
インストールが完了すると、

```
Press Return to close this window...
```

と表示されるのでエンターキーを押します。

⑥　Rails 環境の構築

ここからは、Linux 用セットアップ(Ubuntu)と同じですので、Linux 用セットアップを参照してください。

2.2.6 クラウドサービス(AWS Cloud9)

AWS Cloud9 のサイト

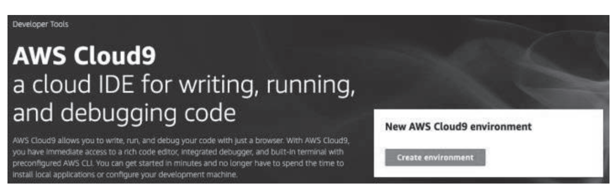
https://aws.amazon.com/jp/cloud9/

① ブラウザを選択する

Internet Explorer を利用している場合は、Google Chrome または Firefox をインストールしてください。（一部の機能が IE では動かない場合があります。）

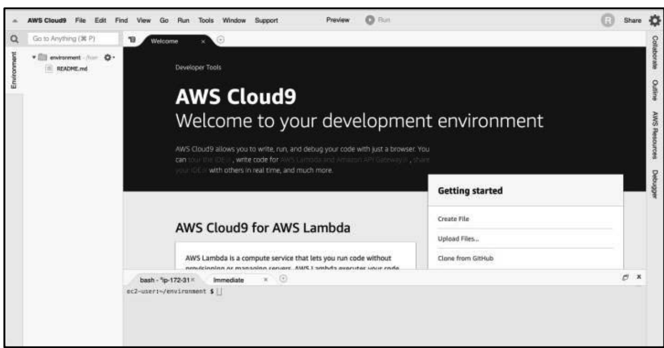
② アカウントを作成する

AWS のアカウントを作成し、AWS Cloud9 コンソールにサインインしましょう。 具体的には 個人ユーザーの AWS Cloud9 セットアップ - AWS Cloud9 の手順で作業を進めてください。 サインイン後に Welcome to AWS Cloud9 にアクセスして次のような画面が出たら OK です。



③ Ruby on Rails の開発用に Enviroment を設定する

Welcome to AWS Cloud9 にアクセスしてください。（サインインしていない場合は先にサインインをしてください）Create environment をクリックします。Environment name には好きな名前をつけましょう。Description は任意なので空欄でも構いません。Next Step をクリックしましょう。 Configure settings では Platform に Ubuntu Server を指定してください。その他は初期設定のままで OK です。Next Step をクリックしてください。 Review で入力内容を確認します。Create environment をクリックしましょう。 利用可能になるまで少し待ってください。 次のような画面が開いたら OK です。



以降の手順ではこの画像の下部にあるターミナルを使います。

④　標準でインストールされている RVM をアンインストールする

1. RVM 関連ファイルの削除

```
/usr/bin/sudo rm -rf $HOME/.rvm
```

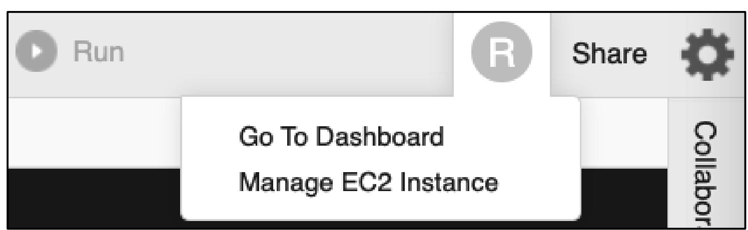
2. RVM 関連設定の削除

```
sed -i -e '/rvm/d' ~/.bashrc
```

3. AWS Cloud9 のインスタンスを再起動して操作を反映させる

AWS Cloud9 を再起動して $GEM_HOME, $GEM_PATH を更新します。

画面右上のメニューから Manage EC2 Instance をクリックして EC2 の管理画面に移動しましょう。



アクション をクリックして インスタンスの状態 のメニューから 再起動 をクリックしてください。

AWS Cloud9 の画面に戻りましょう。少し待って利用可能な状態になったら再起動は完了です。
これで RVM が正常にアンインストールされました。

⑤　Rails のインストール

次のコマンドを 1 行ずつ実行すると各種インストールが完了します。

```
sudo apt-get update –y
sudo apt-get install -y snapd
sudo snap install node --classic --channel=10
sudo apt-get --ignore-missing install build-essential git-core curl openssl
    libssl-dev libcurl4-openssl-dev zlib1g zlib1g-dev libreadline6-dev libyaml-
    dev libsqlite3-dev libsqlite3-0 sqlite3 libxml2-dev libxslt1-dev libffi-dev
    software-properties-common libgdm-dev libncurses5-dev automake autoconf
    libtool bison postgresql postgresql-contrib libpq-dev pgadmin3 libc6-dev –y
sudo apt-get install imagemagick --fix-missing –y
git clone https://github.com/rbenv/rbenv.git ~/.rbenv
echo 'export PATH="$HOME/.rbenv/bin:$PATH"' >> ~/.bashrc
echo 'eval "$(rbenv init -)"' >> ~/.bashrc
source ~/.bashrc
git clone https://github.com/rbenv/ruby-build.git "$(rbenv root)"/plugins/ruby-
    build
rbenv install 2.6.5
rbenv global 2.6.5
gem install rails -v 5.1.3 --no-document
```

⑥　プロジェクトを作成する場合に

左側はフォルダとファイルを表示、選択できます。 中央部はエディタです。ここでファイルを編集します。 下部はターミナルです。ここでコマンドを実行します。 必要なものは全てブラウザにあります。ブラウザのほかにエディタやターミナルを起動する必要はありません。 ガイドやチュートリアルを読む場合には、(Windows マシンを利用している場合でも)Linux 用のコマンドを使ってください。コマンドはクラウド上で実行され、その環境が Linux マシンだからです。

ガイドやチュートリアルで、サーバを起動する時のコマンドは `rails server` ではなく `rails server -b 0.0.0.0` を実行してください。何も指定しなかった場合はコマンドを実行した環境以外からはアクセスできないため、操作しているマシンからも表示がうまく行えません。 ガイドやチュートリアルで、ブラウザから例えば `http://localhost:3000` へアクセスする場合は、アドレス欄に入力するのではなく、画面上部から ‘Preview’ - ‘Priview Running Application’ を選ぶことで同じ操作ができます。 例えば、`http://localhost:3000/posts` へアクセスしたい場合は、’Preview’ - ‘Priview Running Application’ を選んだあと、’/posts’ をアドレス欄の末尾に加えてください。

⑦　動作確認

```
rails new sample
cd sample
rails g scaffold book
rails db:migrate
rails server -b 0.0.0.0
```

ブラウザの URL 欄に http://localhost:3000/books と入力して、画面が表示されれば成功です。
動作が確認できたら、CTRL キーと C キーを同時に押して、Rails server を停止しましょう。

2.3 Git 基礎 1

2.3.1 Git

Git とは

①分散型バージョン管理システム

「Git とは何か？」という話になると、必ず出てくるのがこの分散型バージョン管理システムという言葉です。

この言葉をパッと聞いただけでは、どういうものなのか中々イメージしづらいと思います。

なので、今回は分散型とバージョン管理システムの２つに分けて説明していきましょう。

②バージョン管理システムについて

バージョン管理システムとは、ファイル自体とファイルの変更履歴をセットで管理するシステムです。

「いつ」「誰が」「どんな内容で」ファイルを変更したのかを記録していき、複数人で共有することが出来ます。

普通のファイル共有等を用いた管理方法と比べると、

ファイルを変更した後でも、特定の時点まで戻ってやり直す事が出来る

変更した履歴を付けているので、その差分を見ることが出来る

「他の人が変更したファイルを上書きして、他の人の変更内容を消す」等の事故が起きにくい。といった利点があります。

③ 集中型と分散型

バージョン管理システムでは、「リポジトリ」というファイルやその変更履歴をまとめておく場所が定義されていて、

そのリポジトリの中で管理されているファイルに対して履歴を付けます。

見かけ上で言うなら、リポジトリ = フォルダ(ディレクトリ)と考えてもいいでしょう。

このリポジトリの扱い方によって、バージョン管理システムは２つの方式に分類されます。

【集中型】

・リポジトリがリモート(インターネット上のサーバ、社内ネットワーク内のサーバ 等)に一つだけある

・ローカル（自分の PC 等）で編集を行い、ファイルの変更はリモートに直接アクセスして行う

→ オンライン作業, 変更の履歴を付けるのはリモートだけ

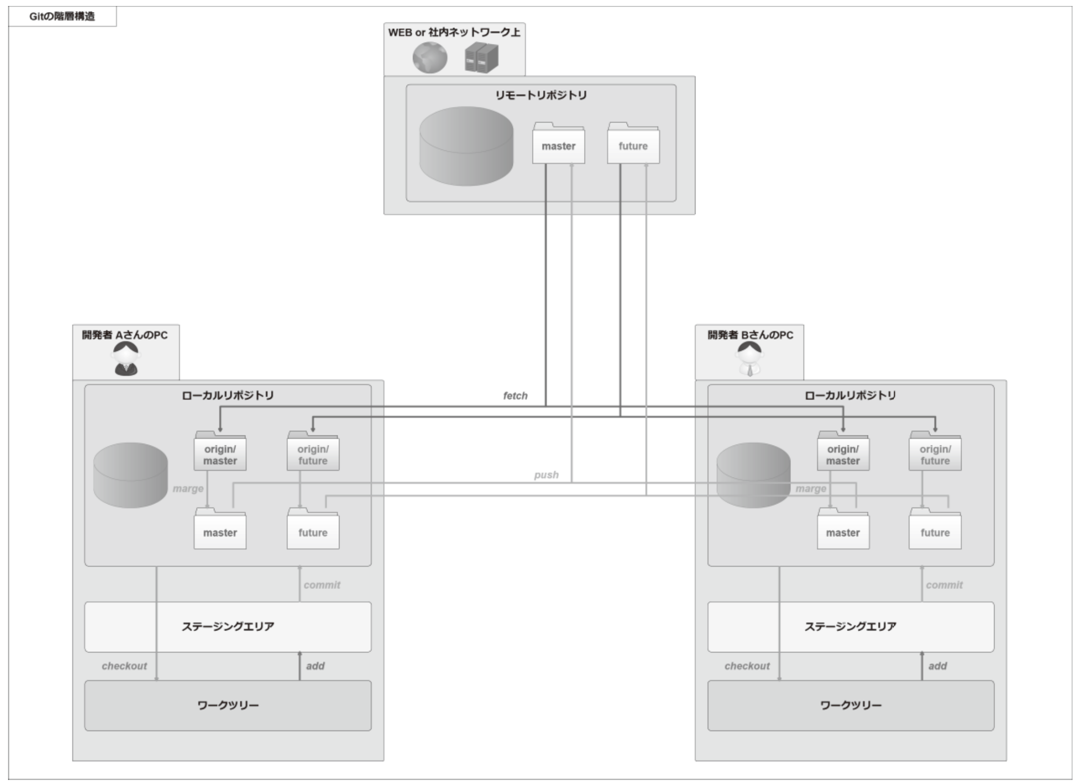
【分散型】

・リモートの他に、ローカルにもリモートから複製したリポジトリを作成出来る(リポジトリが複数存在する)
・毎回の変更はローカルに積み重ねておいて、ある程度まとまった段階でリモートに反映させる
→ オフラインでも作業可, 変更の履歴を付けるのはローカルとリモートの両方(リモートへはローカルから反映)

Git の仕組み

① Git の階層構造

Git は何段もの階層構造になっていて、それぞれの階層に役割があります。



【リモートリポジトリ】
・共有,管理のベースとなるリポジトリ
・ネットワーク上に作成する
・全員で共有するファイルを保管しておく場所

【ローカルリポジトリ】
・リモートリポジトリから複製した、またはローカルで新規に作成したリポジトリ
・実際に作業する PC 上に作成する
・個人で作業するためのファイル、または編集済みのファイル等を保管しておく場所

【ステージングエリア】
・ローカルリポジトリに登録する前の一時保存場所

【ワークツリー】
・実際に作業を行う PC 上の領域
・画面上のフォルダ(ディレクトリ)に表示されているものの状況が反映される

② ブランチ

Git には「ブランチ」というものを作成する機能があります。
実は、最初にリポジトリへ変更履歴を付けた時には、自動で master というブランチが 作成されるようになっています。
自身で作成する際には、この master 等の既存ブランチから派生させる形で作成していきます。
ブランチを作成する時には、
ファイルが格納されているフォルダ(既存ブランチ)を丸ごとコピーして、新しいフォルダ(新規ブランチ)を作成している というとイメージを掴みやすいかもしれません。

ブランチはそれぞれが独立しているので、他の人に影響を与えずに自分の作業を進められるようになります。

作成したブランチは、他のブランチに統合することが出来ます。
1 つのブランチに複数のブランチの内容をまとめることも出来るので、実際に開発では機能毎にブランチを作成して、 個々が完成してから master 等のまとめ役になるブランチに統合すると、よりスムーズに開発することが出来ます。

このブランチの運用方法に関しては、後ほどの GitHub 基礎 - GitHub Flow とはで詳しく解説します。

2.3.2 GitHub

GitHub とは

① 概要

- Git ホスティングサイト

GitHub は、Git を使った共有ウェブサービスの事で、主にソフトウェア開発を目的として使用されています。
簡潔に述べるならば、Git 基礎で解説したリモート(リモートリポジトリを置いておく所)です。

- 開発者に嬉しいサポート機能

また、単純にリモートの役割を果たすだけでなく、開発者同士のコミュニケーションをとれる SNS 機能や、　タスク管理にコードレビューの補助が出来るようなものまで、開発を効率的に進めるための機能が備わっています。
これらは非常に強力で、サポート機能を活かした GitHub Flow というワークフローも考案されています。
このワークフローは今回のセミナーでも活用していきますので、後ほど詳しく解説します。

② 代表的なサポート機能

- Issues

Issues は、「issue」というものを作成してリポジトリごとに管理する機能です。
この issue は直訳すると「問題」になるのですが、ここでは「課題」という意味で捉えると良いと思います。
issue は、そのリポジトリの中で議論したいことや対応する課題を書き出します。
タグを付けたり、担当者を割り振ったりすることも出来るので、運用によって様々な働きを見せます。
作成した issue には ID が割り振られていて、その ID で issue へのリンクを貼ることが出来たり、 後述する Pull Requests や Projects でも、issue と紐付いた機能を使用することが出来ます。

- Pull Requests

Pull Requests とは、とても優れたコードレビューの補助機能です。
この機能では Pull Request(以降は PR)というものを作成します。
この PR は２つのブランチを使用して作成されます。
ちなみに、「Pull」とは Git のコマンド git pull を意味していて、　このコマンドを実行すると、他のブランチの変更内容を自分のブランチに取り入れることが出来ます。
その「Request」(要求)をするわけですから、この PR は
「私が変更した内容をレビューして、良ければ取り入れてくださいね」という依頼のような意味になります。

実際に PR を作成・使用する流れは以下のようになります。
ブランチを作成して、その中で作業を進める
リモートに変更を反映させて、GitHub 上で PR を作成する
他の人が PR を基にレビューをする
修正点があれば修正する
OK ならマージ(取り込み)する
PR の優れている点は、何と言ってもレビューのしやすさです。

PR を作成した時に、マージして欲しい相手のブランチと自分が作成したブランチの差分が自動で作成されます。
レビューをする人はその差分を見ることで、とても効率的かつ簡単にレビューをすることが出来ます。

また、差分には 1 行ごとにコメントを付けることが出来るので、わざわざ別の何かで「○○行目の××」と指定しなくても、
そのコメントしたい行で「ここ！」と言えるので、コメントする方も見る方も楽になります。

PR には名前を自由につけることが出来ますが、その先頭に[WIP]という文言がついている場合があります。
この[WIP]とは"Work In Progress"の略で、PR に使用したブランチが、まだ作業中であることを表しています。
これは主に、何か相談事があったり、実装に不安があるので一度フィードバックが欲しい等、「取り入れては欲しくないけど、レビューはして欲しい」時に使用されています。
PR には説明文も付けることが出来るので、困っている内容等を一緒に書き込むことで、相談用のツールとしても活用できるでしょう。

- Projects

この Projects では、「カンバン機能」と呼ばれているものを使用することが出来ます。
ざっくり言うと電子化されたホワイトボードといったところでしょうか。

アジャイルに関する内容で、バックログと呼ばれるものが出てくるのですが、
そこではホワイトボードに付箋を貼って、タスクやストーリーと呼ばれるものを管理しています。
近年ではリモートワーク等の普及によりオンラインでの作業が好まれる傾向にあり、
直に顔を突き合わせてホワイトボードで作業、というのが難しくなってきています。
そこで考えられたのが、この「カンバン機能」、GitHub では「Projects」という名前で実装されている機能です。
ホワイトボード代わりの WEB ページに、カードという付箋代わりのものを貼り付けていき、WEB 上で現実のホワイトボードと同じ作業を出来るようにしようという機能です。

2.3.3 Git の利用

これからの課題・演習では Git と GitHub を利用したワークフローGitHub Flow を使用して進めていきます。 ここでは Git と Github を利用するための事前準備と基本的なコマンドや利用方法について説明します。

Git のインストール

1. Git がインストールされているか確認する

コマンドプロンプトもしくはターミナルで以下のコマンドを入力します。

```
git –version
```

2. Git をインストールする

1 を実行してバージョンが表示されない場合（バージョン 1.8 未満も含む）は、以下の URL からダウンロード後、Git をインストールします。 ※ バージョンが表示されている場合はこの作業は不要です。

- git

```
https://git-scm.com/downloads
```

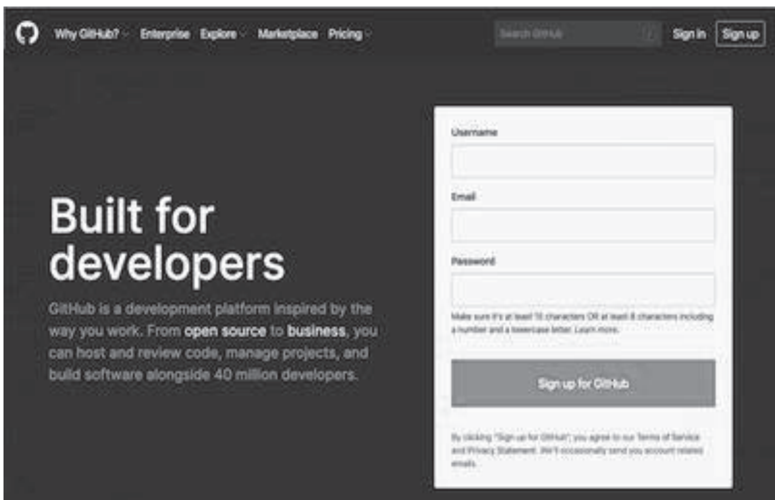
自分のアプリを、コマンドラインで GitHub に Push する

1. Github のアカウントの作成と Git の初期設定

「GitHub」の公式サイト(https://github.com/)にアクセスしてアカウントを作成します。既にアカウントがある場合はログインしてください。

- GitHub 公式サイト

```
https://github.com/
```



2. Git を利用するための初期設定をする

- コマンドプロンプトもしくはターミナルで以下のコマンドを入力します。

```
git config --global user.name "your-name"
git config --global user.email "your-email"
```

※ “your-name” , ”your-email” の部分は GitHub で登録した Username と Email を入れてください。

- コマンドプロンプトもしくはターミナルで以下のコマンドを入力し、name と email の設定が反映されていることを確認してみましょう。

```
git config –list
```

3. Git の利用と基本のコマンド操作

- リポジトリの作成

まずは Git を利用するために、リポジトリを作成しましょう。
今回は、GitHub 上でリポジトリを作成しておいて、それをローカルにクローン（複製）する形式で行います。
GitHub 上にリポジトリを作成する手順は以下の通りです。

① 右上の＋メニュー等から、「New repository」を選択する
② 「Repository name」の項目に、“fizzbuzz”と入力して「Create repository」ボタンを押す
③ リポジトリのトップページに遷移する

作成自体はこれで完了です。
次に、実際に作業をする場所を作成していきましょう。

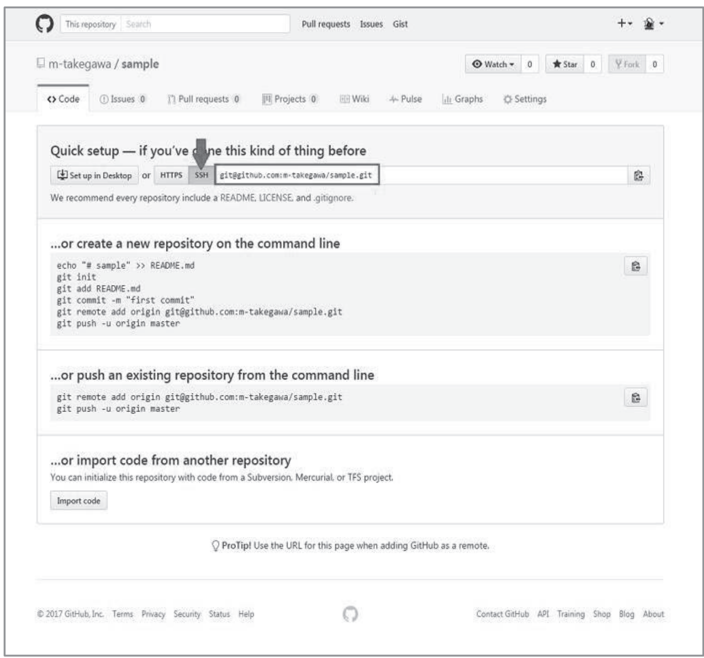
● ワークスペースの作成

今回使用する環境は、Cloud9 を利用して作成します。
Cloud9 には GitHub との連携を補助する機能がありますので、それも合わせて利用していきます。

【ssh-key の設定】
以下の手順で登録をしておくことで、簡単に ssh-key の利用をすることが出来ます。

1. Cloud9 側の「Account Settings」メニュー内で「SSH Keys」を選択する
2. "ssh-rsa"から始まる枠内の文字列をコピーする
3. GitHub 側の「Settings」メニュー内で「SSH and GPG keys」を選択する
4. 「New SSH key」ボタンを押して、「Title」に"Cloud9"（目的や用途が分かる名前）「Key」にコピーした文字列をそのまま貼り付ける
5. 「Add SSH key」ボタンを押して ssh-key を登録する

【Git clone】
Cloud9 では、ワークスペースの作成時にリポジトリのクローンを同時に行うことが出来ます。
作成時に「Clone from Git or Mercurial URL」の項目に、クローンしたいリポジトリの URL を入力することで、
Git コマンド等を使用しなくても、作成と同時にクローンしてくれます。
リポジトリの URL は、作成したリポジトリのトップページから確認することが出来ます。



今回は、先ほど作成した fizzbuzz リポジトリの URL を使用して、ワークスペースを新規作成しましょう。

- master ブランチをリモートに作成し、コミットする。

ワークスペースが作成出来たら、master ブランチをリモートにも作成するために first commit をしましょう。

① git commit コマンドを使ってファイルの変更内容をコミットする

git を使用すれば簡単にファイルの進捗を管理することができます。
進捗を管理するためにはファイルの変更や追加などを保存しておく必要があります。そのためのコマンドが git commit です。 git commit を利用すればファイルの変更を git レポジトリに保存することができます。

それでは実際に git commit を使用してみましょう。
以下のコマンドを実行し、readme.md ファイルを新規作成してください。

```
$ echo "# fizzbuzz" >> README.md
```

git add <ファイル名> のコマンドを実行すると、ステージングエリア（インデックス）に一時保存することができます。

```
$ git add readme.md
```

そして git status コマンドで確認してみると、以下のようにステージングエリアにファイルが移動されたことがわかります。

```
$ git status
```

ファイルの変更を git レポジトリに保存するために変更内容をコミットします。
git commit で変更の履歴を付けます。

```
$ git commit -m "first commit"
```

コミットの詳細は git show コマンドで確認することができます。

```
$ git show
```

```
commit 4d11ceb234e846682c72b330e49b8475b7e94e0b (HEAD -> master)
Author: ████████ <██████████@███████>
Date:   Fri Jan 10 21:07:14 2020 +0900

    first commit

diff --git a/readme.md b/readme.md
new file mode 100644
index 0000000..e69de29
```

リポジトリの変更履歴は git log コマンドで確認することができます。

```
$ git log
```

```
commit 4d11ceb234e846682c72b330e49b8475b7e94e0b (HEAD -> master)
Author: ████████ <██████████@g███████>
Date:   Fri Jan 10 21:07:14 2020 +0900

    first commit
```

log は今までしたコミットをコミットメッセージの一覧という形で表示します。
そのため git log をした際に一目で何をしたのかわかるようなコミットメッセージを書く必要があります。

⑧ ローカルリポジトリの変更をリモートリポジトリに反映する

ローカルリポジトリで開発したとしても、それをリモートリポジトリに反映できなければ、他の人が書いているコードは過去のバージョンのコードのままになっています。

開発コードの状態の矛盾がおきないように、git では git push というリモートリポジトリにローカルリポジトリの変更を反映するためのコマンドが用意されています。
ではローカルの master ブランチで、コミットした変更内容をリモートリポジトリに反映しましょう。 以下のコマンドを実行してください。

```
$ git push origin master
```

上記のコマンドは、origin というリモートリポジトリの master というブランチにプッシュするという意味になります。
基本的に開発はこの master ブランチを基にして、作業内容毎にブランチを切って進めていくことになります。

- git commit コマンドオプション

git commit コマンドは、開発を進めていく際に最も多く使うコマンドのひとつです。
そのため、いくつかのオプションを覚えておくことで、効率よく開発を進めていくことができます。

① 変更したファイルを全てコミットする

既に git に追加済のファイルで、変更があったものを全てコミットするには以下のように入力します。

```
$ git add –a
$ git commit
```

上記と同じことを、「.」を使って１回で行うことができます。

```
$ git commit .
```

② 新規ファイルも含め、全てコミットする

新規に作成した（git に追加していない）ファイルも含め、コミットするには以下のように入力します。

```
$ git add -A
$ git commit
```

先程と同様にこちらも「.」を使って行うことができます。

```
$ git add .
```

```
$ git commit
```

git add -a コマンドと git add -A コマンドの挙動が紛らわしいので、新規ファイルを追加する場合は git add . で行う方法で覚えておくとよいです。

③　コミットメッセージを指定してコミットする

「-m」オプションを指定すると、コミット時にコミットメッセージを指定することができます。

```
$ git commit -m "コミットメッセージ"
```

④　間違ったコミットを取り消す

間違ってコミットしてしまった場合は git reset コマンドを使用します。
コミットを取り消す処理は、本当に取り消してよいか、十分注意して行いましょう。

- コミットだけを取り消す（ローカルのワークディレクトリの内容は変更しない）

```
$ git reset --soft
```

- ローカルのワークディレクトリの内容も含め、コミットを取り消す

```
$ git reset --hard
```

- 直前の間違ったコミットを取り消す（「HEAD^」は、直前のコミットという意味を表すオプションです）

```
$ git reset --soft HEAD^
```

2.4 Git 基礎 2

2.4.1 ワーキングディレクトリと作業ブランチについて

作業ブランチの作成

```
$ git branch call_fizzbuzz
$ git checkout call_fizzbuzz
```

上記のコマンドは、git branch コマンドでブランチを作成し git checkout コマンドで作業ブランチを切り替えています
また、以下のように１つのコマンドにまとめることも出来ます。

```
$ git checkout -b call_fizzbuzz
```

- ブランチの一覧表示

git branch コマンドの引数を指定せずに実行すると、ブランチの一覧を表示することができます。現在のブランチは頭に * がついています。

```
$ git branch
  * call_fizzbuzz
  Master
```

- 問題：作業ブランチに変更内容をコミットする

それでは、以前作成した FizzBuzz プログラムの内容を変更内容として追加しましょう。
前のコマを参考にしながら、コミットしてみてください。

2.4.2 作業のやりなおしについて

開発作業中に何かをやり直したくなる場面はよくあります。ここでは、行った変更を取り消すための基本的なコマンドについて説明します。

- ローカルリポジトリの作業ブランチへの変更を取り消す

```
$ git checkout .
```

git checkout <ファイル名>とすると変更の取り消しを行いたいファイルのみ取り消す事ができます。
git checkout .の場合は、作業ブランチで変更した内容をすべて取り消してはくれますが、新規追加したファイルは削除されません。
完全にもとに戻したい場合は別途削除する必要があります。

```
$ git clean -df .
```

add の取り消し

```
$ git reset HEAD .
```

commit の取り消し

```
$ git reset [<mode>] [<commit>] .
```

mode のデフォルトは--mixed なので、変更したファイルはすべて変わらず、git の履歴のみ変更されます。
変更したファイルも含めてすべて指定したの commit の状態に戻したいので場合には mode に--hard を指定して下さい。 新規追加したファイルは--hard を指定したとしても削除されず、残ったままになるため 完全に元に戻したいのであれば別途削除する必要があります。
リモートリポジトリに push した内容を取り消す

```
$ git revert [<commit>]
$ git push
```

2.4.3 GitHub Flow とは

概要

GitHub Flow は、Git と GitHub を利用したワークフローで、とてもシンプルな構造であるにも関わらず、生産性を大きく向上させる効果を持っています。

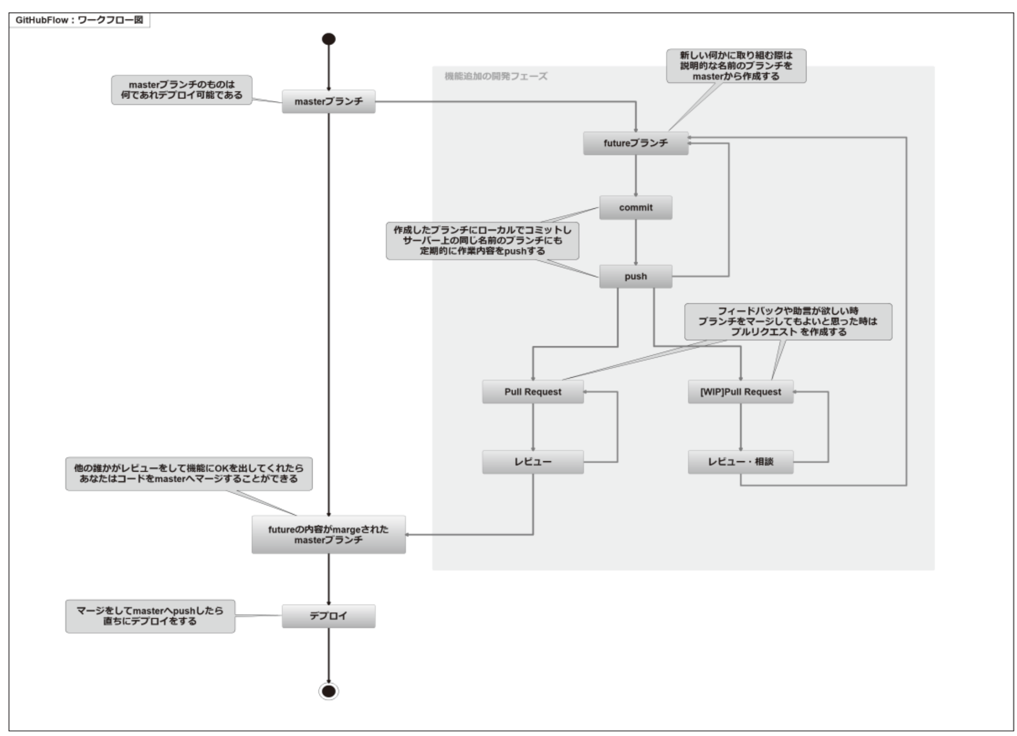
コマンドライン上の操作だけではなく、GUI を用いた仕組みを取り入れることで、Git に慣れていない人でも比較的ハードルの低いワークフローになっています。

基本原則

- master ブランチのものは何であれデプロイ可能である
- 新しい何かに取り組む際は、説明的な名前のブランチを master から作成する（例: new-oauth2-scopes）
- 作成したブランチにローカルでコミットし、サーバー上の同じ名前のブランチにも定期的に作業内容を push する
- フィードバックや助言が欲しい時、ブランチをマージしてもよいと思ったときは、 プルリクエスト を作成する
- 他の誰かがレビューをして機能に OK を出してくれたら、あなたはコードを master へマージすることができる
- マージをして master へ push したら、直ちにデプロイをする

実際のワークフロー

最初にもある通り、このワークフローはとてもシンプルな構造になっています。
具体的には、基本原則を一通りなぞっていけば、それがこのワークフローの全てだと言える程です。
以下がワークフローの詳細です。



この教材での開発はこのワークフローを使用して進めていくことになります。

第３章　テスト駆動型開発

第３章　テスト駆動型開発

3.1 Ruby: 自動テスト

ここでは、意図通りに動作するプログラムを素早く正確に作るために、プログラムのテストを自動化する方法を学びます。

プログラムのコンポーネント単位（特定のメソッドなど）をテストの対象にする「単体テスト」（ユニットテスト）を扱います。

3.1.1 なぜテストを自動化するのか

すこし前の授業で、与えられた数値について以下の動作を行うプログラムを作りました。

- 3 で割り切れる場合には、`Fizz` を出力する
- 5 で割り切れる場合には、`Buzz` を出力する
- 3 と 5 の両方で割り切れる場合には、`FizzBuzz` を出力する
- どれでもない場合には与えられた数値を出力する

この FizzBuzz プログラムを作るとき、どのように作業を進めたでしょうか？　おそらく、以下のようなステップを繰り返したと思います。

1. プログラムを書く
2. 動作を確認する
3. 不具合があればプログラムを直す
4. もう一度動作を確認する

FizzBuzz プログラムはとても単純ですが、それでも出力が正しいことを目視で確認するのは面倒だったのではないでしょうか。

このような手動のテストに時間を使うのは非効率ですし、気をつけていても確認漏れが起こることもあります。なにより、人間が繰り返すにはあまりにも退屈な作業です。

決められた作業を正確に行うのは人間よりもプログラムのほうが得意です。そこで、プログラムをテストするためのプログラムを作って、テストを自動化するというプラクティスが広く行われています。

3.1.2 テストケースを考える

それでは FizzBuzz プログラムのロジックを確認するためのテストケースを用意しましょう。「どんな入力を与えたとき、どんな出力を期待するか」をテストケースと呼びます。
もっとも簡単なテストケースは以下のようになります。

入力	期待する出力
1	1
3	Fizz
5	Buzz
15	FizzBuzz

3.1.3 テストしやすい設計

外部から見ると同じ動作をするプログラムでも、自動テストを書きやすいものと書きにくいものがあります。
まず、次のコードを読んでみてください。

```
# main.rb
1.upto(100) do |n|
  words = ''
  words << 'Fizz' if n % 3 == 0
  words << 'Buzz' if n % 5 == 0
  puts words.empty? ? n : words
end
```

10 行足らずのごく短いコードですが、自動テストを書くのは簡単ではありません。
繰り返し処理の中にすべての処理が直接書かれていて、動作をテストするにも 1 から 100 までの全出力を扱わなければいけないからです。

一方、次のコードでは、数字を受け取って出力内容を返す処理が `fizzbuzz` メソッドとして切り出されています。

```
# fizzbuzz.rb
def fizzbuzz(n)
  words = ''
  words << 'Fizz' if n % 3 == 0
  words << 'Buzz' if n % 5 == 0
  words.empty? ? n : words
end

# main.rb
require_relative './fizzbuzz'
```

```
1.upto(100) do |n|
 puts fizzbuzz(n)
end
```

このように処理をメソッドとして切り出すことで、テストケースの入力と出力をメソッドの引数と返り値と対応づけることができます。

3.1.4 テストを書いてみる

それではテストを書いてみましょう。

```
require_relative './fizzbuzz'

def check(input, expected)
  actual = fizzbuzz(input)
  unless actual == expected
    abort "テストに失敗しました。\n 入力値: #{input}, 期待する出力: #{expected}, 実際の出力: #{actual}"
  end
end

check(1, 1)
check(3, 'Fizz')
check(5, 'Buzz')
check(15, 'FizzBuzz')

puts 'テストに成功しました'
```

`fizzbuzz` メソッドから返ってきた実際の出力と期待する出力を比較して、違いがあれば報告されるようになっています。

3.1.5 仕様書としてのテストコード

以下のようにコメントをつけることで、テストコード上でプログラムの仕様を説明することができます。

```
# 普通の数字を入力するとそのまま返される
check(1, 1)

# 3 の倍数を入力すると Fizz が返される
check(3, 'Fizz')
```

```
# 5の倍数を入力するとBuzzが返される
check(5, 'Buzz')

# 15の倍数を入力するとFizzBuzzが返される
check(15, 'FizzBuzz')
```

このようにテストコードとペアになった仕様書には次のようなメリットがあります。

- テストコードとして実行できるため、正しさを検証しやすい
- テストコードは開発にも使用するため、更新忘れが起きにくい

3.1.6 すべてをテストすることはできない

ところで、今回使ったテストケースを通過したとしても、コードが正しく動作すると保証することはできません。

入力	期待する出力
1	1
3	Fizz
5	Buzz
15	FizzBuzz

たとえば、仕様を誤解して以下のようなプログラムを作ってしまったらどうなるでしょうか？

```
def fizzbuzz(n)
  case n
  when 0...3
    n
  when 3...5
    'Fizz'
  when 5...15
    'Buzz'
  when 15...Float::INFINITY
    'FizzBuzz'
  else
    n
  end
end

<!--
# 動作確認用コード
[1, 3, 5, 15].each do |n|
  p fizzbuzz(n)
```

```
    end
    -->
```

このようにプログラムに誤りがあっても、たまたま期待通りの出力になってしまい、コードの誤りを見つけられないこともあります。

今回の fizzbuzz メソッドはすべての数値を引数として受け付けるため、有限のテストケースでは動作を完全に検証することはできません。むやみにテストケースを増やすとテストの実行時間が長くなり、メンテナンスの手間もかさみますから、効率のよいテストケースを考える必要があります。

ここでは詳細を説明することはできませんが、少ないテストケースでできるだけ効率よくプログラムの動作を検証するための技法は古くから研究されており、それを主題にした書籍も多数出版されています。また、コードが仕様通りに動作することを数学的に証明する形式手法などのアプローチも実用化されています。関心があれば「テスト設計」「形式手法」などをキーワードにして探してみてください。

3.1.7 自動テストとアジャイル開発

今回学んだ自動テストは、アジャイル開発のいくつかのプラクティスと深い関わりを持ちます。

a) テスト駆動開発

テスト駆動開発は、自動テストを開発の起点にすることで変更コストの低いコードを作ることを目指します。

b) 継続的インテグレーション・継続的デプロイ

成果物を素早くメインライン（trunk ブランチ、master ブランチなどと呼ばれます）にマージして、自動的に出荷可能な状態にするプラクティスです。

成果物がマージ可能であること、出荷可能であることを保証するための手段として自動テストが利用されます。

3.1.8 まとめ

ここではプログラムの自動テストについて学びました。ポイントをおさらいしておきましょう。

- プログラムをテストするプログラムを作ることで、テストを自動化できる

- 自動化されたテストは、意図通りに動作するプログラムを素早く正確に作ることに役立つ
- 同じ動作をするプログラムでも、内部設計によって自動テストの作りやすさに違いがある
- ロジックをメソッドとして切り出すことで自動テストを作りやすいプログラムにすることができる
- 自動テストだけでプログラムの正しさを保証することができるとは限らない
- 自動テストはアジャイルのプラクティスと深い関わりを持っている

3.1.9 発展課題

- FizzBuzz プログラムに「数字が 7 で割り切れる場合には、数字の代わりに"Jazz"を出力する」というルールを追加したとき、テストケースはどのように変化するか考えてみましょう。
- じゃんけんプログラムを作り、それに対する自動テストを作ってみましょう。仕様は以下の通りです。

  1. プレイヤーから G, C, P のどれかの入力を受け取り、プレイヤーの手とする
     入力と手の対応は G: グー, C: チョキ, P: パーとする
  2. プログラムはランダムに自分の手を選び、画面に表示する
  3. プレイヤーとプログラムの手を比較し、どちらが勝ったかを画面に表示する

この授業で作った自動テストの仕組みに不便なところはありますか？　もし不便なところがあれば、その理由を考えて改善してみましょう。

3.2 Ruby: 自動テスト（minitest）

多くのプログラミング言語では、自動テストを記述するためのフレームワーク（テスティングフレームワーク）が用意されています。

ここは、Ruby 本体に組み込まれている minitest というテスティングフレームワークの概要と使い方を解説します。

3.2.1 概要

minitest は Ruby 本体に gem（ライブラリ）として組み込まれています。また、Rails の標準のテスティングフレームワークでもあります。Rails アプリケーションのひな形に最初からインストールされていて、Rails そのものの自動テストにも minitest が使われています。

minitest では、テストケースのまとまりを Ruby のクラスとして表し、個々のテストケースはインスタンスメソッドとして記述します。

構造が比較的シンプルで、Ruby の基本的な知識があれば使うことができます。

このようにテストケースをクラスとメソッドとして表現して、`assert_○○` というメソッドで値をチェックするテスティングフレームワークは xUnit と総称されます。他のプログラミング言語の xUnit 系のフレームワークには、Smalltalk の SUnit、Java の JUnit などがあります。

3.2.2 テストの書き方

`String` クラスのテストを minitest で書くと、以下のようになります。

```
# minitest の読み込み
require 'minitest/autorun'

# Minitest::Test クラスを継承する
class TestString < Minitest::Test

  # "test_"で始まるメソッドがテストメソッドになる
  def test_length
```

```
    # assert_equal で 2 つの値が等しいことをテストする
    assert_equal(3, 'abc'.length)
  end

  # テストメソッドはいくつでも作ることができる
  def test_match

    # assert_instance_of で最初の引数が 2 番目の引数のインスタンスであることをテストする
    assert_instance_of(MatchData, 'abc'.match(/./))
  end
end
```

上記のソースコードを test_string.rb などの名前をつけて保存して、 ruby test_string.rb で実行してみましょう。

以下のように、最後に 2 runs, 2 assertions, 0 failures, 0 errors, 0 skips と表示されればテストは成功です。

```
Run options: --seed 61764

# Running:

..

Finished in 0.001184s, 1689.1889 runs/s, 1689.1889 assertions/s.
2 runs, 2 assertions, 0 failures, 0 errors, 0 skips
```

テストコードを順番に見ていきましょう。

最初に require 'minitest/autorun' で minitest を読み込んでいます。これによって、テストの定義に必要な Minitest::Test などのクラスが読み込まれます。

次に、 Minitest::Test クラスを継承してテスト用クラスを作成します。このクラスに test_length と test_match というメソッドを定義しています。minitest では、このように test_ で始まるメソッド１つ１つがそのままテストケースになります。

test_length の中では assert_equal というメソッドを使って、 length メソッドの実際の返り値と、期待する値が一致するか確認しています。最初の引数が期待する値で、2 番目の引数が実際の値です。

test_match の中では assert_instance_of というメソッドを使っています。これは、2 番目の引数が最初の引数のインスタンスである

（2 番目の引数.instance_of?(最初の引数)）ことを確認するメソッドです。

このように実行結果が期待している値と一致しているか確認することを、minitest では「アサーション」（assertion）と呼びます。

3.2.3 テストの実行結果を読む

テストの実行結果を読んでみましょう。

まず、わざと失敗するテストを作ってみます。先ほどのテストコードのアサーションを以下のように書き換えて、テストを再実行してみてください。

```
  def test_length
    assert_equal(100, 'abc'.length)
  end

  def test_match
    assert_instance_of(Integer, 'abc'.match(/./))
  end
```

実行結果と期待する値が食い違っているため、エラーになります。

```
Run options: --seed 45537

# Running:

FF

Finished in 0.001130s, 1769.9115 runs/s, 1769.9115 assertions/s.

  1) Failure:
TestString#test_length [test_string.rb:7]:
Expected: 100
  Actual: 3

  2) Failure:
TestString#test_match [test_string.rb:11]:
Expected #<MatchData "a"> to be an instance of Integer, not MatchData.

2 runs, 2 assertions, 2 failures, 0 errors, 0 skips
```

後ろからさかのぼって読んでいきましょう。

出力の最後に `2 runs, 2 assertions, 2 failures, 0 errors, 0 skips` とあります。これは「2 件のテストケースを実行して、合計 2 つのアサーションを実行した。2 つのテストケースが失敗して、エラー（例外）が発生したテストケースとスキップされたテストケースは 0 件だった」という意味です。

その前の数行には、それぞれのテストケースでアサーションがどのように失敗したかが表示されています。

その前の Finished in ...の行ではテストの実行速度をレポートしています。アプリケーションの規模によってはテストケースが数百件以上になり、テストが完了するまでに数十秒から数分、あるいはそれ以上を要することもあります。

Running:の後に出てくる FF はテストケースごとの状態を表します。１文字が１テストケースの状態を表していて、成功したら .、失敗したら F となります。アプリケーションが育っていくと . がずらりと並ぶことになります。

最初に表示されている Run options: --seed 45537 はテストを再実行するためのコマンドラインオプションです。--seed は自動的に追加されるオプションで、テストケースの実行順を決めるために使われます。これについては次の節で説明します。

3.2.4 テストケースの実行順

minitest は実行するたびにテストケースの順番をランダムに変更します。

たとえば次のように、あるテストケースが他のテストケースの実行結果を前提にしているテストは、実行するごとに成功したり失敗したりします。

```
require 'minitest/autorun'

$array = []

class TestArray < Minitest::Test
  def test_plus
    $array += [1, 2]
    assert_equal([1, 2], $array)
  end

  def test_minus
    # test_plus で$array = [1, 2]になることを期待している
    $array -= [1]
    assert_equal([2], $array)
  end
end
```

なぜこのような機能がわざわざ実装されているのでしょうか？

これは「テストケースは単体で実行可能であるべき」という原則を守ることを助けるための機能です。

テストケースが他のテストケースに依存していると、あるテストケースを変更したら他のテストケースが意図せず壊れてしまうようになります。そのような状態に陥ってしまうと、テストをメンテナンスするのが難しくなります。それを防ぐため、あえてテストケースの実行順が固定されないようにしているのです。

テストの実行順のシード値はコマンドラインの `--seed` オプションで指定することができます。同じ内容のテストファイルについて同じシード値を使うと、テストケースを同じ順番で実行することができます。

minitest に限らず、多くのテスティングフレームワークは同様の機能を持っています。

3.2.5 さまざまなアサーションメソッド

minitest にはさまざまなアサーションメソッドがあります。チェックしたい値の性質に合ったアサーションメソッドを使うことで、テストの意図が明確になり、失敗したときのエラーメッセージもわかりやすくなります。

・基本的なアサーションメソッド

メソッド名	成功条件
assert(a)	a が真
assert_empty(a)	a が empty?に真を返す
assert_equal(a, b)	a == b が真
assert_includes(a, b)	a.include?(b)が真
assert_nil(a)	a =~ b が真
assert_predicate(a, op)	a.nil?が真
assert_operator(a, op, b)	a.send(op)が真
assert_raises(*e, &block)	block を実行すると e のいずれかの例外を発生させる

```
# array.empty?が真なら成功
assert_predicate(array, :empty?)

# a <= b が真なら成功
assert_operator(a, :<=, b)

# ブロックが StopIteration 例外を発生させれば成功
assert_raises(StopIteration) { raise StopIteration }
```

・誤差を許容するアサーションメソッド

主に計算誤差が発生する浮動小数点数を比較するために使われます。

メソッド名	成功条件
assert_in_delta(a, b, delta)	a - b の差の絶対値が delta 以下
assert_in_epsilon(a, b, epsilon)	a と b の相対的な差が epsilon 以下

・クラスやメソッドに関するアサーションメソッド

メソッド名	成功条件
assert_instance_of(a, b)	a.instance_of?(b)が真
assert_kind_of(a, b)	a.kind_of?(b)が真
assert_respond_to(a, b)	a.respond_to?(b)が真

・その他のアサーションメソッド

メソッド名	成功条件
assert_output(stdout, stderr, &block)	block が標準出力に stdout, 標準エラー出力に stderr にマッチする文字列を出力する
assert_path_exists(path)	File.exist?(path)が真
assert_same(a, b)	a.equal?(b)が真
assert_send(receiver, method, *args)	receiver.send(method, *args)が真
assert_silent(&block)	block が標準出力と標準エラー出力に何も表示しない
assert_throws(tag, &block)	block が tag を throw する

いくつかのアサーションメソッドには、成功条件を反転させた refute_〇〇というアサーションメソッドがあります。たとえば、refute_equal(a, b)は assert_equal(a, b)とは逆に、a != b のときだけアサーションが成功します。

また、アサーションメソッドの最後の引数に文字列を渡すと、デフォルトのエラーメッセージの代わりにその文字列が使われます。

3.2.6 テストケースに影響を及ぼす機能

最後に、テストケースに影響を及ぼす機能をいくつか紹介します。

`setup` メソッドと `teardown` メソッドを定義すると、それぞれテストケースの実行前、実行後に呼び出されます。テストケースで使うデータを用意する処理をまとめるのに便利です。

テストケース内で `skip` メソッドを呼ぶと、そのテストケースをスキップすることができます。一時的にテストの実行を止めたいときに使うと便利です。

```
class TestSample < Minitest::Test
  # テストケースの実行前に呼ばれる
  def setup
    @user = User.create(name: '佐藤太郎')
  end

  # テストケースの実行後に呼ばれる
  def teardown
    @user.destroy
  end

  # skip メソッドを呼ぶとテストをスキップできる
  def test_foo
    skip 'このテストは xx の実装後に書く'

    # skip メソッドより後の処理は実行されない
    assert(false)
  end
end
```

minitest には他にも多くの機能があり、プラグインで拡張することもできます。関心のある人は<https://github.com/seattlerb/minitest>で公式のドキュメントを参照してみましょう。

3.2.7 まとめ

ここでは minitest による自動テストの作り方について学びました。ポイントをおさらいしておきましょう。

- minitest ではテストケースをメソッドとして表す

- minitest のテストケースでは assert_〇〇 というメソッドを使って実行結果が期待通りか確認する

3.2.8 発展課題

- fizzbuzz メソッドの自動テストを書いてみましょう。
- Ruby の標準クラスをひとつ選んで、いくつかのメソッドの自動テストを書いてみましょう。
- minitest のアサーションメソッドの実装を読んでみましょう。
- minitest の実行結果はカスタムレポーターを定義することで自由にカスタマイズできます。カスタムレポーターとその自動テストを書いてみましょう。

minitest のソースコードは GitHub で公開されています
（https://github.com/seattlerb/minitest）

3.3 Ruby: 自動テスト（RSpec）

RSpec は、前回取りあげた minitest と並んで人気のあるテスティングフレームワークです。ここでは RSpec の概要と使い方を紹介し、minitest との違いについて学びます。

3.3.1 概要

RSpec の「R」は Ruby の頭文字です。「Spec」は Specification、つまり「プログラムの仕様」を意味しています。RSpec は単なる自動テストではなく、プログラムの振る舞いをソースコードで表現することを重視したテスティングフレームワークです。

RSpec のテストコードは minitest と同じく Ruby そのものですが、テストコードは自然言語（日本語、英語などの日常的に使う言語）に近づけられています。そのため、テストケースの定義や実行結果の検証の構文は minitest とは異なります。

たとえば、「Array クラスのインスタンスを引数なしで作った場合、そのインスタンスは empty?メソッドで true を返す」というテストコードをそれぞれのフレームワークで書くと、以下のようになります。

```
# minitest
class TestArray < Minitest::Test
  def test_empty
    assert(Array.new.empty?)
  end
end

# RSpec
RSpec.describe Array do
  describe '.new' do
    context '引数がないとき' do
      it '空であること' do
        expect(Array.new).to be_empty
      end
    end
  end
end
```

3.3.2 gem のインストール

minitest と違って、RSpec は Ruby 本体には組み込まれていません。gem install rspec で gem をインストールしてください。

gem をインストールすることで、RSpec のテストを実行するための rspec コマンドもインストールされます。

3.3.3 Array クラスのテストを書く

Array クラスを題材にして RSpec のテストを書いてみましょう。

プロジェクトを作る

まず、 `rspec_array` という名前のディレクトリを作ってください。
次に、 `RSpec` の設定ファイルを用意します。 `rspec_array` ディレクトリで `rspec –init` を実行してください。

- `.rspec`
- `spec/spec_helper.rb`

という 2 つのファイルが作られます。これらの設定ファイルの役割はあとで説明します。

空のテスト

RSpec では、テストコードは `spec` ディレクトリに `***_spec.rb` という名前で配置することになっています。

以下のプログラムを `spec/array_spec.rb` に保存してください。

```
# spec/array_spec.rb

RSpec.describe Array do
end
```

`rspec_array` ディレクトリで `rspec` コマンドを実行してみましょう。以下のように、最後に 0 examples, 0 failures と表示されれば成功です。

```
$ rspec
No examples found.
Finished in 0.0003 seconds (files took 0.14107 seconds to load)
0 examples, 0 failures
```

テストケースを追加する

まずは、 Array.new が空の配列を返すことをテストしてみましょう。テストコードは次のようになります。

```
# spec/array_spec.rb

# RSpec.describe でテストの対象を指定する
RSpec.describe Array do

  # it でテストケースを定義する
  it do

    # expect(...).to ...で実行結果をテストする
    expect(Array.new).to eq []
  end
end
```

最初の RSpec.describe でテストの対象を指定します。そのブロックの中にある it がテストケースです。it の中の expect(Array.new).to eq [] が「 Array.new == [] である」というテストコードになります。
minitest ではクラスとメソッドを使ってテストケースを表現していましたが、RSpec ではこのように入れ子のブロックを使ってテストケースを表現します。
また、minitest では assert_○○ というアサーションで処理結果を検証していました。それに対して、RSpec では expect(...).to ... というエクスペクテーションで処理結果を検証します。

新しいテストコードを入力して rspec コマンドを実行してみましょう。以下のように 、 1 example, 0 failures と表示されていれば成功です。

```
$ rspec

Finished in 0.0055 seconds (files took 0.15194 seconds to load)
```

最初の . は minitest と同じようにテストケースの結果を表しています。次の Finished in の行はテストの実行時間を示しています。最後の 1 example, 0 failures は、「1 つのテストケースを実行して、失敗は 0 件だった」という意味になります。

3.3.4 テストケースを失敗させる

次はテストケースを失敗させてみましょう。テストコードを以下のように書き換えてください。

```
# spec/array_spec.rb
```

```
RSpec.describe Array do
  it do
    expect(Array.new).to eq [nil]
  end
end
```

これで「Array.new == [nil] である」というテストになります。
rspec コマンドを実行すると以下のように出力されます。

```
$ rspec
F

Failures:

  1) Array is expected to eq [nil]
     Failure/Error: expect(Array.new).to eq [nil]

       expected: [nil]
            got: []

       (compared using ==)

       Diff:
       @@ -1,2 +1,2 @@
       -[nil]
       +[]

     # ./spec/array_spec.rb:3:in block (2 levels) in <top (required)>'

Finished in 0.01932 seconds (files took 0.12781 seconds to load)
1 example, 1 failure

Failed examples:

rspec ./spec/array_spec.rb:2 # Array is expected to eq [nil]
```

出力を上から順番に見ていきましょう。
最初の F はテストケースが 1 つ失敗したことを表しています。
次に失敗したテストケースの詳細が表示されています。

- どのようなエクスペクテーションで失敗したか
- どんな値を期待していて、実際の値はどうだったのか
- 失敗したエクスペクテーションはどのファイルの何行目か

という情報が読み取れます。
その次の Finished in は成功時の出力と同じです。 1 example, 1 failures は、「1 つのテストケースを実行して、失敗は 1 件だった」という意味になります。
最後に、失敗したテストケースが列挙されています。

3.3.5 テストケースを整理する

RSpec の出力では、テストケースを example と呼んでいることに気づいたでしょうか？ RSpec のテストケースは単なるテストではなく、サンプルコード（実行例）のように書くことが勧められています。

RSpec はテストケースを実行例として整理するために、このようにテストケースに名前をつけて階層化する機能を備えています。

まず、テストケースに名前をつけてみましょう。次のように it の最初に引数に文字列を渡すと名前をつけることができます。

```
RSpec.describe Array do
  it '引数なしで Array.new を実行すると空の配列を返す' do
    expect(Array.new).to eq []
  end

  it '整数を引数にして Array.new を実行するとその長さの配列を返す' do
    expect(Array.new(3).size).to eq 3
  end
end
```

さらに、次のように describe メソッドと context メソッドを入れ子にすることができます。

```
RSpec.describe Array do

  # describe メソッドで「何についてのテストなのか」を示す
  describe '#new' do

    # context メソッドで「どんな場合のテストなのか」を示す
    context '引数がない場合' do
      it '空の配列を返す' do
        expect([]).to eq []
      end
    end

    context '整数を引数に渡した場合' do
      it '整数と同じ長さの配列を返す' do
        expect(Array.new(3).size).to eq 3
      end
    end
  end
end
```

--format documentation オプションをつけて rspec コマンドを実行すると、次のようにテストの実行結果がドキュメント形式で表示されます。

```
$ rspec --format documentation

Array
  #new
    引数がない場合
      空の配列を返す
    整数を引数に渡した場合
      整数と同じ長さの配列を返す

Finished in 0.00261 seconds (files took 0.11846 seconds to load)
2 examples, 0 failures
```

ここまでに describe, context, it, expect という４つのメソッドを紹介しました。それぞれの役割は以下の通りです。

メソッド名	含むことができるメソッド	役割
describe	describe, context, it	何についてのテストなのか」を示す
context	describe, context, it	「どんな場合のテストなのか」を示す
context	describe, context, it	どんな場合のテストなのか」を示す
it	expect	１つのテストケースを表す
expect	なし	期待する実行結果を表す

3.3.6 さまざまなマッチャー

minitest には、検証したい値の種類に応じていろいろな種類のアサーションメソッドがありました。RSpec では同じものを「match するかどうか確かめるもの」という意味でマッチャーと呼んでいます。

マッチャーは expect(actual).to への引数を返すメソッドです。サンプルコードでは eq マッチャーだけを使っていましたが、RSpec のマッチャーは minitest よりも数が多く、機能も複雑です。

ここでは代表的なものだけを紹介します。

eq

```
expect(actual).to eq expected
```

actual == expected であることを確かめます。もっとも基本的なマッチャーです。

include

```
expect(actual).to include expected
```

actual.include?(expected) が真であることを確かめます。

be_truthy, be_falsey

```
expect(actual).to be_truthy
expect(actual).to be_falsey
```

be_truthy は actual が真値であることを確かめます。be_falsey は偽値であることを確かめます。

述語マッチャー

```
expect(actual).to be_positive
expect(actual).to be_empty
```

actual に ○○? というメソッドが定義されているとき、be_○○という形式のマッチャーが使えます。
actual.○○?が真であることを確かめます。

raise_error

```
expect { ... }.to raise_error(exception_class)
```

expect に渡したブロックが exception_class の例外を起こすことを確かめます。expect の引数がブロックになっていることに注意してください。

マッチャーの否定

```
expect(actual).not_to be_empty
```

to のかわりに not_to を使うことで、マッチャーの意味を逆にすることができます。
この例では actual.empty? が偽であることを確かめています。

3.3.7 テストケースに影響を及ぼす機能

before メソッドにブロックを渡すと、テストケースの実行前にブロックが実行されます。

```
RSpec.describe Array do
  context '配列が空' do
    before { @array = [] }
    it { expect(@array).to be_empty }
    it { expect(@array.size).to be_zero }
  end

  context '配列が空ではない' do
    before { @array = [1, 2, 3] }
    it { expect(@array).not_to be_empty }
    it { expect(@array.size).to be_positive }
  end
end
```

after メソッドにブロックを渡すと、テストケースの実行後にブロックが実行されます。

以下のように、 it を skip に置き換えたり、 it のブロックの中で skip メソッドを呼ぶことでテストケースをスキップすることができます。

```
RSpec.describe Integer do
  skip { expect(3.fizzbuzz).to eq 'Buzz' }

  it do
    skip 'このテストは fizzbuzz の実装後に書く'
    expect(5.fizzbuzz).to eq 'Fizz'
  end
end
```

minitest と RSpec の違い

RSpec のような記法のテスティングフレームワークは、プログラムの振る舞い（Behavior）を重視するという意味で、BDD (Behavior Driven Development) スタイルと呼ばれています。

minitest は BDD スタイルもサポートしているので、以下のようなテストコードを書くこともできます。

```
require 'minitest/autorun'

describe Array do
  describe '配列が空' do
    before { @array = [] }
    it { _(@array).must_be_empty }
    it { _(@array.size).must_be :zero? }
  end

  describe '配列が空ではない' do
    before { @array = [1, 2, 3] }
    it { _(@array).wont_be_empty }
    it { _(@array.size).must_be :positive? }
  end
end
```

RSpec と minitest の大きな違いは内部の構造にあります。RSpec では、テストケースもマッチャーもすべてがオブジェクトです。

そのため、テストケースにラベルをつけて特別な処理をしたり、複数のマッチャーを合成するなど、minitest にはない拡張性があります。その反面、minitest に比べると、ライブラリとしての構造が複雑で、何か意図しないことが起きたときに内部の処理を追うのに手間がかかります。

次の章からは RSpec を使ってテストコードを書いていきますが、みなさんが自分でプロジェクトを始めるときには、チーム全体の考え方などに合わせて適切なテスティングフレームワークを選択してください。

3.3.8 まとめ

ここでは RSpec による自動テストの作り方について学びました。ポイントをおさらいしておきましょう。

- RSpec はプログラムの振る舞いを記述することを重視したテスティングフレームワークである
- RSpec では `it` メソッドを使ってテストケースを定義する
- RSpec のテストケースでは `expect` メソッドで実行結果が期待通りか確認する
- RSpec では `describe`，`context` を使ってテストケースを階層化して整理する

3.3.9 発展課題

- minitest で書いた自動テストを RSpec を使うように書き換えてみましょう。
- RSpec の書き方のガイドラインを読んでみましょう。（http://www.betterspecs.org/jp/）
- 入れ子になった `describe` や `context` のそれぞれで `before` メソッドを呼び出してみましょう。 `before` メソッドのブロックはどのような順番で実行されるでしょうか？
- RSpec のドキュメントを読んで、このテキストで説明されていないマッチャーを使ってみましょう。（https://relishapp.com/rspec/rspec-expectations/docs/built-in-matchers）
- RSpec のマッチャーを自分で定義してみましょう。

RSpec のソースコードは GitHub で公開されています。minitest と違い、３つの gem に分割されています。

- （https://github.com/rspec/rspec-core）：テストケースの実行機能などを提供します
- （https://github.com/rspec/rspec-expectations）：エクスペクテーションを提供します
- （https://github.com/rspec/rspec-mocks）：テスト用のダミーオブジェクトを作るための機能を提供します

3.4 テスト駆動開発

ここではテスト駆動開発（Test Driven Development: TDD）について学びます。テスト駆動開発はアジャイル開発のプラクティスのひとつで、特にエクストリーム・プログラミング（XP）で重視されています。

3.4.1 概要

テスト駆動開発は「動作するきれいなコード」を作ることを目的とする開発手法です。
「動作する」と「きれいな」は次のような意味です。

- 動作する: プログラムが意図したような動きをすること
- きれいなコード: プログラムが読みやすく、機能拡張しやすいこと

テストコードは単なる動作確認の道具ではなく、開発を進めるためのガイドとして使います。
開発のステップは次のようになります。

1. プログラムにどのような機能を追加するか決める
2. （レッド）その機能のテストコードを書く
3. （グリーン）テストコードが成功するようにプログラムを変更する
4. （リファクタリング）テストコードを追加する状態を維持しながらプログラムを改善する
5. 1 に戻る

これまでの授業ではプログラムを書いたあとでテストコードを作っていましたが、テスト駆動開発で最初に書くのはテストコードです。これを「テストファースト」といいます。

ステップ 2 ではテストを書いただけでプログラム本体には変更を加えていないので、テストは失敗するはずです。この状態を「レッド」と呼びます。

ステップ 3 でテストコードが成功した状態を「グリーン」といいます。ステップ 3 はテストコードを通せばよく、本当の意味で正しいプログラムを書く必要はありません。

最後のステップ 4 で、プログラムを修正していきます。この作業をリファクタリングと呼びます。
本当にこのステップが「動作するきれいなコード」が作るのに役立つのでしょうか？
もし役立つとしたら、それはなぜでしょうか？　実例をもとに見ていきましょう。

3.4.2 実例

与えられた年がうるう年か判定するプログラムを書いてみることにします。

判定ルールは以下の通りです。

- 通常の年は平年（うるう年ではない年）とする
- 4 で割り切れる年はうるう年とする
- ただし、100 で割り切れる年は平年とする
- ただし、400 で割り切れる年はうるう年とする

手順 1 の「プログラムにどのような機能を追加するか決める」から進めていきます。

最初のテスト

上のルールのどこから手をつけていけばいいでしょうか？　素直に最初のルールから作っていきたいところですが、「通常の年」というのはあまりはっきりしない概念です。わかりやすい「4 で割り切れる年はうるう年とする」というルールから手をつけていきましょう。
まずは RSpec でテストを書きます。うるう年は英語で leap year といいますから、`leap_spec.rb` というファイル名にします。

```
# leap_spec.rb

RSpec.describe
```

ここまで書いたところで手が止まりました。テストコードを書き進めるには、うるう年の判定処理の呼び出し方を決める必要があります。

簡単に `leap_year?` のようなメソッドをトップレベルに定義してもよさそうですが、ここでは `Date` という標準クラスに特異メソッド `leap?` を定義してみることにします。このメソッドは、引数として受け取った数値がうるう年であれば true を返すことにします。

Date.leap? のテストコードの枠組みは次のようになります。

```
require 'date'

RSpec.describe Date do
  describe '.leap?' do
  end
end
```

Date.leap?に 4 で割り切れる数値を渡すと true が返されるというテストを書いてみましょう。次のようになります。

```
require 'date'

RSpec.describe Date do
  describe '.leap?' do
    context '4 で割り切れる年' do
      it 'うるう年になる' do
        expect(Date.leap?(4)).to be true
      end
    end
  end
end
```

このように、最初にテストコードを書こうとすると、「どのように作るか」の前に「何を作るか」を考えることになります。いきなりプログラムを書き始めると「どのように作るか」と「何を作るか」を同時に考えなければいけません。「どのように作るか」が難しいときには、「何を作るか」がぶれてしまったり、クラスやメソッドの設計に歪みが生まれることもあります。

テストのテスト

では、rspec leap_spec.rb でテストコードを実行してみましょう。Date.leap?メソッドはまだ作っていませんから、NoMethodError が出力されるはずです。

```
$ rspec leap_spec.rb
.
Finished in 0.00278 seconds (files took 0.11796 seconds to load)
1 example, 0 failures
```

予想に反して成功してしまいました。調べてみると、 `Date.leap?` は Ruby 本体ですでに定義されていることがわかりました。

このように、失敗すると思っていたテストが成功してしまったり、予想とは違う理由で失敗したりすることがあります。プログラムを書いてしまってからそのような状況にぶつかると、テストに問題があるのか、プログラムに問題があるのか分からなくなってしまいます。もっと悪い場合には、テストが間違っているのにそれに気づかずに開発を進めてしまうこともあります。テストを書いたら、そのテストが期待通りに失敗することを忘れずに確認しましょう。

テストコードのおかげで `Date.leap?`というメソッドを知ることができましたが、このテキストでは `leap?`メソッドを引き続き作っていくことにします。

以下のようなコードを書いて、`leap_spec.rb` から読み込みます。今度はテストが失敗するはずです。

```
# leap.rb

require 'date'

class Date
  def self.leap?(year)
  end
end

# leap_spec.rb

require_relative './leap'

RSpec.describe Date do
  describe '.leap?' do
    context '4で割り切れる年' do
      it 'うるう年になる' do
        expect(Date.leap?(4)).to be true
      end
    end
  end
end
```

```
$ rspec leap_spec.rb
F

Failures:

  1) Date.leap? 4で割り切れる年 うるう年になる
     Failure/Error: expect(Date.leap?(4)).to be true

       expected true
            got nil
     # ./leap_spec.rb:9:in block (4 levels) in <top (required)>'

Finished in 0.02341 seconds (files took 0.11577 seconds to load)
1 example, 1 failure

Failed examples:

rspec ./leap_spec.rb:8 # Date.leap? 4で割り切れる年 うるう年になる
```

うまく失敗しました。今は「レッド」の状態です。これで「その機能のテストコードを書く」というステップは完了です。

最小のステップ

次のステップは「テストコードが通過するようにプログラムを書き換える」です。次のようにしてみましょう。

```
# leap.rb

require 'date'

class Date
  def self.leap?(year)
    true
  end
end
```

テストが成功したので、「グリーン」の状態にすることができました。予想通りの返り値を返すときちんとテストが成功することが確かめられたので、ステップは完了です。

しかし、これは明らかに間違ったプログラムです。year をまったく使っていません。次の「テストコードを追加する状態を維持しながらプログラムを改善する」のリファクタリングで修正していきましょう。

間違いを直す

「4 で割り切れる」が条件なので、year を 4 で割った余りを確認することにします。

```
# leap.rb

require 'date'

class Date
    def self.leap?(year)
       year % 4 == 0
  end
end
```

テストは成功です。
これで「4 で割り切れる年はうるう年とする」というルールは満たせました。

- 通常の年は平年（うるう年ではない年）とする
- ~~4 で割り切れる年はうるう年とする~~
- ただし、100 で割り切れる年は平年とする
- ただし、400 で割り切れる年はうるう年とする

先ほどの「グリーン」にするためだけの一時的な修正を「仮実装」と呼びます。今回の例ではプログラム本体もテストコードもごく簡単なものだったので、ただの遠回りのように見えますが、複雑な問題では便利な足がかりになります。崖にピッケルを打ち込んで足場を作るように、仮実装を使ってテストを成功させてから、徐々にプログラムを正しい形に近づけていくことができます。

第 2 のテスト

では、「プログラムにどのような機能を追加するか決める」という最初のステップに戻りましょう。

ルールをあらためて読んでみると、100 も 400 も 4 で割り切れるので、「通常の年」は「4 で割り切れない年」と読み替えられることがわかります。「通常の年は平年（うるう年ではない年）とする」というルールを「4 で割り切れない年は平年とする」に置き換えて、手をつけてみることにします。

次のステップは「その機能のテストコードを書く」でした。

```
# leap_spec.rb
```

```
require_relative './leap'

RSpec.describe Date do
  describe '.leap?' do
    context '4で割り切れる年' do
      it 'うるう年になる' do
        expect(Date.leap?(4)).to be true
      end
    end

    context '4で割り切れない年' do
      it '平年になる' do
        expect(Date.leap?(5)).to be false
      end
    end
  end
end
```

テストを実行してみると成功します。

```
$ rspec leap_spec.rb

Finished in 0.00339 seconds (files took 0.12956 seconds to load)
2 examples, 0 failures
```

「4 で割り切れる」なら true、「4 で割り切れない」なら false という処理になっているので、そのままで問題ないことがわかります。次の「テストコードが通過するようにプログラムを書き換える」というステップは不要でした。また「テストコードを追加する状態を維持しながらプログラムを改善する」も、まだ必要ないようなのでスキップします。

これで「4 で割り切れない年は平年とする」も満たすことができました。

- ~~4 で割り切れない年は平年とする~~
- ~~4 で割り切れる年はうるう年とする~~
- ただし、100 で割り切れる年は平年とする
- ただし、400 で割り切れる年はうるう年とする

このように、プログラムを書いていく過程でやるべきことが変化したり、より簡単な条件に置き換えられることがあります。また、テスト駆動開発のステップは必要に応じてスキップすることができます。

第 3 のテストと明白な実装

次は「ただし、100 で割り切れる年は平年とする」というルールに手をつけることにします。

まずテストを書きましょう。

```
# leap_spec.rb

require_relative './leap'

RSpec.describe Date do
  describe '.leap?' do
    context '4 で割り切れる年' do
      it 'うるう年になる' do
        expect(Date.leap?(4)).to be true
      end
    end

    context '4 で割り切れない年' do
      it '平年になる' do
        expect(Date.leap?(5)).to be false
      end
    end

    context '100 で割り切れる年' do
      it '平年になる' do
        expect(Date.leap?(100)).to be false
      end
    end
  end
end
```

4 で割り切れるルールのため、うるう年と判定されるはずです。予想通りに失敗して、レッドになることを確かめます。

```
$ rspec leap_spec.rb
..F

Failures:

  1) Date.leap? 100 で割り切れる年 平年になる
     Failure/Error: expect(Date.leap?(100)).to be false

       expected false
```

```
            got true
       # ./leap_spec.rb:21:in block (4 levels) in <top (required)>'

Finished in 0.02186 seconds (files took 0.18779 seconds to load)
3 examples, 1 failure

Failed examples:

rspec ./leap_spec.rb:20 # Date.leap? 100で割り切れる年 平年になる
```

どのようにプログラムを直せばいいのかは明らかです。修正してみましょう。

```
# leap.rb

require 'date'

class Date
  def self.leap?(year)
    return false if year % 100 == 0

    year % 4 == 0
  end
end
```

無事にテストが通りました。グリーンです。
プログラムを修正する必要はないようなので、リファクタリングのステップは省略します。
このように「仮実装」を使わずに正しいプログラムに直接修正することを「明白な実装」と呼びます。

- ~~4で割り切れない年は平年とする~~
- ~~4で割り切れる年はうるう年とする~~
- ~~ただし、100で割り切れる年は平年とする~~
- ただし、400で割り切れる年はうるう年とする

最後のテスト

最後に残った「ただし、400で割り切れる年はうるう年とする」に手をつけましょう。

まずテストを書いて、正しく失敗することを確かめ、レッドにします。

```
# leap_spec.rb

require_relative './leap'

RSpec.describe Date do
  describe '.leap?' do
    context '4で割り切れる年' do
      it 'うるう年になる' do
        expect(Date.leap?(4)).to be true
      end
    end

    context '4で割り切れない年' do
      it '平年になる' do
        expect(Date.leap?(5)).to be false
      end
    end

    context '100で割り切れる年' do
      it '平年になる' do
        expect(Date.leap?(100)).to be false
      end
    end

    context '400で割り切れる年' do
      it 'うるう年になる' do
        expect(Date.leap?(400)).to be true
      end
    end
  end
end
```

そしてグリーンにします。

```
# leap.rb

require 'date'

class Date
  def self.leap?(year)
    return true if year % 400 == 0
    return false if year % 100 == 0

    year % 4 == 0
  end
end
```

最後にリファクタリングのステップに移りましょう。このプログラムに改善できるところはあるでしょうか？　すこし複雑な条件文ですが、テストコードがあるので、誤ってプログラムを壊してしまうことを心配せずに変更を試すことができます。

十分に改善できたと感じたら、うるう年を判定する「動作するきれいなコード」の完成です。

- ~~4で割り切れない年は平年とする~~
- ~~4で割り切れる年はうるう年とする~~
- ~~ただし、100で割り切れる年は平年とする~~
- ~~ただし、400で割り切れる年はうるう年とする~~

3.4.3 ふりかえり

テスト駆動開発のステップを繰り返しながらプログラムを実際に作成してみました。実際のテスト駆動開発でも、課題の難しさに応じて「仮実装」と「明白な実装」で歩幅の大きさを調節しながら一歩ずつ進んでいくことになります。

テスト駆動開発を実際のプログラムに適用するためには練習が必要となります。また、テストから実装方針を検討するための「三角測量」などのテクニックや、大きなプログラムを「グリーン」に保ちながら少しずつ書き換えていく手法についてはここでは扱いませんでした。関心のある人は章末の参考文献の『テスト駆動開発』を参照してください。

3.4.4 テスト駆動開発の意義

テスト駆動開発では、「動作するきれいなコード」という目標を「動作するコード」と「きれいなコード」に分割します。そして、テストコードで後戻りを防ぎながら一歩ずつ完成に近づいていきます。
アジャイル開発は顧客から小刻みにフィードバックを受け取ってプロジェクトを適切な方向に進めていく開発手法でした。テスト駆動開発は、テストコードから小刻みにフィードバックを受け取って、プログラムを適切な方向に育てていく開発手法と考えることができます。
テストコードの成功・失敗がフィードバックにあたります。また、紙幅の都合から具体例を上げることはできませんでしたが、テストコードから先に作ることそのものがプロ

グラムの設計に対するフィードバックとなり、よりよい設計を考えるきっかけになります。
とはいえ、細かくフィードバックを受けなくても前に進んでいける局面もあります。テスト駆動開発を学んだからといって、常にテスト駆動開発を使わないといけないわけではありません。「動作するきれいなコード」にひと息でたどりつけるときには、あえて回り道をする必要はないでしょう。もし道を見失ったときには、テスト駆動開発のテクニックを思い出してみてください。

3.4.5 テスト駆動開発の限界

テスト駆動開発がうまく適用できない場面もあります。以下のようなポイントについては、自動テストから適切なフィードバックを得るのは難しいでしょう。
- 使い勝手のよいユーザーインターフェースになっているか: 自動テストを使えば、仕様通りのインターフェースが作れているかを確認することはできます。しかし、インターフェースが使いやすいかどうか、また、インターフェースをどのように改善するべきかといったフィードバックを得ることは難しいでしょう。
- 並行処理が正しく実装できているか: 自動テストで状況を再現するのが難しい分野がいくつかあります。処理の実行タイミングに左右される並行処理はその代表例です。
また、テスト駆動開発で作られたテストコードは製品の品質を保証するテストとしては必ずしも十分ではありません。テストケースを通過することは確かめられますが、テストケースそのものが十分かどうかはわからないのです。（たとえば、今回作った `Date.leap?` メソッドに小数を渡したら、どのような結果を返すべきでしょうか？）

3.4.6 まとめ

ここではテスト駆動開発の目的と進め方、意義と限界について学びました。ポイントをおさらいしておきましょう。

- テスト駆動開発は「動作するきれいなコード」を作ることを目的とする
- テスト駆動開発はテストコードからフィードバックを得て、小刻みに開発を進める
- テスト駆動開発がうまく適用できない場面もある

3.4.7 発展課題

- これまでに書いたプログラムをテスト駆動開発で作り直してみましょう。どれだけ細かいステップに分解できるでしょうか？
- 最寄りの映画館のチケット料金の計算ルールを調べて、チケット料金を計算するプログラムをテスト駆動開発で作ってみましょう。

3.4.8 参考資料

- 和田卓人「50 分でわかるテスト駆動開発」
  https://channel9.msdn.com/Events/de-code/2017/DO03
- テスト駆動開発の実演を含む講演の動画です。テスト駆動開発の意義と進め方がコンパクトに解説されています。
- Kent Beck『テスト駆動開発』、和田卓人訳、オーム社、2017 年
- テスト駆動開発の提唱者による書籍の新訳版です。テスト駆動開発の意義が詳細に説明されています。また、複雑なプログラムをテスト駆動開発で作る実例が盛り込まれています。
- Martin Fowler ほか「Is TDD dead?」https://martinfowler.com/articles/is-tdd-dead/
- 2014 年に Ruby on Rails の作者が提起した「テスト駆動開発は有効な開発手法ではない」という主張をめぐる議論をまとめたページです。テスト駆動開発を実践する際の課題とその解決策が語られています。

3.5[評価課題]自動テストとテスト駆動開発

ここで、自動テストについて学んできたことのポイントを確認してみましょう。

3.5.1 自動テスト

プログラムのコンポーネントを対象にするテストを(Q1: 　　　　　　　　)、または(Q2: 　　　　　　　)と呼びます。

自動テストに関するアジャイル開発のプラクティスのひとつが(Q3: 　　　　　　　)インテグレーションです。成果物を素早くメインラインにマージすることを目指します。

以下のような仕様のプログラムの作成を依頼したところ、

- 3 で割り切れる場合には、"Fizz"を出力する

- 5 で割り切れる場合には、"Buzz"を出力する
- 3 と 5 の両方で割り切れる場合には、"FizzBuzz"を出力する
- どれでもない場合には与えられた数値を出力する

誤って以下のようなプログラムが作成されてしまいました。

```
def fizzbuzz(n)
  case n
  when 0...3
    n
  when 3...5
    'Fizz'
  when 5...15
    'Buzz'
  when 15...Float::INFINITY
    'FizzBuzz'
  else
    n
  end
end
```

誤りを検出するためのテストケースをひとつ作成してください。

入力	期待する出力
(Q4:)	(Q5:)

3.5.2 テスティングフレームワーク

(Q6:)は Ruby 本体に gem(ライブラリ)として組み込まれているテスティングフレームワークです。テストケースのまとまりを Ruby の(Q7:)として表し、個々のテストケースはインスタンスメソッドとして記述します。

Q6 のテスティングフレームワークを使って、以下の Array クラスのテストコードを完成させてください。

```
class TestArray < (Q8:              )

  # 3 つの要素を持つ Array オブジェクトの size は 3
  def test_size
    array = [0, 1, 2]
    (Q9:                )
  end

  # 要素のない Array オブジェクトの empty は true
  def test_empty
```

```
    array = []
    (Q10:               )
  end
end
```

RSpec を使って、以下の Array クラスのテストコードを完成させてください。

```
RSpec.(Q11:         ) Array do
  describe '#size' do
    it '要素の数を返す' do
      array = [0, 1, 2]
      (Q12:         )
    end
  end

  describe '#empty?' do
    (Q13:         ) do
      it 'true を返す' do
        array = []
        (Q14:         )
      end
    end
  end
end
```

3.5.3 テスト駆動開発

テスト駆動開発は「(Q15:   )コード」を作ることを目的とする開発手法です。
テスト駆動開発では、プログラム本体を変更する前にテストコードを書きます。これを(Q16:   )といいます。テストコードを書いた直後で、テストが失敗している状態のことを(Q17:   )と呼びます。テストが成功する状態を(Q18:   )と呼び、この状態を保ったままでプログラムを修正することを(Q19:   )と言います。

テストを成功させるための一時的な修正は(Q20:   )と呼ばれます。それに対して、Q20 を使わずに正しいプログラムに直接修正することを「明白な実装」と言います。

第４章　Ruby on Rails 基礎

第４章　Ruby on Rails 基礎

4.1 Ruby on Rails：Web システム概念

4.1.1 インターネットの概要

インターネットとは、TCP/IP というプロトコルを利用して、コンピュータや、スマートフォン等ネットワーク機能を内蔵した電子機器で構成されるネットワークです。
そのネットワーク上で、電子メールでのテキストの送受信、スマートフォンで撮影したデジタル画像の送受信、WWW、SNS での情報交換、クラウドサービスの利用等ができます。
機器の接続には、有線と無線(WiFi 等)があり、一般的にインターネットに接続している機器は、異なる IP アドレスを持ち、自分と相手を特定して通信を行います。
インターネットは、プライベートのネットワークではありません。大勢の人の努力の結果、インターネットに接続することで、地球の裏側にあるインターネットに接続されている機器やサービスとも接続をすることができます。つまり、インターネットはひとかたまりのネットワーク(The Internet)です。

インターネット上で提供されているサービスには、自分を証明するアカウントを持たなくても利用できるサービスと、アカウントが必要なサービスがあります。
どちらの場合でも、サービスを提供する側はセキュリティを考慮する必要があります。
セキュリティを考慮しないと、たとえば、だれでも利用できるサービスの場合は、提供している文字や画像、動画、プログラム等を提供者の意図と異なるものに差し替えられる等が考えられます。また、アカウントが必要なサービスの場合は、アカウント名やパスワードを盗まれて、秘密であるはずの情報が第三者に閲覧、取得されてしまうこと等が考えられます。ですので、インターネットに接続するサービスでは、提供するコンテンツへのアクセス権や、暗号化、2 段階認証等の技術を利用してセキュリティを十分考慮する必要があります。

4.1.2 HTTP プロトコルと Web サーバの概要

インターネットで利用できるサービスは、いろいろありますが、ここでは、特に Web サービスの構成に利用されるプロトコル、HTTP について解説します。

一般に HTTP は、HTTP サーバー(Web サーバー)と HTTP クライアント(Web ブラウザ)間で利用されます。
HTTP サーバーは、HTTP クライアントからテキストのリクエスト、画像のリクエスト、画像の登録等のリクエストがあるのを待機しています。一般的に TCP/IP の 80 番ポートが利用されます。
HTTP クライアントは、利用したいサーバーの TCP/IP の 80 番ポートをめがけてリクエストをします。

たとえば、Google に接続してサービスを利用したい場合は Web ブラウザのアドレス欄に、http://www.google.co.jp と入力します。
一般的に HTTP サーバとの通信は 80 番ポートを使うことになっているので、HTTP クライアントでポート番号を省略すると 80 番ポートとして動作します。
つまり、http://www.google.co.jp:80 と入力したのと同じです。
ただし 80 番ポートはデータ通信をする際に暗号化されないので、秘密にしたい情報のやりとりには向きません。その場合は、セキュアなプロトコル SSL/TLS で保護された通信を利用します。一般的に HTTPS での通信は 443 番ポートが利用されます。
こちらは、https://www.google.co.jp と入力します。https://www.google.co.jp:443 と入力したのと同じです。

ただし、実際には、http://www.google.co.jp と入力すると、google の HTTP サーバのポリシーによって、セキュアな接続を利用するように https://www.google.co.jp へリダイレクト(転送)されるサービスもあります。

HTTP サーバーは HTTP クライアントからリクエストを受け付けると、HTTP クライアントの画面に表示される文字情報、画像、音声、動画等目に見える情報(HTML や PNG,MP4 等)と、それらを配置するレイアウト情報(CSS)、HTTP クライアント側で動作するプログラム(JavaScript)等を要求のあったクライアントに向けて返信します。
1 つのサーバーが同時に複数のリクエストを受け付けても、リクエストをした HTTP クライアントには固有の IP アドレスが割り当てられていて、その IP アドレスも含めてリクエストするので、それぞれの HTTP クライアントのリクエストに応じた適切な内容を返信します。
HTTP は、リクエストへの返信が完了すると、一旦、HTTP クライアントとの接続を切断します。つまり、次に同じ HTTP クライアントからリクエストがあっても、以前にリクエストされた HTTP クライアントと同一なのか判断できません。
このままでは、ショッピングサイトで、買い物をしようと商品をカートに入れても、別の商品をカートに入れる際、最初のカートの情報がわからなくなってしまい、いつまでもレジに進むことができません。

そこで、セッション管理機能を使うことで、HTTP サーバーが最初のリクエストを受け付けた時に、HTTP クライアントに返信するデータにセッション情報を含めて、次の HTTP クライアントからのリクエストには、そのセッション情報も含めてもらうことで、別の商品をカートに入れる際も、最初のカート番号が識別できるようになります。

4.1.3 Web ブラウザ、HTML,JavaScript の概要

HTTP クライアントは、Web ブラウザとも呼ばれます。Web ブラウザの一例として、Internet Exploer, Chrome, Firefox, Safari 等があります。
Web ブラウザには、HTTP サーバにリクエストを送信する機能、返信された文字や画像等のコンテンツを受信して、定義されているレイアウトに応じてコンテンツを配置する機能、動画や音声を再生する機能、JavaScript で記述された Web ブラウザ側で動作するプログラムを実行する機能、パソコンやスマートフォンのカメラやマイクを利用する機能等があります。

また、セキュアな接続が必要な場合は、その接続先が信頼できるか確認したり、暗号化した通信をする機能も含まれます。

```
HTML,CSS,JavaScript を含んだコンテンツの例:
<!DOCTYPE html>
<html>
<head>
    <meta charset="UTF-8">
    <title>HTML の例</title>
    <script>
        function hi() {
            alert("Hi!");
        }
    </script>
    <style>
        .red {
            color: #FF0000;
        }
    </style>
</head>
<body>
    <div class="red">
        <p>HTML の例</p>
        <input type="button" value="ボタン" onclick="hi()">
    </div>
</body>
</html>
```

① HTML

HTTPのリクエストに対する応答の本文です。
<body>と</body>のようにタグと呼ばれる文字を利用しながらコンテンツの内容、構造を記述します。タグの中にタグを含めることができます。

② CSS

上記の例では<style>タグに記述していますが、別のファイルに記述することが多いです。
CSSは主にコンテンツの装飾を担います。
コンテンツの装飾はわざわざCSSに分けなくても記述できますが、コンテンツ本体と装飾が混在した状態では記述内容が分かりにくいこと、HTMLに含まれるコンテンツの再利用がしづらいこと等の理由から、現在は分割するスタイルが主流です。

③ JavaScript

Webブラウザ側で動作させることができるプログラム言語です。
上記の例では<script>タグにプログラムを記述し、ボタンをクリックしたらJavaScriptのプログラムを実行しています。
例にはありませんが、変数も使用できますし、functionで様々なメソッド(処理のかたまり)を作成して、WebブラウザからHTTPサーバーへ送信するリクエストの内容を確認したり、入力するデータの補助機能を記述したり、音や画像を使ったゲームを記述することもできます。

4.2 Ruby on Rails:View の機能

ここでは、Ruby on Rails の「View の機能」、「SASS」「SCSS」「ERB」について説明します。

4.2.1 SASS/SCSS とは

SASS とは、「Syntactically Awesome StyleSheet」の略で、CSS を効率的に記述できるように開発された言語です。 簡単に言えば「CSS をプログラムの要素を入れて効率的に記述できるようにした言語」です。 SASS では、変数を利用したり、演算を行ったり、関数を作成・利用したり、「@extend」を使用しスタイルを継承したり、「@import」でファイルをインポートしたり、よりプログラムらしく書くことで、効率よくコーティングでき、より保守性の高いプログラムを作成することができます。この仕組みは SCSS でも同様です。

4.2.2 SASS と SCSS の違い

SCSS と SASS の違いは、記述方法にあります。 SASS はインデント構文です。 インデント構文は、先頭からのスペースの数に意味があります。スタイルブロックを区切る{}の代わりにインデントを使用し、プロパティを区切る;の代わりに改行を用いて記述します。SASS はスタイルブロックをインデントで表したり、;を改行で表しますが、SCSS では、{}を使用しスタイルブロックを表します。 またファイルの拡張子も SASS の場合「.sass」、SCSS の場合「.scss」となります。 SCSS は、SASS と CSS との互換性が不十分だったために開発された言語で、SCSS の書き方だけではなく、CSS の書き方でそのまま記述することも可能なため、一般的に SCSS の方が使用されています。

SASS の場合

```
#main
  color: red
  font-size: 12px
```

CSS にコンパイル後

```
#main {
  color: red;
  font-size: 12px;
}
```

今回は一般的に使われている SCSS の書き方について見ていきます。

4.2.3 SCSS の書き方

SCSS でコメントを記述する

コメント行先頭に/*を付ける。 インデントを加えてコメント行とする。

```
// 単一行のコメント

/*
  複数行の
  コメントです
*/

#main {
  color: blue;
  font-size: 12px;
}
```

変数を定義する

先頭に「$」を付ける

```
$main_color: yellow;

#main {
  color: $main_color;
  font-size: 12px;
}
CSS にコンパイル後
#main {
  color: yellow;
  font-size: 12px;
}
```

演算を行う

下記のようにプロパティの値に対して演算を行うことができます。

```
$main_height: 500px;
$main_width: 500px;


#main {
  height: $main_height + 100;
  width: $main_width / 2;
  background-color: red;
}
```

CSS にコンパイル後

```
#main {
```

```
  height: 600px;
  width: 250px;
  background-color: red;
}
```

関数を使う

```
@function Double($value) {
  @return $value * 2;
}

@function Halve($value) {
  @return $value / 2;
}

.box {
  height: Double(100px);
  width: Halve(500px);
  background-color: green;
}
```

CSS にコンパイル後

```
.box {
  height: 200px;
  width: 250px;
  background-color: green;
}
```

@extend でスタイルを継承する

「@extend」を使用し他のスタイル宣言ブロックを継承することができます。

```
.main_box {
  background-color: yellow;
  height: 400px;
  width: 500px;
}

.sub_box1 {
  @extend .main_box;
  background-color: blue;
}
.sub_box2 {
  @extend .main_box;
  background-color: red;
}
```

CSS にコンパイル後

```
.sub_box1 {
  background-color: blue;
  height: 400px;
  width: 500px;
}
```

```
.sub_box2 {
  background-color: red;
  height: 400px;
  width: 500px;
}
```

@import でファイルを読み込む

1 つの SCSS ファイルで全てのスタイルを書くと記述量が膨大になるため、CSS ファイルを分割し別ファイルでまとめて読み込むといった方法が用いられます。 ファイルを読み込む場合「@import」を使用します。 拡張子を省略することも可能です。

```
@import "top.scss";
@import "index";
```

「@mixin」を使用する

「@mixin」で処理をひとまとめにし、「@include」でまとめた処理を呼び出します。

```
@mixin border($color: yellow) { // 引数にデフォルト値を指定している
  border: solid $color;
  height: 300px;
  width: 300px;
}

#header {
  @include border;
}

#footer {
  @include border(gray);
  height: 200px;
  width: 200px;
}
```

CSS にコンパイル後

```
#header {
  border-top: 2px solid white;
}

#footer {
  border-top: 2px solid gray;
}
```

4.2.4 ERB とは

ERB は、「Embedded Ruby」の略で、Rails での基本的な View 開発に使用されます。 拡張子に「.erb」をつけることで ruby のコードを埋め込むことができます。 Rails で使用される「.html.erb」ファイルは Ruby のスクリプトが埋め込まれた HTML ファイルと考えてよいでしょう。 Ruby のスクリプトを埋め込むことができるので、html ファイル内で if 文を使用した条件分岐や繰り返し構文などの処理も記述できます。

4.2.5 ERB の書き方

実際に Ruby のスクリプトを埋め込む記述方法を説明します。
「`<% %>`」 と 「`<%= %>`」を使用し Ruby のコードを記述します。

```
<% Ruby のコード %>
<%= Ruby のコード %>
```

「`<% %>`」と「`<%= %>`」の違いは下記の通りです。
「`<% %>`」はブロック内のコードを実行するだけで、値は返しません。「`<%= %>`」は、ブロック内の式を評価結果を返します。

```
<% price = 10000 %>  <!-- 代入処理を行うのみ -->
<% result = price * 1.1 %>  <!-- 代入処理を行うのみ -->
%>
<div><%= result %></div>  <!-- 変数 result の値を返す -->
```

このように HTML ファイルに Ruby のスクリプトを埋め込むことで、Controller から送られてきた配列やハッシュなどのオブジェクトに対して繰り返し構文を使用し値を参照し、HTML で表示するなどといった処理を記述することが可能です。

4.2.6 Rails プロジェクトでの SCSS と ERB の使い方

Rails で SASS(SCSS)を使用するには、 `sass-rails` という gem をインストールする必要がありますが、Rails3.1 以降から `Rails new` でプロジェクトを作成すれば自動的にインストールされる仕様となったため、 `sass-rails` をインストールするために必要な作業は特にありません。 また SASS(SCSS)は、最終的にブラウザが解釈できる CSS にコンパイルす

る必要がありますが、Rails では、アセットパイプラインで、「`.scss`」から「`.css`」へのコンパイルは自動的に行われるので、これについても特に意識する必要はありません。

4.3 Ruby on Rails：Rails 基礎 1

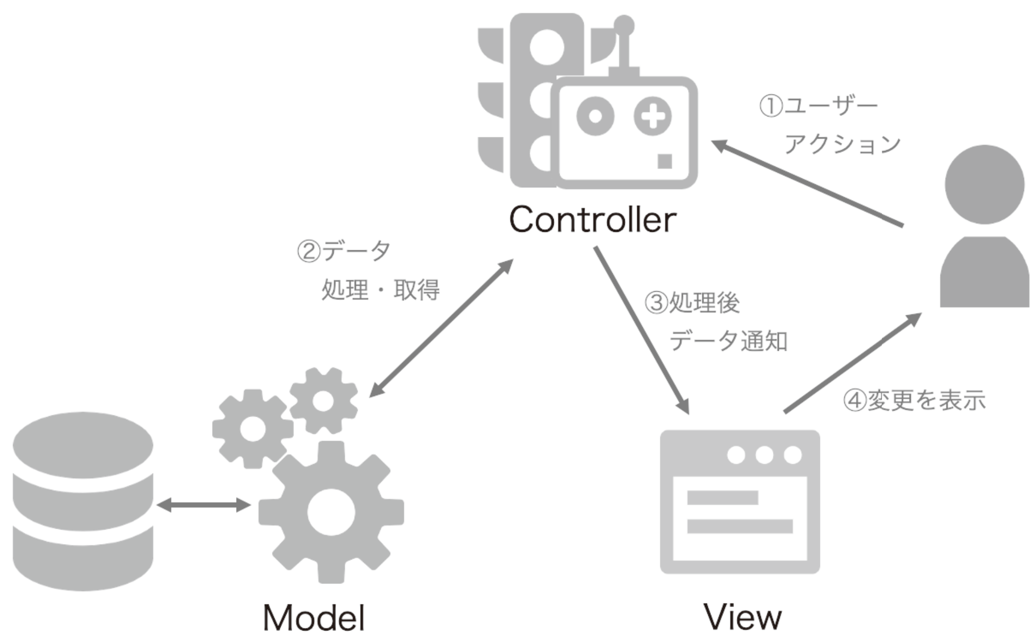
4.3.1 Ruby on Rails について

【Ruby on Rails とは】 Ruby on Rails とは、Ruby 言語製の Web アプリケーションフレームワークです。 とても強力なフレームワークで、WEB アプリを開発する上で便利な機能が多く搭載されています。

2004 年に最初のバージョンが公開されてから、その便利さが話題になり多くの人に広がっていきました。 実際、業務として Ruby を扱う場合は、Rails での WEB アプリ開発がそのほとんどを占めています。

【Rails を知る前に知っておきたいこと】 Rails の学習を進める前に、アーキテクチャや基本理念について解説しておきましょう

【MVC】 MVC とは、WEB アプリのような UI を持つアプリを実装する際に使用される、デザインパターンの１つです。 ちなみに「MVC」という名前は、“Model”,“View”,“Controller”の頭文字を取った略称です。



それぞれの役割は以下のようになります。

- \- Model データの処理を取り扱う部分です。 データベース（DB）とやり取りをして、データの登録・取得・更新・削除等を行うことが出来ます。 DB 関連だけ

ではなく、View や Controller に依存しないデータの処理等は Model に記載して
おいて、 View や Controller から呼び出して利用したりもします。

- View ユーザの見える範囲を取り扱う部分です。 主に画面に表示する HTML 等を管理する事になります。
- Controller クライアントからの個々のリクエストに応じた処理を行います。 リクエストに応じて、Model にアクセスしデータの取得・更新を行ったり、 その結果を描画する View に引き渡すのが、Controller の役割です。

【DRY】 DRY とは、“Don’t Repeat Yourself”の略称です。 プログラミングを行う際に心がける理念の１つであり、 “同じような記述を繰り返してはいけない”といった意味になります。

全く同じコードや似たようなコードがあちこちに散らばっていると、後々メンテナンスをする際に、全て変更するのに時間がかかったり、カバー漏れで不具合が出てしまうことになりかねません。 そうならないためにも、出来るだけまとめられるものは一箇所にまとめて管理しておこう、という考え方です。

【CoC】 CoC とは、“Convention over Configuration”の略称です。 意味は“設定より規約”で、従来のフレームワークのように設定を大量に記述するのではなく、 個々の設定に悩むより規約をあらかじめ決めておいて、開発者はロジックを考えることに専念できるようにしよう、というアプリ開発の思想の１つです。

具体的に言えば、Rails では、あらかじめ名前付けのルールが用意されています。 例えば、次のコマンドを実行すると、Model は User(単数形)、DB のテーブル名は users(複数形)で作成されるルールがあります。

```
rails generate model user
```

このような厳格なルールがあるため、命名規則等を熟考をしなくてよくなるだけでなく、 アプリの構造自体がシンプルになったり、他の開発者との意思疎通が容易になったりと、良い面が多くあります。

Rails にはこれらを補助する機能が備わっています。 活用していけば、とても心強い味方になってくれるでしょう。

4.3.2 Cloud9 上で Rails アプリケーションを作成

① ワークスペースの作成

まず Cloud9 上にワークスペースを作成しましょう。

実はここで template に Ruby を選択すると、後の工程にある Rails のインストールや Rails のアプリ作成を、ワークスペースの作成と同時に行ってくれます。 ただし、今回は Rails のバージョンを指定してインストールしたいので、template は Blank にしておきましょう。

ワークスペースが出来たら、続いて Rails のインストールに移りましょう。

④　Rails のインストール

今の状態では、Rails はまだインストールされていません。 Rails は RSpec 等と同じで gem として配布されていますので、以下のコマンドでインストールすることが出来ます。

```
gem install rails -v 5.1.3
```

-v オプションを付けると、特定のバージョンを指定してインストールすることが出来ます。 コマンドを実行すると、Rails 本体以外にも必要な gem がいくつかインストールされます。 すべてインストールされれば完了です。

⑤　Rails アプリの作成

それでは早速、インストールした Rails を使用してアプリを作成しましょう。 アプリの作成は以下のコマンドで行います。

```
rails new monka-railsbasic
```

これで“monka-railsbasic”という名前のアプリ（プロジェクト）の雛形が出来上がります。 コマンドを実行すると以下のように、フォルダやファイルが大量に生成されているはずです。

こう見ると何だかややこしそうですが、初めからこの全てを理解する必要はありません。 必要な部分や重要な部分は後ほど解説します。

今後の操作はアプリの中で行いたいので、以下のコマンドでディレクトリを移動しておきましょう。

```
cd monka-railsbasic
```

次に以下のコマンドを実行してみてください。

```
rails db:create
```

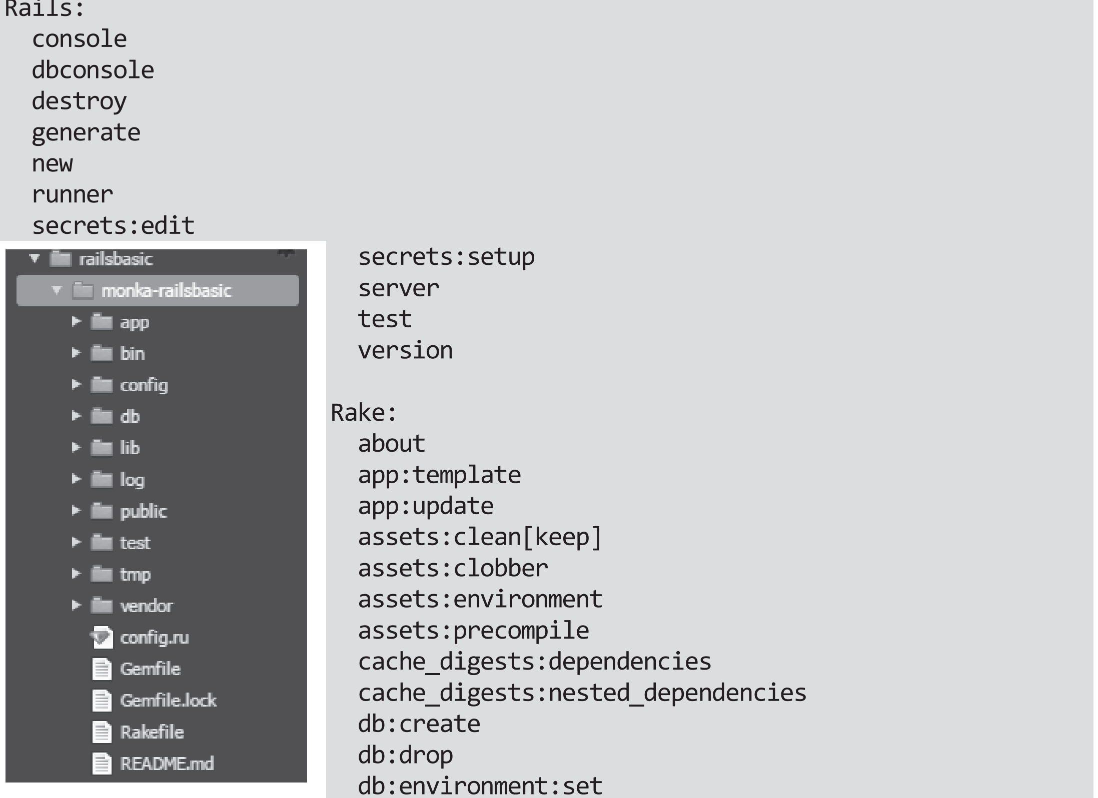
rails は Rails で使えるコマンドで、DB 操作やテストの一括実施等が出来ます。 上記のコマンドは DB 操作で、一番最初の DB を作成するコマンドです。 まだ中身は空っぽですが、これで一応接続先の DB が作成できました。
実はこの状態で、もう既にアプリとして最低限の機能は使えるようになっています。 試しに以下のコマンドを実行してみてください。

```
rails server -b 0.0.0.0 -p 8080
```

rails server が Rails アプリを起動するコマンドで、後の部分はオプションです。 -b 0.0.0.0 は bind オプションで、IP アドレス等を制御します。 -p 8080 は port オプションで、アクセスポート番号を指定します。
コンソール内に表示されている、「http://0.0.0.0:8080」 をクリックして、 “open”を選ぶとブラウザで画面が表示されます。
まだデフォルトの画面が表示されるだけですが、きちんと表示されていますね。 ここをベースにアプリを作成していくことになります。
ここで、rails コマンドについてもう少し調べてみましょう。 以下のコマンドを実行してみてください。

```
rails --help
```

次の一覧が表示されます。




```
Rails:
  console
  dbconsole
  destroy
  generate
  new
  runner
  secrets:edit
  secrets:setup
  server
  test
  version

Rake:
  about
  app:template
  app:update
  assets:clean[keep]
  assets:clobber
  assets:environment
  assets:precompile
  cache_digests:dependencies
  cache_digests:nested_dependencies
  db:create
  db:drop
  db:environment:set
```

```
  db:fixtures:load
  db:migrate
  db:migrate:status
  db:rollback
  db:schema:cache:clear
  db:schema:cache:dump
  db:schema:dump
  db:schema:load
  db:seed
  db:setup
  db:structure:dump
  db:structure:load
  db:version
  dev:cache
  initializers
  log:clear
  middleware
  notes
  notes:custom
  restart
  routes
  secret
  stats
  test
  test:db
  test:system
  time:zones[country_or_offset]
  tmp:clear
  tmp:create
  yarn:install
```

Rails4 以前は、各種コマンドが rails と rake に分散していましたが、 Rails5 からは、rake コマンドを rails コマンドで実行できるようになっています。 一覧に表示された Rake:以下のコマンドは、rails コマンドで実行可能となっています。 これは、rake コマンドと rails コマンドの使い分けに論理的な理由がなかったため、rails コマンドに統一しようという方向性のためです。

4.4 Ruby on Rails ： Rails 基礎 2

4.4.1 アプリの構造解説

重要な部分を抜粋して紹介していきます。

- app

アプリケーション本体が格納される

- assets

WEB ページ内で使用する image ファイルや、ページのレイアウトに使用する css、ページの動きを制御する js 等が格納される

- controllers

ユーザアクションを基にアプリを制御する controller が格納される
MVC の“C”

- helpers

「ヘルパーメソッド」と呼ばれるメソッドをまとめたファイルが格納される
「ヘルパーメソッド」とは、主に view を記述する際に役立てるメソッドであり、 フォーム要素の生成、文字列や数値の整形するメソッド等、view でよく利用する操作がデフォルトで用意されている 例えば、link_to メソッドでは、与えられた引数を元にハイパーリンクを生成することができる ここでは、独自に使用するヘルパーメソッドを定義する

- models

データの処理全般を管理する model が格納される
MVC の“M”

- views

画面に表示する部分の view が格納される
MVC の“V”

- config

Rails アプリの設定に関するファイルが格納される
ルーティングを制御する routes.rb 等が格納されている
ルーティングとは、ブラウザからのリクエスト(URL)をサーバ側の Rails と結びつける仕組みである

- db

  DB 関係のファイルが格納される
  DB のテーブルをアプリ側から操作出来るようにした migrattion ファイルや、
  DB の初期投入データを管理できる seeds.rb 等が格納されている

- test

  テスト関係のファイルが格納される
  今回は使用せず、RSpec 導入したあとに出来る spec フォルダを代わりに使用する

- Gemfile

  アプリで使用する gem をまとめたファイル
  gem の種類だけでなく、バージョンや使う環境を限定出来る
  では次からは、実際に簡単なアプリとしてカスタマイズしていきながら、

Rails の各種機能に触れて学んでいきましょう。

4.4.2 Rails の基本

【scaffold コマンド】

自前でページを作成するには、必要なものが多くあります。
例えば画面に表示する View であったり、登録処理をするならそのためのテーブルも必要ですし、テーブルを操作するための Model や、それらを制御する Controller も必要です。
また、WEB ページにアクセスした時に画面表示等の処理を行うには、ブラウザで入力した URL とアプリの処理を連動できるようにしなければいけません。
そのためには、routes.rb というファイルに設定を書き込む必要があります。（ルーティング）この、WEB ページで何か機能を使用するために必要なものの塊をリソースと呼びます。
リソースを全て 1 から作るのは中々大変ですが、Rails ではこのリソースを一括で作成できる scaffold という機能があります。早速この機能を使用してみましょう。

```
rails generate scaffold User name:string email:string
```

rails generate がファイルを作成するコマンドで、scaffold はそのうちのモードの 1 つです。
User は生成されるファイルの名前のベースになります。
生成されるファイルは大体命名規則があるので、それに則ってファイルの名前を付けてくれます。

あとの name:string email:string は、テーブルで使用するカラムを予約しておくためのものです。
この scaffold をした段階では、まだテーブルだけは生成されていないので後でも大丈夫なのですが、この時点で予約しておくとちょっと便利なことがあるので、オプションとして付けておきます。
このコマンドを実行すると、また新たにファイルやフォルダが生成されていると思います。
それらについては、また後ほど個別に解説していきます。

【CRUD について】

Web アプリケーションには、基本となる新規作成（create）、表示（read）、更新（update）、削除（destroy）の 4 つの機能があります。これらの頭文字を取って CRUD と呼ばれています。 scaffold では、新規作成、表示、更新、削除の各機能も一括で作成されています。

【テーブルの作成】

先に話した通り、DB のテーブルだけはまだ生成されていません。
ただし、実はテーブルを生成するための準備はもう出来ています。
そもそもテーブルを作るためには、本来なら DB のコマンドを直接実行して DB を操作しなくてはいけません。
しかし、Rails ではアプリ側から DB を操作することが出来ます。
その DB の操作に使われるのが、migration という機能です。
この migration では、テーブルを生成したり、テーブルのカラムを変更したりすることが出来ます。「準備が出来ている」と言ったのは、先程 scaffold を使用した時に、この migration の機能を使うためのファイルが出来ているからです。
そのファイルは/db/migrate/の配下にあり、ファイル名は日時_create_users.rb になっています。
中身を見てみると、以下のようになっています。

```
class CreateUsers < ActiveRecord::Migration[5.0]
  def change
    create_table :users do |t|
      t.string :name
      t.string :email

      t.timestamps
    end
  end
end
```

コマンドのオプションとして付けておいたカラムが入っていますね。

他にも timestamps という記述がありますが、これは scaffold で生成すると自動で書き込まれるものです。この記述を入れておくと、テーブルにレコードの作成日時と更新日時を記録するためのカラムが追加されます。
この 2 つの日時は、登録・更新時に勝手に記録してくれる等、Rails 側で管理してくれるようになっています。
このファイルを使用して migration を行うには、以下のコマンドを実行します。

```
rails db:migrate
```

これでテーブルが生成されるので、リソースが機能として使用出来るようになりました。
試しに Rails アプリを起動して、ブラウザでページを開いたあと、URL の末尾に/users と付け足してみて下さい。 その URL でページを移動すると、ほぼ真っ白な Users と書かれたページが表示されていると思います。
これはテーブルの情報を一覧で表示するページで、「New User」のリンクを押すと新規登録ページに移動します。新規登録したあとには一覧ページに登録したデータが表示され、そこから編集等の操作が出来るようになっています。
このように、まだデザインもシンプルで機能も最低限に留まってはいますが、WEB ページとして欲しい機能の基礎が出来上がりました。
これはどういう仕組みで動いているのでしょうか？
具体的な仕組みや構造を、生成された各ファイルを見ながら辿ってみましょう。

【routes.rb】

まず初めに、ルーティングの確認をしてみましょう。
先程 URL の末尾に/users を付け足して、きちんとページが表示されたということは、
その URL にアプリが対応できるようになったということです。
その設定は/config/routes.rb のファイルに書かれているはずなので、そのファイルを確認してみましょう。

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

記述の中に resources :users と書かれている部分がありますね。
実はこの一行のコードで、8 個分のルーティングが設定できるようになっています。
このルーティングの設定は、以下のコマンドで確認できます。
Rails アプリを実行中の場合は、もう一つターミナルを立ち上げて実行してみてください。

```
rails routes
```

実行すると以下のような内容が表示されるはずです。

```
   Prefix Verb   URI Pattern                  Controller#Action
    users GET    /users(.:format)             users#index
          POST   /users(.:format)             users#create
 new_user GET    /users/new(.:format)         users#new
edit_user GET    /users/:id/edit(.:format)    users#edit
     user GET    /users/:id(.:format)         users#show
          PATCH  /users/:id(.:format)         users#update
          PUT    /users/:id(.:format)         users#update
          DELETE /users/:id(.:format)         users#destroy
```

右端の列にあるのは、アクションメソッドと呼ばれる Controller に定義されたメソッドです。
上記の一覧では、「特定の URL にアクセスすると、それに対応した Controller のメソッドが呼び出される」
ということが表されています。このアクションメソッドを CRUD に当てはめると下記のようになります。

CRUD	アクション
Create	新規作成(new, create)
Read	データ一覧・個別の表示(index, show)
Update	データの更新(edit, update)
Delete	データの削除(delete)

4.5 Ruby on Rails：Rails 基礎 3

次は、呼び出された Controller のアクションメソッドがどういう働きをしているか見てみましょう。

【users_controller.rb】

今回使用されている Controller は、/app/controllers/の配下にある users_controller.rb です。
この中身を確認してみましょう。

```
class UsersController < ApplicationController
  before_action :set_user, only: [:show, :edit, :update, :destroy]

  # GET /users
  # GET /users.json
  def index
    @users = User.all
  end

  # GET /users/1
  # GET /users/1.json
  def show
  end

  # GET /users/new
  def new
    @user = User.new
  end

  # GET /users/1/edit
  def edit
  end

  # POST /users
  # POST /users.json
  def create
    @user = User.new(user_params)

    respond_to do |format|
      if @user.save
        format.html { redirect_to @user,
        notice: 'User was successfully created.' }
        format.json { render :show, status: :created, location: @user }
      else
        format.html { render :new }
        format.json { render json: @user.errors,
    status: :unprocessable_entity }
      end
    end
  end

  # PATCH/PUT /users/1
  # PATCH/PUT /users/1.json
```

```
  def update
    respond_to do |format|
      if @user.update(user_params)
        format.html { redirect_to @user,
        notice: 'User was successfully updated.' }
        format.json { render :show, status: :ok, location: @user }
      else
        format.html { render :edit }
        format.json { render json: @user.errors,
    status: :unprocessable_entity }
      end
    end
  end

  # DELETE /users/1
  # DELETE /users/1.json
  def destroy
    @user.destroy
    respond_to do |format|
      format.html { redirect_to users_url,
      notice: 'User was successfully destroyed.' }
      format.json { head :no_content }
    end
  end

  private
    # Use callbacks to share common setup or constraints between actions.
    def set_user
      @user = User.find(params[:id])
    end

    # Never trust parameters from the scary internet, only allow the white list
    through.
    def user_params
      params.require(:user).permit(:name, :email)
    end
end
```

scaffold で生成すると、親切にもメソッドに対応する URL をコメントで記載してくれています。
最初にアクセスした時は`/users` だったので、実行されたのは `index` メソッドになります。

メソッドの中身には`@users = User.all` しか書かれていませんが、このメソッドで行っていることは他にもあります。
実はアクションメソッドには、「メソッドの最後に特定の場所にある View を表示する」という特徴があります。
その View とは、「Controller 名と同じ名前のフォルダ内にある、メソッド名と同じ名前の View」のことです。

例えば index メソッドなら、users フォルダ内にある index.html.erb というファイルが表示されることになります。

そして、@users = User.all では何をやっているかというと、
users テーブルのレコードを全て取ってきてインスタンス変数に代入しています。
User が Model に該当し、これに対して all メソッドを使用するとテーブルの全件取得が出来ます。
それをなぜインスタンス変数に代入しているかというと、Controller 内で定義されたインスタンス変数は、
同じ Controller 内のアクションメソッドから表示される View でも参照が出来るようになるからです。

以上をまとめると、呼び出された index メソッドでは、
「/views/users/index.html.erb を users テーブルのレコードを全て取ってきた結果を渡した状態で表示する」
という処理を行っていることになります。

他にも、Controller にはいくつか特徴があります。

【Controller の定義】

確認してみると、Controller はクラスとして定義されていることが分かります。
更に、ApplicationController というクラスを継承していることも分かります。
この ApplicationController は、あらかじめ Controller 用に用意されているクラスで、Controller 全体で共通の処理等を書いておく場所になっています。 この辺りは、前に述べた DRY の理念に則った構造と言えるでしょう。
ちなみに、ApplicationController は ActionController::Base というクラスを継承していますが、実はこの ActionController::Base を継承していることが、Controller として認識される条件になっています。
なので、Controller クラスを定義する際には、直接 ActionController::Base を継承するか、ApplicationController のような、継承しているクラスを更に継承する必要があります。

【StrongParameter】

StrongParameter とは、画面上の操作で送られてきたデータを安全に受け取る仕組みです。 Rails に標準的に組み込まれていて、アプリの作成者が意図していない値を受け取らないように、アプリのプログラム内で受け取ることのできる値に制限を設けます。
上記の Controller 内であれば、以下の部分が該当します。

```
def user_params
  params.require(:user).permit(:name, :email)
end
```

この permit メソッドで指定された値のみが扱えるようになります。
scaffold で生成した場合はこのように自動で追加してくれますが、自分で何か追加した時には permit メソッドを使用して指定しなくてはいけません。

【/views/users/index.html.erb】

今度は実際に表示されている View ファイルの中身を見てみましょう。

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

<h1>Users</h1>

<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Email</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @users.each do |user| %>
      <tr>
        <td><%= user.name %></td>
        <td><%= user.email %></td>
        <td><%= link_to 'Show', user %></td>
        <td><%= link_to 'Edit', edit_user_path(user) %></td>
        <td><%= link_to 'Destroy', user,
        method: :delete, data: { confirm: 'Are you sure?' } %></td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>

<%= link_to 'New User', new_user_path %>
```

HTML の知識がある人なら、このファイルを見た時違和感があったと思います。
上記のファイルには、通常の HTML の構造でいう body の部分しか書かれていません。
それ以外の部分はどこに書かれているかというと、/views/layouts/application.html.erb というファイルに書かれています。

```
<!DOCTYPE html>
<html>
  <head>
    <title>MonkaRailsbasic</title>
    <%= csrf_meta_tags %>
```

```
    <%= stylesheet_link_tag    'application', media: 'all',
    'data-turbolinks-track': 'reload' %>
    <%= javascript_include_tag 'application',
    'data-turbolinks-track': 'reload' %>
  </head>

  <body>
    <%= yield %>
  </body>
</html>
```

表示される時にはこの二つが組み合わさって表示されます。

またこれらの View ファイルには、<%= =>や<= =>を使うことで HTML の中に Ruby のコードが書けるようになっています。
これを利用して、Ruby の繰り返し文を使って HTML の要素を生成する事も出来ます。

【user.rb】

最後に、データの取得や登録時には Model が使用されているのでそちらを確認して見ましょう。

```
class User < ApplicationRecord
end
```

現在はデータに対しての操作が何もないので、Model の中身には何も書かれていません。
実際にアプリを作成していくと、ここにメソッドを定義してデータの操作が出来るようにしていきます。

Model にはいくつか特徴的な機能があります。
せっかくなので、今回のアプリもより WEB アプリらしい形になるように機能を追加して見ましょう。

現在の状態では、登録されるデータに対して制限が何もありません。
例えば、名前が空白のまま登録されたりすると困るので、空白が入力された時は登録出来ないようにしたいですね。

制限をかける時にはバリデーションという機能が使用されます。
実際にバリデーションを使用して、上記の処理が出来るようにして見ましょう。

```
class User < ApplicationRecord
  validates :name, presence: true
end
```

これで、画面上から登録する時に、名前が空白だと登録が出来ずに警告が出るようになっているはずです。

また、もしメールアドレスをログインの ID 代わりにするなら、二つ同じものがあっては困ります。
そういう時には以下のようにしてみましょう。

```
class User < ApplicationRecord
  validates :name, presence: true
  validates :email, uniqueness: true
end
```

これで、一度登録したメールアドレスを使って登録しようとした時には警告が出るようになります。

このように、Model へコードを追加していくことで、色々なデータの操作が出来るようになります。

ここまでで紹介したものが、Rails の基本的な作り方です。
今回のものではまだ簡単な登録だけだったり、デザインも何もついていないので、
次の章からはその辺りも考慮しながら、より WEB アプリらしいものを作成して見ましょう。

4.6 [評価課題]Rails 基礎

ここで、Rails の基礎で学んできたことのポイントを確認してみましょう。

rails コマンドの基本的な使い方

Rails のインストール

```
$ gem (Q1:                  ) rails -v 5.1.3
```

Rails アプリの作成

```
$ rails (Q2:                  ) monka-railsbasic
```

データベースの作成

```
$ rails db:(Q3:                  )
```

モデル、ビュー、コントローラの作成

```
$ rails (Q4:                  ) scaffold User name:string email:string
```

データベースへの反映

```
$ rails db:(Q5:                  )
```

Rails サーバーの起動

```
$ rails (Q6:                  ) -b 0.0.0.0 -p 8080
```

4.6.1 Rails のディレクトリ構成

- (Q7:   )
  アプリケーション本体が格納される

- assets
  WEB ページ内で使用する image ファイルや、ページのレイアウトに使用する css、ページの動きを制御する js 等が格納される

```
- (Q8:                    )
    ユーザアクションを基にアプリを制御するコントローラが格納される
    MVC の"C"

- helpers
    「ヘルパーメソッド」と呼ばれるメソッドをまとめたファイルが格納される

- (Q9:                    )
    データの処理全般を管理するモデルが格納される
    MVC の"M"

- (Q10:                   )
    画面に表示する部分のビューが格納される
    MVC の"V"

- (Q11:                   )
  Rails アプリの設定に関するファイルが格納される

- db
  DB 関係のファイルが格納される

- test
  テスト関係のファイルが格納される

- (Q12:                   )
  アプリで使用する gem をまとめたファイル
```

4.6.2 StrongParameter

StrongParameter とは、画面上の操作で送られてきたデータを安全に(Q13:)仕組みです。Rails に標準的に組み込まれていて、アプリの作成者が(Q14:)を受け取らないように、アプリのプログラム内で受け取ることのできる値に制限を設けます。以下の部分が該当します。

```
def user_params
  params.require(:user).permit(:name, :email)
end
```

この permit メソッドで指定された値のみが(Q15:)ようになります。

第５章　Ruby on Rails デザイン

第５章　Ruby on Rails デザイン

5.1 Ruby on Rails：Gem とは

Rails は様々な Gem が集まったものだというのは、もうご存知だと思います。Gem を使いこなすためには REDME はもちろん、Gem のソースコードを読む必要が出てきます。また、Gem のバージョン管理、依存関係を解消してくれる bundler の存在も欠かせません。簡単に bundler と Gem の構成を確認してみましょう。

5.1.1 bundler について

bundler は、gem のバージョン管理と依存関係を解決した環境を提供してくれるとても便利なツールです。bundler 自体も gem として提供されています。cloud9 で開発を進めてきた方は、すでにインストールされていますので分からなかったかもしれませんが、その他の端末で作業をする場合はインストールが必要です。

```
$ gem install bundle
```

あとは、Gemfile さえ用意すれば bundle install コマンドで Gemfile.lock を作成してくれます。bundler に関して、以下の 2 点は押さえておきましょう。
bundle exec コマンど
新しいブログラミング言語やソフトの環境構築をするとき、PATH が間違っているために動かなかったという経験があるかと思います。Gem に関してその PATH 問題を解決するのが bundle exec です。ターミナル等で rspec を実行するとき、下記のとおりコマンドの前に書いて実行すると、Gemfile に沿ったバージョンの rspec を実行してくれます。

```
$ bundle exec rspec
```

rails コマンドを実行するときも PATH 問題が起きやすいので、気をつけてください。Bundler でインストールした Gem に対するものは、bundle exec で実行するか、シェルの$PATH 設定を追加して確実に意図した Gem が使えるようにしましょう。

Gemfile の書き方

Gemfile には Gem の名前はもちろん、いろいろな情報を設定することができます。以下の項目を参考に、関わっているプロジェクトの Gemfile をチェックしてみましょう。

- バージョン(>=,<,~>)
- 参照先(source,git,path 等)
- グループ設定(development,test,production)

また、Ruby のバージョンを明記しておくこともできます。記入しておくとログにワーニングが出力されて間違いも防げますので、最近は設定しておくプロジェクトが多いです。

そのほかの bundler の機能については、マニュアルを参照ください。シンプルで読みやすい英語ですので、

下記の Reference(Primary Commands,Utilities)は一通り目を通すことをおすすめします。

```
http://bundler.io/docs.html
```

5.1.2 Gem の構成とコードリーディング

Gem の構成

最近は、bundler gem コマンドで Gem を作成するのがデファクトスタンダードになってきましたので、Gem の構成はだいたい以下のとおりです。

```
gem_name/
├.git/
├bin/
├lib/
├test/
├.gitignore
├Gemfile
├LICENSE.txt
├Rakefile
└gem_name.gemspec
```

Gem 自体も Rails アプリと同様に git で管理し、テストも実装されています。特徴としては、Gem のライセンスを示すファイル(上記の場合は LICENSE.txt)とその Gem についての情報がまとめられている*.gemspec があります。Gem を rubygems.org へ公開したとき、この*.gemspec の内容が詳細ページの情報として使用されます。Gem のメインのコードは lib フォルダに保存されています。

コードリーディング

みなさんが Ruby で何かつまづいたとき、どうやって解決しますか?ほとんどの場合は検索したり、Ruby や Rails のマニュアルを読んだりすると思います。これからはもう一歩進んで、コードを読んで解決してみてください。

コードリーディングのメリットは以下のとおりです。

- Ruby や Rails の知識が広がり、理解が深まる
- README やマニュアルなどに記載されていない機能がわかる
- 正確な情報を得られる(ただし、まれに README などが間違っていることもある)
- コード内のコメントのほうが README より分かりやすい場合がある
- 英語で解説されたブログを読まなくても最新情報が得られる

プログラミングを長く続けていると、どうしても慣れた書き方に偏っていきます。知識を広げるにはコードリーディングが一番です。
ここで、ActiveSupport の機能を 1 つ紹介します。紹介するコードは以下のところにあります。初めは Github サイト上で見るのが一番見やすいかもしれません。Ruby のバージョンや ActiveSupport のバージョンは、みなさんが使っているものを参照してください。

- github: https://github.com/rails/rails/blob/master/activesupport/lib/active_support/core_ext/numeric/conversions.rb
- rbenv: ~/.rbenv/versions/2.4.2/lib/ruby/gems/2.4.0/gems/activesupport-5.1.4/lib/active_support/core_ext/numeric/conversions.rb
- rvm: ~/.rvm/gems/ruby-2.4.2/gems/activesupport-5.1.4/lib/active_support/core_ext/numeric/conversions.rb

ファイルを開いてすぐにわかると思いますが、.to_s メソッどに引数が付いています。機能の拡張です。当然、純粋な Ruby にはこのような機能はありません。

```
$ irb
irb(main):001:0> 12345678.to_s(:phone)
TypeError: no implicit conversion of Symbol into Integer
    from (irb):3:in to_s'
    from (irb):3
    from /Users/hogehoge/.rbenv/versions/2.4.2/bin/irb:11:in <main>'
```

```
$ rails console
irb(main):001:0> 12345678.to_s(:phone)
=>"1-234-5678"
```

コードを読み進めていくと、def to_s(format = nil, options = nil)の中で、:phone を指示している部分があります。

```
when :phone
  return ActiveSupport::NumberHelper.number_to_phone(self, options || {})
```

どうも、ヘルパーとして.number_to_phone というものがあるようですね。興味のある方はヘルパーのコードものぞいてみてください。これらを.to_s にまとめた理由まではわかりませんが、Rails ではこういった Ruby メソッドの上書きはよくあります。
さらにファイルの一番下には、Ruby2.4 から Bignum が Fixnum と一緒になった影響で救済措置が書いてあります。仕方なく取ってつけたような感じがするのは私だけでしょうか。

```
# Ruby 2.4+ unifies Fixnum & Bignum into Integer.
if 0.class == Integer
  Integer.prepend ActiveSupport::NumericWithFormat
else
  Fixnum.prepend ActiveSupport::NumericWithFormat
  Bignum.prepend ActiveSupport::NumericWithFormat
End
```

このように、シンプルなファイルを 1 つ見るだけでも新しい発見や疑問が出てくると思います。最後に、呼び出すメソッドがどこにあるかを示すメソッドを紹介しておきますので参考にしてください。

cloud9 の場合

```
$ rails console
2.4.2 :001 > 12345678.method(:to_s).source_location
=> ["/usr/local/rvm/gems/ruby-2.4.0/gems/activesupport-
   5.1.4/lib/active_support/core_ext/numeric/conversions.rb", 102]
```

5.1.3 よく使われる Gem の紹介

以下に業務システムで使いやすい Gem をあげます。基本的には Github に README とコードがありますので、気になるものはどんどん試していってください(順不同)

でバッグ系

- gem ‘pry’ # irb 代替&拡張:ハイライトなどでコーどを見やすくする。プラグインでさらに拡張可能
- gem ‘awesome_print’ # pp 拡張:irb などでコーどを見やすくする
- gem ‘hirb’ # irb 拡張:DB テーブルの内容を ascii テーブルで表示するなど
- gem ‘letter_opener’ # 擬似メール送信:メールサーバーの代わりにブラウザへメール内容を表示
- gem ‘simplecov’ # テストカバレッジ分析

でザイン系

- gem ‘html2slim’ # html から slim への変換
- gem ‘html2haml’ # html から haml への変換
- gem ‘kaminari’ # ページネーション

認証・権限系

- gem ‘devise’ # 認証設定
- gem ‘koala’ # Facebook 向けライブラリ
- gem ‘cancancan’ # 権限設定
- gem ‘pundit’ # ポリシー設定

インフラ系

- gem ‘net-ssh’ # ssh 接続
- gem ‘whenever’ # cron 作成
- gem ‘dotenv’ # 環境変数管理
- gem ‘sidekiq’ # job キュー:ActiveJob のバックエンど

どキュメント系

- gem ‘prawn’ # pdf 出力(Ruby 記述)
- gem ‘pdfkit’ # pdf 出力(html 記述)
- gem ‘thinreports’ # pdf 出力(GUI)
- gem ‘docx_templater’ # docx テンプレート出力
- gem ‘axlsx’ # xlsx 出力
- gem ‘rubyzip’ # zip 出力

その他機能追加

- gem ‘carrierwave’ # ファイルアップローど
- gem ‘simple_form’ # html フォーム作成支援

5.2 Ruby on Rails：デザインテンプレート

デザインテンプレートでは「フロントエンド」について注目していきます。Rails アプリケーション開発に限らず、web アプリケーションの制作でデータベースに近い領域は「バックエンド」、デザインやユーザーエクスペリエンスに近い領域は「フロントエンド」と呼ばれます。web アプリケーションで使われる技術は日進月歩で進化しており、制作は 1 人では出来ない時代になっています。そのため、Rails アプリケーションでは「バックエンド」を担当する Ruby エンジニア、サーバーエンジニアと「フロントエンド」を担当するデザイナーがチームを組んで制作を進めることが多くあります。

多くのエンジニアとデザイナーが共通認識を持って効率よく開発を進めるため、またレスポンシブデザインの対応等今となっては当たり前となった仕組みを取り入れるために「フロントエンド」のライブラリを利用することがあります。今回は Bootstrap を使った「フロントエンド」の開発を体験してみましょう。

http://getbootstrap.com/

5.2.1 Bootstrap インストール

Bootstrap のインストールは gem を使ってインストールしましょう。Gemfile に bootstrap、jquery-rails を追加してください。Bootstrap の JavaScript は jQuery に依存しているので必要です。

```
gem 'bootstrap', '~> 4.0.0.beta'
gem 'jquery-rails'
```

Bootstrap にある CSS を利用するためには次の内容を app/assets/stylesheets/application.scss に追加する必要があります。

```
@import "bootstrap";
```

もし、.scss ファイルがない場合は作成する必要があります。内容は必要ありませんので、echo コマンドで初期化しておきます。

```
$ mv app/assets/stylesheets/application.css
    app/assets/stylesheets/application.scss
$ echo '' > app/assets/stylesheets/application.scss
```

また、Bootstrap にある JavaScript を利用するためには次の内容を app/assets/javascripts/application.js に追加する必要があります。

```
//= require jquery3
//= require popper
//= require bootstrap-sprockets
```

いままで説明した内容は Github にある手順を日本語訳したものです。Rails アプリケーションにインストール場合のみを訳しましたが、その他のインストール方法については URL を参照してください。
https://github.com/twbs/bootstrap-rubygem#a-ruby-on-rails

5.2.2 デザインの適用

では、実際に Bootstrap のデザインを適用してみましょう。ここでは利用者 Model を作成して、簡単にデザインを変更できることを体験しましょう。氏名、電話番号は必須入力とします。

```
$ rails generate scaffold User name:string phone_number:string
  class User < ApplicationRecord
    validates :name, presence: true
    validates :phone_number, presence: true
  end
```

では、実際に Bootstrap が提供しているクラスを指定してみましょう。

- app/views/layouts/application.html.erb

```
  <!DOCTYPE html>
  <html>
    <head>
      <title>MonkaBootstrap</title>
      <%= csrf_meta_tags %>

      <%= stylesheet_link_tag 'application',
          media: 'all', 'data-turbolinks-track': 'reload' %>
      <%= javascript_include_tag 'application',
          'data-turbolinks-track': 'reload' %>
    </head>

    <body>
      <nav class="navbar navbar-light bg-faded">
        <h1 class="navbar-brand mb-0">Navbar</h1>
      </nav>
```

```
    <div class="container">
      <div class="row">
        <div class="col-md-9">
          <%= yield %>
        </div>
        <div class="col-md-3">
          <div>
            <h4>About</h4>
            This system is for user registration.
          </div>
        </div>
      </div>
    </div>
  </body>
</html>
```

- app/views/users/_form.html.erb

```
<%= form_with(model: user, local: true) do |form| %>
  <% if user.errors.any? %>
    <div class="card border-danger">
      <div class="card-header bg-danger text-white">
        <%= pluralize(user.errors.count, "error") %>
        prohibited this user from being saved:
      </div>
      <div class="card-body">
        <ul class="mb-0">
          <% user.errors.full_messages.each do |message| %>
            <li><%= message %></li>
          <% end %>
        </ul>
      </div>
    </div>
  <% end %>

  <div class="form-group">
    <%= form.label :name %>
    <%= form.text_field :name, id: :user_name, class: "form-control" %>
  </div>

  <div class="form-group">
    <%= form.label :phone_number %>
    <%= form.text_field :phone_number, id: :user_phone_number,
    class: "form-control" %>
  </div>

  <div class="actions">
    <%= form.submit class: "btn btn-primary" %>
  </div>
<% end %>
```

- app/views/users/index.html.erb

```
<% if notice.present? %>
  <div class="alert alert-success" role="alert">
    <%= notice %>
  </div>
<% end %>

<h1>Users</h1>

<table class="table table-striped">
  <thead>
    <tr>
      <th>Name</th>
      <th>Phone number</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @users.each do |user| %>
      <tr>
        <td><%= user.name %></td>
        <td><%= user.phone_number %></td>
        <td><%= link_to 'Show', user %></td>
        <td><%= link_to 'Edit', edit_user_path(user) %></td>
        <td><%= link_to 'Destroy', user, method: :delete,
        data: { confirm: 'Are you sure?' } %></td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>


<%= link_to 'New User', new_user_path %>
```

- app/views/users/show.html.erb

```
<% if notice.present? %>
  <div class="alert alert-success" role="alert">
    <%= notice %>
  </div>
<% end %>

<div class="row">
  <div class="col-md-6">
    <span class="text-muted">Name:</span><%= @user.name %>
  </div>
  <div class="col-md-6">
    <span class="text-muted">Phone number:</span><%= @user.phone_number %>
  </div>
</div>

<%= link_to 'Edit', edit_user_path(@user) %> |
<%= link_to 'Back', users_path %>
```

デザインは期待通りに変わってくれたでしょうか？スクリーンショットを載せるので参考にしてくださいね。この他にもたくさんの機能を Bootstrap は提供してくれます。サンプルも充実しているので参考にしてくださいね。

http://getbootstrap.com/docs/4.0/examples/

- 利用者一覧

Navbar

Users

Name	Phone number

New User

About

This system is for user registration.

- 入力画面

Navbar

New User

2 errors prohibited this user from being saved:

- Name can't be blank
- Phone number can't be blank

Name

Phone number

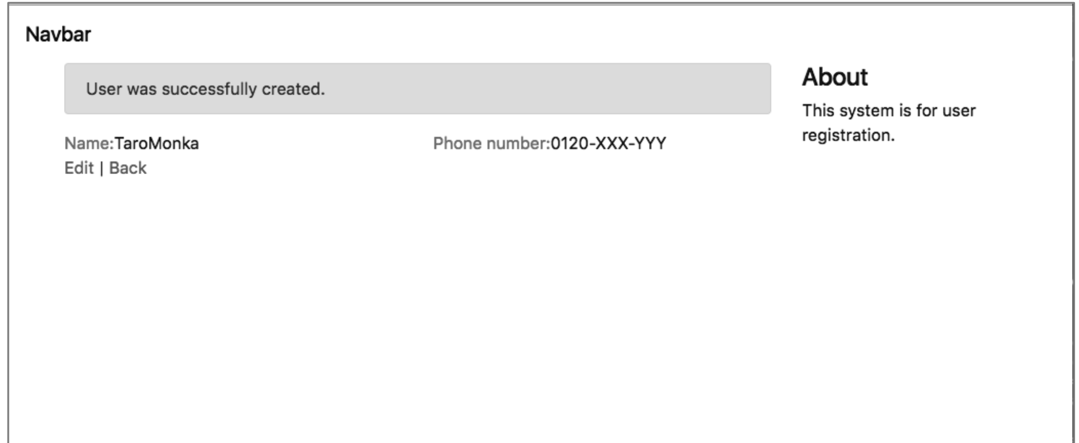
Create User

Back

About

This system is for user registration.

- 詳細画面



5.2.3 国際化対応（I18n)

Bootstrap を利用した開発はいかがだったでしょうか？Ruby エンジニアだけでも見栄えのする web アプリケーションができました。実際の現場ではデザインはもちろんのこと、「利用する」、「使用する」等の表現のばらつきの統一や、海外へのサービス展開による英語化といった文言を変更する必要があります。Rails アプリケーションでは I18n という gem を使って国際化対応をすることができます。

I18n を使うと I18n.translate メソッドを使った YAML ファイルに定義した内容を参照することができます。具体的な設定は後に解説しますが、まずは結果は次のような内容になります。

- app/views/users/show.html.erb

```
<% if notice.present? %>
  <div class="alert alert-success" role="alert">
    <%= notice %>
  </div>
<% end %>

<div class="row">
  <div class="col-md-6">
    <span class="text-muted"><%= User.human_attribute_name('name') %>:</span>
    <%= @user.name %>
  </div>
  <div class="col-md-6">
    <span class="text-muted"><%=
  User.human_attribute_name('phone_number') %>:
    </span>
    <%= @user.phone_number %>
  </div>
</div>
```

```
<%= link_to t('.edit'), edit_user_path(@user) %> |
<%= link_to t('.back'), users_path %>
```

link_to メソッドの引数に Edit とありましたが、それは t('.edit')と置き換えることができます。メソッドから文言を参照することで YAML ファイルだけを変更することで簡単に変更することができます。t は I18n.translate の省略形で、同じ機能を提供してくれます。カラム名は User.human_attribute_name('name')で参照することができます。
では、さっそく設定を進めていきましょう。I18n の設定する項目は 2 つあります。 1. Rails アプリケーションの言語設定を日本語にする 1. YAML ファイルをダウンロードする

言語設定の日本語化

- config/initializers/locale.rb

```
Rails.application.config.i18n.default_locale = :ja
```

YAML ファイルダウンロード

日本語向けの YAML ファイルが提供されているのでそれを利用しましょう。

```
$ curl -o config/locales/ja.yml -L https://raw.github.com/svenfuchs/rails-
    i18n/master/rails/locale/ja.yml
```

ダウンロードした YAML ファイルに利用者についての情報を追加しましょう。YAML ファイルはインデントが重要になるので先頭から内容を表示していますが、同じ内容があれば追加する必要はありません。

```
ja:
  activerecord:
    errors:
      messages:
        record_invalid: "バリデーションに失敗しました: %{errors}"
        restrict_dependent_destroy:
          has_one: "%{record}が存在しているので削除できません"
          has_many: "%{record}が存在しているので削除できません"
    models:
      user: 利用者
    attributes:
      user:
        name: 名前
```

```
      phone_number: 電話番号
users:
  default: &default
    new: 新規作成
    edit: 編集
    show: 詳細
    destroy: 削除
    back: 戻る
    confirm: 本当に削除しますか？
  index:
    <<: *default
    title: 一覧画面
  show:
    <<: *default
  new:
    <<: *default
    title: 新規画面
  edit:
    <<: *default
    title: 編集画面
  create:
    success: 利用者を新規作成しました
  update:
    success: 利用者を更新しました
  destroy:
    success: 利用者を削除しました
```

I18n を利用してみましょう。変更する箇所は次を参照してください。

- app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  before_action :set_user, only: [:show, :edit, :update, :destroy]

  # GET /users
  # GET /users.json
  def index
    @users = User.all
  end

  # GET /users/1
  # GET /users/1.json
  def show
  end

  # GET /users/new
  def new
    @user = User.new
```

```
  end

  # GET /users/1/edit
  def edit
  end

  # POST /users
  # POST /users.json
  def create
    @user = User.new(user_params)

    respond_to do |format|
      if @user.save
        format.html { redirect_to @user, notice: t('.success') }
        format.json { render :show, status: :created, location: @user }
      else
        format.html { render :new }
        format.json { render json: @user.errors,
          status: :unprocessable_entity }
      end
    end
  end

  # PATCH/PUT /users/1
  # PATCH/PUT /users/1.json
  def update
    respond_to do |format|
      if @user.update(user_params)
        format.html { redirect_to @user, notice: t('.success') }
        format.json { render :show, status: :ok, location: @user }
      else
        format.html { render :edit }
        format.json { render json: @user.errors,
          status: :unprocessable_entity }
      end
    end
  end

  # DELETE /users/1
  # DELETE /users/1.json
  def destroy
    @user.destroy
    respond_to do |format|
      format.html { redirect_to users_url, notice: t('.success') }
      format.json { head :no_content }
    end
  end

  private
    # Use callbacks to share common setup or constraints between actions.
    def set_user
      @user = User.find(params[:id])
    end

    # Never trust parameters from the scary internet, only allow the white
list through.
```

```
    def user_params
      params.require(:user).permit(:name, :phone_number)
    end
end
```

- app/views/users/index.html.erb

```
<% if notice.present? %>
  <div class="alert alert-success" role="alert">
    <%= notice %>
  </div>
<% end %>

<h1><%= t('.title') %></h1>

<table class="table table-striped">
  <thead>
    <tr>
      <th><%= User.human_attribute_name('name') %></th>
      <th><%= User.human_attribute_name('phone_number') %></th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @users.each do |user| %>
      <tr>
        <td><%= user.name %></td>
        <td><%= user.phone_number %></td>
        <td><%= link_to t('.show'), user %></td>
        <td><%= link_to t('.edit'), edit_user_path(user) %></td>
        <td><%= link_to t('.destroy'), user, method: :delete,
        data: { confirm: t('.confirm') } %></td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>
<%= link_to t('.new'), new_user_path %>
```

- app/views/users/show.html.erb

```
<% if notice.present? %>
  <div class="alert alert-success" role="alert">
    <%= notice %>
  </div>
<% end %>

<div class="row">
  <div class="col-md-6">
    <span class="text-muted"><%= User.human_attribute_name('name') %>:
    </span><%= @user.name %>
```

```
  </div>
  <div class="col-md-6">
    <span class="text-muted"><%=
  User.human_attribute_name('phone_number') %>:
    </span><%= @user.phone_number %>
  </div>
</div>

<%= link_to t('.edit'), edit_user_path(@user) %> |
<%= link_to t('.back'), users_path %>
```

- app/views/users/edit.html.erb

```
<h1><%= t('.title') %></h1>

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

<%= link_to t('.show'), @user %> |
<%= link_to t('.back'), users_path %>
```

- app/views/users/new.html.erb

```
<h1><%= t('.title') %></h1>

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

<%= link_to t('.back'), users_path %>
```

- app/views/users/_form.html.erb

```
<%= form_with(model: user, local: true) do |form| %>
  <% if user.errors.any? %>
    <div class="card border-danger">
      <div class="card-header bg-danger text-white">
        <%= t('errors.template.header.other',
        model: User.model_name.human, count: user.errors.size) %>
      </div>
      <div class="card-body">
        <ul class="mb-0">
          <% user.errors.full_messages.each do |message| %>
            <li><%= message %></li>
          <% end %>
        </ul>
      </div>
    </div>
  <% end %>

  <div class="form-group">
    <%= form.label :name %>
    <%= form.text_field :name, id: :user_name, class: "form-control" %>
```

```
  </div>

  <div class="form-group">
    <%= form.label :phone_number %>
    <%= form.text_field :phone_number, id: :user_phone_number,
    class: "form-control" %>
  </div>

  <div class="actions">
    <%= form.submit class: "btn btn-primary" %>
  </div>
<% end %>
```

第６章　Ruby on Rails アジャイル開発

第６章　Ruby on Rails アジャイル開発

ここからは、EC サイトアプリケーションの開発を例に各実装を解説していきます。 例題で各実装について解説し、問題を解くことにより EC サイトが少しずつ完成していきます。

6.1 Ruby on Rails：データベース設計、多対多の関連付け

まず、この章では、EC サイトのデータベース設計の解説を行います。 設計のために、ActiveRecord、データベース設計の基礎、多対多の関連付けについて簡単に説明します。

6.1.1 ActiveRecord とは

Active Record とは、Rails に付属する、重要なライブラリの 1 つで、MVC の M(モデル)に相当します。 Active Record は、ORM (オブジェクトリレーショナルマッピング) で実装されています。ORM とは、簡潔に説明すると、アプリケーションが持つオブジェクトとリレーショナルデータベース(RDBMS)を繋ぐプログラミング技法です。
また、ActiveRecord では、下記の仕組みが特に重要となっています。

- モデルとそのデータを表す仕組み
- モデル間の関連性を表す仕組み
- 関連するモデルを通した階層の継承を表す仕組み
- DB に保存する前に検証する仕組み
- オブジェクト指向の手法で DB 操作を実行する仕組み

6.1.2 データベース設計の基礎

データベース設計はとても大事で難しい部分もあります。それだけで本が 1 冊書けるぐらいです。ここでは詳しく解説しませんが、データベースの構成次第で、あとの機能の実装がやりにくかったり複雑になってしまうので、くれぐれも注意してください。できるかぎり、サイトで扱うもの、関連のある人、サイトの使われ方などを具体的にイメージしながら考えていきましょう。
データベース設計で決めることは次の 4 つです。

- 作業 1：アプリで扱うデータをモレなく書き出す
- 作業 2：データの正規化（グループ分け）をする
- 作業 3：各データの型（データの種類）を決める
- 作業 4：各データのフィールド名（アルファベット）を決める

ここにあげたものは、必ずしも順番にする必要はありません。いろいろ考えていくうちに他に必要なデータが見えてきたり、「こっちの名前がこうなら、あっちの名前はこうしよう」 ということが出てきます。 サイトを利用する場面や利用する人のことを想像しながら、納得するまで考えます。可能であれば、他のプログラミング経験者の意見を聞くことをおすすめします。

6.1.3 多対多の関連付け

まず、多対多の関連付けとは、お互いのテーブルのレコード同士が複数の相手側レコードと関連付けられる関係の事です。
Rails で多対多の関連付けをする方法は以下の 2 通りがあります。

- has_many :through での関連付け

2 つのモデルの間に、互いのモデルの ID を保持している中間テーブル(第 3 のモデル)を作成して、紐付けします。 例えば、次の[3.3-例題]のように、先生と授業の関係を表します。実際にどのように実装するかは例題を見ていきましょう。

- has_and_belongs_to_many での関連付け

has_many :through ではなく、has_and_belongs_to_many でも多対多の関連付けが可能ですが、 今回は、has_many :through での関連付けで実装していきますので詳細な説明は省きます。 has_and_belongs_to_many は has_many :through と仕様が異なっているため、詳しい仕様が気になる人は調べてみましょう。

6.1.4 ActiveRecord の代表的なメソッド

ここでは、以下のモデルを例に ActiveRecord の代表的なメソッドを簡潔に説明していきます。

モデル User:ユーザー

field 名	名称	型
id	ID	integer
name	名前	string
mail_address	メールアドレス	string

ActiveRecord の代表的なメソッド一覧

※表の発行 SQL は、わかりやすさのため、テーブル名は省略してカラム名のみ記述しています。

メソッド	説明	使用例	発行 SQL
all	全件取得(全カラム)	User.all	SELECT * FROM users
select	全件取得(カラム指定)	User.select(:name)User.select('name,mail_address')	SELECT name FROM usersSELECT name,mail_address FROM users
find	検索(id 指定)	User.find(1)	SELECT * FROM users WHERE id = 1
find_by	検索(条件指定)	User.find_by(id:1)User.find_by('id > 1')	SELECT * FROM users WHERE id = 1 LIMIT 1SELECT * FROM users WHERE id > 1 LIMIT 1
where	検索(条件指定)	User.where(id:1)User.where('id > 1')	SELECT * FROM users WHERE id = 1SELECT * FROM users WHERE id > 1
first	最初のデータをとる	User.firstUser.first(2)	SELECT * FROM users ORDER BY id ASC LIMIT 1SELECT * FROM users ORDER BY id ASC LIMIT 2
last	最後のデータをとる	User.lastUser.last(2)	SELECT * FROM users ORDER BY id DESC LIMIT 1SELECT * FROM users ORDER BY id DESC LIMIT 2
order	ソート	User.order(:name)User.order(name: :DESC)	SELECT * FROM users ORDER BY name ASCSELECT * FROM users ORDER BY name DESC
limit	制限	User.limit(2)	SELECT * FROM users LIMIT 2

各メソッドの詳細

- all

レコードを全件取得します

- select

カラムを指定し、レコードを取得します。引数の値がカラムとなります。

- find

指定した id のレコードを取得します。引数の値が指定する id となります。

find は、該当するデータが見つからない場合は例外（RecordNotFound）が発生します。

- find_by

特定のカラムの条件を指定し、該当する 1 件を取得します。引数の値が条件となります。

find_by は該当するデータが見つからない場合は、nil を返します。

- where

特定のカラムの条件を指定し、該当する全件を取得します。引数の値が条件となります。
where は、該当するデータが見つからない場合は空の ActiveRecord::Relation を返します。

- first

レコードの最初の 1 件を取得します。引数を渡すと最初の n 件と指定することもできます。

- last

レコードの最後の 1 件を取得します。引数を渡すと最後の n 件と指定することもできます。

- order

レコードを引数に指定したカラムで並び変えます。デフォルトの並び順は ASC(昇順)になっています。
降順で並び変える場合は User.order(name: :DESC)とします。

- limit

特定のレコード件数を取得します。引数の値が最大取得行数となります。

etc

- ActiveRecord::Relation について

ActiveRecord::Relation とは、モデルオブジェクトのコレクションです。

ActiveRecord::Relation を返す場合は、メソッドチェーンが可能ですので、

例えば where に続いてさらに ActiveRecord のメソッドを使用する事ができます。
ここでは詳細を説明しませんので、気になる場合は調べてみましょう。

- ?の使い方

条件に変数を使用したい場合等に、?(プレースホルダ)を使用します。

変数でなくても、直接値を指定することも可能です。(冗長ですが例も記載しています。)

```
name = "test"
User.where("name = ?", name)
#SELECT  "users".* FROM "users" WHERE (name = 'test')

User.where("name = ?", 'test2')
#SELECT  "users".* FROM "users" WHERE (name = 'test2')
```

6.1.5 例題

先生と授業(科目)のテーブルを作成し、先生と授業の多対多で関連付けしていきます。先生は複数の授業(科目)を担当し、授業(科目)からも複数の先生が受け持っているという多対多の関連付けとして実装します。 データベース構成は以下の通りとします。

Teacher : 先生 テーブル

field 名	名称	型
name	名前	string

Lesson : 授業 テーブル

field 名	名称	型
name	授業名	string

Appointment : 出席者 テーブル（中間テーブル）

field 名	名称	型
teacher_id	先生	references
lesson_id	授業	references

TeacherAppointment domain model

Lesson
created_at datetime
id integer PK
name string
updated_at datetime

Teacher
created_at datetime
id integer PK
name string
updated_at datetime

Appointment
created_at datetime
id integer PK
lesson_id integer FK
teacher_id integer FK
updated_at datetime

① アプリケーションの作成

まずは、例題用の Rails アプリケーションを作成します。

```
$ rails new teacher_sample
```

② Teacher モデルの作成
次に必要なモデルを作成してきます。

```
$ rails generate model Teacher name:string
Running via Spring preloader in process 1718
      invoke  active_record
      create    db/migrate/20170828083314_create_teachers.rb
      create    app/models/teacher.rb
      invoke    test_unit
      create      test/models/teacher_test.rb
      create      test/fixtures/teachers.yml
```

③ Lesson モデルの作成

```
$ rails generate model Lesson name:string
Running via Spring preloader in process 1754
      invoke  active_record
      create    db/migrate/20170828083408_create_lessons.rb
      create    app/models/lesson.rb
      invoke    test_unit
      create      test/models/lesson_test.rb
      create      test/fixtures/lessons.yml
```

③ Appointment モデルの作成

中間テーブルとなるモデルです。 teacher と lesson を参照するように設定して、モデルを生成します。

```
$ rails generate model appointment teacher:references lesson:references
lesson:references
Running via Spring preloader in process 1786
      invoke  active_record
      create    db/migrate/20170828083440_create_appointments.rb
      create    app/models/appointment.rb
      invoke    test_unit
      create      test/models/appointment_test.rb
      create      test/fixtures/appointment.yml
```

teacher、lesson と appointment のモデルが作成できたので、テーブルを作成するためにマイグレーションも実行しましょう。

```
$ rails db:migrate
== 20170828083314 CreateTeachers: migrating ===============================
-- create_table(:teachers)
   -> 0.0014s
== 20170828083314 CreateTeachers: migrated (0.0015s) ======================

== 20170828083408 CreateLessons: migrating ================================
-- create_table(:lessons)
   -> 0.0011s
== 20170828083408 CreateLessons: migrated (0.0012s) =======================

== 20170828083440 CreateAppointments: migrating ===========================
-- create_table(:appointment)
   -> 0.0036s
== 20170828083440 CreateAppointments: migrated (0.0037s) ==================
```

各モデルの関係を設定するために、以下の内容を追記してください。

has_many :through は、appointment をショートカットして、teacher もしくは lesson を参照できるようにします。

app/models/teacher.rb

```
class Teacher < ApplicationRecord
  has_many :appointments
  has_many :lessons,through: :appointments
end
```

app/models/lesson.rb

```
class Lesson < ApplicationRecord
  has_many :appointments
  has_many :teachers,through: :appointments
end
```

appointment モデルは teacher と lesson を参照するように生成したため、既に下記のソースコードとなっています。

app/models/appointment.rb

```
class Appointment < ApplicationRecord
  belongs_to :teacher
  belongs_to :lesson
end
```

6.1.6 問題

今回の EC サイトのデータベース構成は、以下のようにします。

この構成通りに、モデルを作成してみましょう。本と商品は多対多の関係になることに注意して下さい。

Book : 本 テーブル

field 名	名称	型
title	タイトル	string
author	著者	string
published_on	出版日	date
showing	商品表示	boolean
price	価格	integer

Tag : 商品タグ テーブル

field 名	名称	型
name	タグ名	string

Tagging : タグ付け テーブル

field 名	名称	型
book_id	本	references
tag_id	商品タグ	references

6.2 Ruby on Rails: ActiveRecord の応用

ここでは ActiveRecord の応用的な利用方法「N+1 問題について」「サブクエリの作り方」について学習します。

6.2.1 「N+1 問題」とは

一覧画面などに表示するデータを取得するとき、SQL のクエリが「データ量(N) + 1」発行され、データ量が多くなるにつれてパフォーマンスを低下させてしまう問題です。例えばデータの数だけ SELECT 文を発行すると 1 回が 1ms でも、データの数が増えるとその数だけ SELECT 文を発行することになり、パフォーマンスが悪くなっていきます。

では、N+1 問題について具体的なコードを見ながら考えてみましょう。
Model は User と Favorite のものがあったとします。

User: ユーザーテーブル

field 名	名称	型
id	ID	integer
name	名前	string

Favorite: お気に入りテーブル

field 名	名称	型
id	ID	integer
user_id	ユーザーID	integer
title	タイトル	string

ユーザー(User)はたくさんのお気に入り(Favorite)を登録することができるという「1 対多」の関連付をします。

app/models/user.rb

```
class Favorite < ApplicationRecord
  has_many :favorites
```

```
end
app/models/favorite.rb
class Favorite < ApplicationRecord
  belongs_to :user
end
```

Controller では Favorite の一覧を取得するよう以下の内容になっているとします。

```
class Favorite < ApplicationController
  def index
    @favorites = Favorite.order(:id)
  end
end
```

View では取得したお気に入りに関連する user の名前を表示するとします。

```
<h1>お気に入り一覧</h1>
  <% @favorites.each do |fav| %>
    <div><%= fav.user.name %></div>
  <% end %>
```

「お気に入り一覧を表示する」という機能的には問題なく満たしていますが、レコードを取得する際のデータベースへのアクセスに問題があります。

ログを確認してみましょう。

ログ

users テーブルへの SELECT が favorite の数だけ行われています。 これではレコードが増えれば増えるほど SQL のクエリが発行されてしまいパフォーマンスが非常に悪くなってしまいます。 例えば 1 回のクエリにかかる時間が 0.1ms 前後だったとして、取得するレコードが何千何万とあった場合それだけ時間がかかってしまうということになります。これが「N+1 問題」の実態です。

```
Favorite Load (0.5ms)  SELECT "favorites".* FROM "favorites" ORDER BY
    "favorites"."id" ASC
  User Load (0.2ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ?
    LIMIT ?  [["id", 1], ["LIMIT", 1]]
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
    = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
  CACHE User Load (0.1ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
    = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
    = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
    = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
```

```
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.1ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 1], ["LIMIT", 1]]
User Load (0.1ms)  SELECT  "users".* FROM "users" WHERE "users"."id" = ?
  LIMIT ?  [["id", 2], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
  = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
```

```
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
    = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
    = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
    = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
    = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
    = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
    = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
  CACHE User Load (0.0ms)  SELECT  "users".* FROM "users" WHERE "users"."id"
    = ? LIMIT ?  [["id", 2], ["LIMIT", 1]]
```

6.2.2 「N+1 問題」解消方法

解決方法としては以下の 4 つのメソッドを使用するのが一般的です。 メソッドは組み合わせて使うこともできます。

(a) includes を使用する

引数には、Association 先を指定します。

```
def index
  @favorites = Favorite.order(:id).includes(:user)
end
```

するとクエリの実行は 2 回で済みます。 1 回目でお気に入りを全件取得し、2 回目で user_id を指定して紐づく user を取得しています。

```
Favorite Load (1.0ms)  SELECT "favorites".* FROM "favorites" ORDER BY
    "favorites"."id" ASC
User Load (0.1ms)  SELECT "users".* FROM "users" WHERE "users"."id" IN (1, 2,
    3, 4, 5, 6, 7, 8, 9)
```

(b) preload を使用する

```
def index
  @favorites = Favorite.preload(:user).order(:id)
end
```

```
Favorite Load (11.2ms)  SELECT "favorites".* FROM "favorites" ORDER BY
    "favorites"."id" ASC
User Load (0.3ms)  SELECT "users".* FROM "users" WHERE "users"."id" IN (1, 2,
    3, 4, 5, 6, 7, 8, 9)
```

(c) eager_load を使用する

```
def index
  @favorites = Favorite.eager_load(:user).order(:id)
end
```

引数で指定した Association を LEFT OUTER JOIN(左外部結合)します。 1 回のクエリで済みます。

```
SELECT
    "favorites"."id" AS t0_r0,
    "favorites"."user_id" AS t0_r1,
    "favorites"."title" AS t0_r2,
    "favorites"."created_at" AS t0_r3,
    "favorites"."updated_at" AS t0_r4,
    "users"."id" AS t1_r0,
    "users"."name" AS t1_r1,
    "users"."created_at" AS t1_r2,
    "users"."updated_at" AS t1_r3
  FROM
    "favorites"
    LEFT OUTER JOIN "users"
      ON "users"."id" = "favorites"."user_id"
 ORDER BY
    "favorites"."id" ASC
```

(d) joins を使用する

joins は、指定した Association を INNER JOIN します。 これだけでは「N+1 問題」の解消はできませんが、結合先のテーブルに対しての絞り込みが可能です。

```
def index
  @favorites = Favorite.joins(:user).where(users: {id: 1})
End
```

```
Favorite Load (0.2ms)  SELECT "favorites".* FROM "favorites" INNER JOIN "users"
   ON "users"."id" = "favorites"."user_id" WHERE "users"."id" = ?  [["id", 1]]
```

6.2.3 サブクエリの作り方

User テーブルから抽出する場合

時間がかかるクエリ(SQL 構文)

user テーブルのデータを複数取得したい場合、id 毎にクエリを発行すると、データベースへのアクセス回数が増えパフォーマンスが悪くなってしまいます。

```
SELECT * FROM users WHERE id = 1;
```

```
SELECT * FROM users WHERE id = 2;
SELECT * FROM users WHERE id = 3;
.
.
.
```

IN を使用したクエリ(SQL 構文)

user テーブルから複数のデータを取得したい場合 IN 句を使用すれば発行するクエリは 1 回で済みます。

```
SELECT * FROM users WHERE id IN (1, 2, 3);
```

Rails で書いた場合

rails で user テーブルから 1 度に複数のデータを取得する場合を見てみましょう。
配列で指定する

```
User.where(id: [1, 2, 3])
```

ids メソッドを使用し同じことができます

```
User.where(id: User.ids)
```

どのような SQL クエリが実行されているのか to_sql メソッドを使用して確認してみましょう。

```
User.where(id: [1, 2, 3]).to_sql

# => 'SELECT "users".* FROM "users" WHERE "users"."id" IN (1, 2, 3)'
```

複数条件で絞り込みを行う

`where` メソッドをチェーンしサブクエリを発行します。

```
Favorite.where(title: "title1").where(user: User.where(name: "name2"))
```

`to_sql` メソッドで確認してみましょう。
`favorites` テーブルに対してサブクエリを含む 2 つの条件(AND 条件)で絞り込みを行うクエリが発行されているのが確認できると思います。

```
SELECT "favorites".*
  FROM "favorites"
 WHERE "favorites"."title" = 'title1'
   AND "favorites"."user_id" IN (
       SELECT "users"."id" FROM "users" WHERE "users"."name" = 'name2'
   )
```

or メソッドを使用し、OR 条件で絞り込みを行う。
Rails5 系から or メソッドが実装されました。

```
User.where(id: 1).or(User.where(name: "user2"))
```

to_sql メソッドで確認してみましょう。
users テーブルに対して 2 つの条件(OR 条件)で絞り込みを行うクエリが発行されているのが確認できると思います。

```
SELECT "users".*
  FROM "users"
 WHERE (
   "users"."id" = 1 OR
   "users"."name" = 'user2'
 )
```

not メソッドを使い否定する。
not メソッドを使用し where メソッドなどで指定した条件を否定することができます。

```
User.where.not(id: 1)
```

to_sql メソッドで確認すると「id が 1 ではない("id" != 1")」というクエリが発行されているのがわかります。

```
SELECT "users".*
  FROM "users"
 WHERE "users"."id" != 1
```

6.3 プロダクトバックログを見積もる

ECサイトの開発

～プロダクトバックログを見積る～

プランニングポーカーを使った相対見積

1．　基準となるバックログを決める

- 大きすぎず、小さすぎないバックログを１つ決める
- そのバックログを2ptか3ptとしてください

2．　他のプロダクトバックログを見積る
- 見積もるプロダクトバックログを１つ選ぶ
- 基準となるプロダクトバックログと比較して
どのぐらいの作業量になるかを感覚的に見積り、
その数字が書かれたカードを選ぶ
（数字が他の人に見えないように額の前にかざす）
- 全員が選び終えたら一斉にカードを表に向ける
- 数字が一致している場合は見積り終了
- 数字が異なっていたら、一番大きな数字を出した人と
一番小さな数字を出した人から理由を聞き、再度見積る
- 再見積りは2回まで（合計３回まで）で終える
（最後まで数字が合わなかった場合は大きい値を選ぶ）

6.4 スプリントバックログの作成

ECサイト開発

～スプリントバックログの作成～

1．　プロダクトバックログに書かれた機能を満たすために必要なタスクを洗い出す

２． 個々のタスクについて
何時間ぐらいあれば完了できるか
見積る

- 学生症候群を防ぐためバッファは極力積まない
- 最大でも１日ぐらいの粒度にする
- 大きすぎるものは意味のある単位で分割する

第７章　Ruby on Rails テスト

第７章　Ruby on Rails テスト

7.1 Ruby on Rails：Rails テスト基礎 1

7.1.1 テスティングツールについて

この章では、Rails アプリケーションでのテストの実装方法について学習します。Rails の標準テスティングツールは Minitest::Test です。rails new コマンドで作成したアプリケーションの雛形を作成したときには、自動的に test フォルダが作られています。

Minitest::Test の他に RSpec というテスティングツールもあります。どちらが優れているということはありません。RSpec は DSL（ドメイン固有言語）を提供しているので、いつも馴染みのある Ruby の文法とは異なる文法を学習する必要があります。RSpec の例を挙げると次のようなソースコードになります。

```
require 'rails_helper'

RSpec.describe User, type: :model do
  it "is valid with name and phone number" do
    user = build(:user)
    expect(user).to be_valid
  end

  it "is invalid without name" do
    user = build(:user, name: nil)
    user.valid?
    expect(user.errors[:name]).to include("can't be blank")
  end

  it "is invalid without phone number" do
    user = build(:user, phone_number: nil)
    user.valid?
    expect(user.errors[:phone_number]).to include("can't be blank")
  end
end
```

it、expect などいつも見ないメソッドに驚かれたのではないでしょうか。筆者も当時はかなり驚いて RSpec での実装は億劫でした。ですが、DSL を利用することで書き方についてのルールを決定することができ、チームでの作業効率が格段とあがります。みな

さんも多くのエンジニアと共に今後、開発していくことを考えて今回は RSpec を利用してテストを実装していきます。

7.1.2 RSpec の環境構築

必要な gem のインストール

RSpec を使ったテストの実装環境を整えましょう。RSpec は強力なテスティングツールですが、テストデータの作成には長けていません。もちろん、腕に自信のある人は自作しても良いですが、OSS の文化では「車輪の再発明」は避ける習慣があります。今回は `rspec-rails`、`factory_bot_rails`、`gimei` の 3 つを利用して実装します。

- rspec-rails
  RSpec を Rails で利用するための gem
- factory_bot_rails
  テストデータ作成の補助をする gem
- gimei
  ランダムに日本人の名前を生成する gem

Gemfile にこれらの gem を追加してみましょう。テスティングツールは本番環境では利用しないので Gemfile のグループは `development` と `test` とします。

```
  group :development, :test do
    # Use RSpec
    gem 'rspec-rails', '~> 3.6'
    # Use FactoryBot
    gem 'factory_bot_rails'
    # Use gimei for generating a Japanese fake name
    gem 'gimei'
  end

bundle install
```

RSpec 実行環境の初期化

RSpec を実行するにあたって、様々な設定が必要になります。複雑になるので、順を追ってゆっくり説明するので安心してくださいね。

RSpec 設定ファイルの生成

設定ファイルを作成するために次のコマンドを実行してください。いくつかのフォルダ、ファイルが生成されます。

```
$ rails generate rspec:install
Running via Spring preloader in process 54405
      create  .rspec
      create  spec
      create  spec/spec_helper.rb
      create  spec/rails_helper.rb
```

生成されたファイルに次の内容を追加してください。

- .rspec

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

spec/rails_helper.rb

```
RSpec.configure do |config|
    # Simplify syntax
    config.include FactoryBot::Syntax::Methods
end
```

Rails 設定ファイルの修正

Minitest::Test が Rails では標準のテスティングツールなので RSpec を利用するようにしてあげる必要があります。

- config/application.rb

```
module RspecMockups
  class Application < Rails::Application
    # Don't generate system test files.
    config.generators.system_tests = nil
  end
end
```

- config/initializers/generators.rb（新規作成）

```
Rails.application.config.generators do |g|
  g.test_framework :rspec
  g.view_specs false
  g.routing_specs false
  g.helper_specs false
  g.fixture_replacement :factory_bot, dir: 'spec/factories'
end
```

test フォルダの削除

前章で説明があったとおり、test フォルダは Minitest::Test に関連するファイルがありますが今回は RSpec を利用してテストを実装しますので、削除します。

```
$ rm -r test
```

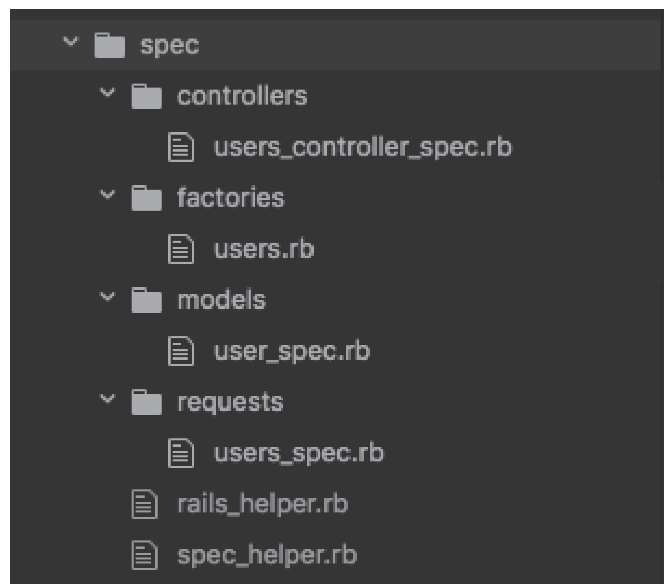
7.2 Ruby on Rails：Rails テスト基礎 2

7.2.1 RSpec のテンプレート作成

ここまでの長い道のり、お疲れ様でした。ここからやっと RSpec の実装を始めることができます。ここでは例題として利用者の氏名と電話番号を登録できる Rails アプリケーションを作成しましょう。

```
$ rails generate scaffold User name:string phone_number:string
Running via Spring preloader in process 55295
      invoke  active_record
      create    db/migrate/20170828063211_create_users.rb
      create    app/models/user.rb
      invoke    rspec
      create      spec/models/user_spec.rb
      invoke      factory_bot
      create        spec/factories/users.rb
      invoke  resource_route
       route    resources :users
      invoke  scaffold_controller
      create    app/controllers/users_controller.rb
      invoke    erb
      create      app/views/users
      create      app/views/users/index.html.erb
      create      app/views/users/edit.html.erb
      create      app/views/users/show.html.erb
      create      app/views/users/new.html.erb
      create      app/views/users/_form.html.erb
      invoke    rspec
      create      spec/controllers/users_controller_spec.rb
      invoke      rspec
      create        spec/requests/users_spec.rb
      invoke    helper
      create      app/helpers/users_helper.rb
      invoke      rspec
      invoke    jbuilder
      create      app/views/users/index.json.jbuilder
      create      app/views/users/show.json.jbuilder
      create      app/views/users/_user.json.jbuilder
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/users.coffee
      invoke    scss
      create      app/assets/stylesheets/users.scss
      invoke  scss
      create    app/assets/stylesheets/scaffolds.scss
```

spec フォルダに新しくファイルが作成できたでしょうか。Rails は MVC パターンでプログラムを作成するので Model、View、Controller のそれぞれにテストを実装します。テストで実装すべき観点は次の通りそれぞれ異なります。



- controllers/users_controller_spec.rb

Controller への単体テストを観点に実装する。HTTP ステータスコードが期待通りであるか、メソッド実行後にデータベースへの操作が期待通りであるかをテストする。

- models/user_spec.rb

Model への単体テストを観点に実装する。仕様書通りにバリデータが実装されているか、スコープの結果が期待通りであるかをテストする。

- requests/users_spec.rb

ルーティング、生成された HTML がシナリオとおりであるかを検証する。

ここまで、View についてのテストがないとお気づきかもしれません。View はデザインと結びつきが強く HTML の構造が頻繁に変更されます。そのため、メンテナンスすることが難しく requests などの結合テストに含めることが多いです。仕事で Rails アプリケーションを開発する際は、会社によって開発規約が異なりますので、View が必須の場合もあります。

7.2.2 RSpec の実行方法

RSpec を実行して結果を確認してみましょう。まず、テスト環境のデータベースを作成します。

```
$ RAILS_ENV=test bin/rake db:migrate
Running via Spring preloader in process 55637
== 20170828063211 CreateUsers: migrating ================================
-- create_table(:users)
   -> 0.0011s
== 20170828063211 CreateUsers: migrated (0.0014s) =======================
```

では、データベースが問題なく作成できたら、実行してみましょう

```
$ bin/rake
Running via Spring preloader in process 55691
/Users/liberty/.rvm/rubies/ruby-2.4.1/bin/ruby -I/Users/liberty/.rvm/
gems/ruby-2.4.1/gems/rspec-core-3.6.0/lib:/Users/
liberty/.rvm/gems/ruby-2.4.1/gems/rspec-support-3.6.0/lib /Users/
liberty/.rvm/gems/ruby-2.4.1/gems/rspec-core-3.6.0/exe/
rspec --pattern spec/\*\*\{,/\*/\*\*\}/\*_spec.rb

UsersController
  GET #index
    returns a success response
    (PENDING: Add a hash of attributes valid for your model)
  GET #show
    returns a success response
    (PENDING: Add a hash of attributes valid for your model)
  GET #new
    returns a success response
  GET #edit
    returns a success response
    (PENDING: Add a hash of attributes valid for your model)
  POST #create
    with valid params
      creates a new User
      (PENDING: Add a hash of attributes valid for your model)
      redirects to the created user
      (PENDING: Add a hash of attributes valid for your model)
    with invalid params
      returns a success response (i.e. to display the 'new' template)
      (PENDING: Add a hash of attributes invalid for your model)
  PUT #update
    with valid params
      updates the requested user
```

```
      (PENDING: Add a hash of attributes valid for your model)
      redirects to the user
      (PENDING: Add a hash of attributes valid for your model)
    with invalid params
      returns a success response (i.e. to display the 'edit' template)
      (PENDING: Add a hash of attributes valid for your model)
  DELETE #destroy
    destroys the requested user
    (PENDING: Add a hash of attributes valid for your model)
    redirects to the users list
    (PENDING: Add a hash of attributes valid for your model)

User
  add some examples to (or delete) /Users/liberty/monka/workspace/
  rspec-mockups/spec/models/user_spec.rb (PENDING: Not yet implemented)

Users
  GET /users
    works! (now write some real specs)

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

  1) UsersController GET #index returns a success response
     # Add a hash of attributes valid for your model
     # ./spec/controllers/users_controller_spec.rb:45

  2) UsersController GET #show returns a success response
     # Add a hash of attributes valid for your model
     # ./spec/controllers/users_controller_spec.rb:53

  3) UsersController GET #edit returns a success response
     # Add a hash of attributes valid for your model
     # ./spec/controllers/users_controller_spec.rb:68

  4) UsersController POST #create with valid params creates a new User
     # Add a hash of attributes valid for your model
     # ./spec/controllers/users_controller_spec.rb:77

  5) UsersController POST #create with valid params
  redirects to the created user
     # Add a hash of attributes valid for your model
     # ./spec/controllers/users_controller_spec.rb:83

  6) UsersController POST #create with invalid params
  returns a success response (i.e. to display the 'new' template)
     # Add a hash of attributes invalid for your model
     # ./spec/controllers/users_controller_spec.rb:90

  7) UsersController PUT #update with valid params updates the requested user
     # Add a hash of attributes valid for your model
     # ./spec/controllers/users_controller_spec.rb:103

  8) UsersController PUT #update with valid params redirects to the user
     # Add a hash of attributes valid for your model
     # ./spec/controllers/users_controller_spec.rb:110
```

```
  9) UsersController PUT #update with invalid params
  returns a success response (i.e. to display the 'edit' template)
     # Add a hash of attributes valid for your model
     # ./spec/controllers/users_controller_spec.rb:118

  10) UsersController DELETE #destroy destroys the requested user
     # Add a hash of attributes valid for your model
     # ./spec/controllers/users_controller_spec.rb:127

  11) UsersController DELETE #destroy redirects to the users list
     # Add a hash of attributes valid for your model
     # ./spec/controllers/users_controller_spec.rb:134

  12) User add some examples to (or delete)
  /Users/liberty/monka/workspace/rspec-mockups/spec/models/user_spec.rb
     # Not yet implemented
     # ./spec/models/user_spec.rb:4


Finished in 0.3894 seconds (files took 1.94 seconds to load)
14 examples, 0 failures, 12 pending
```

`rails generate scaffold` コマンドだけでこれだけのテストが自動生成されることが分かると思います。これらのテストひとつずつに正しい実装をしていきます。

7.3 Ruby on Rails： Rails テスト基礎 3

7.3.1 例題：テストの実装

Model テスト

Model でバリデータを定義してそれが正しく定義されているか確認するテストを実装してみましょう。利用者は名前と電話番号を持ち、それらは必ず入力しなければならないとします。

- app/models/user.rb

```
class User < ApplicationRecord
  validates :name, presence: true
  validates :phone_number, presence: true
end
```

また、テストデータは FactoryBot を利用するので次のように実装します。

- spec/factories/users.rb

```
FactoryBot.define do
  factory :user do
    name { Gimei.new.kanji }
    phone_number { "XXX-YYYY-ZZZZ" }
  end
end
```

FactoryBot を使って実装したデータは `build(:user)` や `create(:user)` メソッドを使ってテストデータを作成することが出来ます。`rails console` コマンドでプロンプトを立ち上げて実行してみましょう。テストデータが作成されることが確認出来ましたか？

テーブルを作成します。

```
rails db:migrate
```

コンソールを起動します。

```
$ rails console

Running via Spring preloader in process 55944
Loading development environment (Rails 5.1.3)
2.4.1 :001 > FactoryBot.build :user
 => #<User id: nil, name: "池上 羽海", phone_number: "XXX-YYYY-ZZZZ",
 created_at: nil, updated_at: nil>
2.4.1 :002 >
```

肝心のテストですが、利用者についてのテストは次のようになります。実際に入力してテストの結果が変更されることを確認してみましょう。

```
RSpec.describe User, type: :model do
  it "is valid with name and phone number" do
    user = build(:user)
    expect(user).to be_valid
  end

  it "is invalid without name" do
    user = build(:user, name: nil)
    user.valid?
    expect(user.errors[:name]).to include("can't be blank")
  end

  it "is invalid without phone number" do
    user = build(:user, phone_number: nil)
    user.valid?
    expect(user.errors[:phone_number]).to include("can't be blank")
  end
end
```

メソッドをひとつずつ解説するときりがありませんが、RSpec の構造は次のようになります。

- describe(context)

引数にどのようなテストを行うか示すために、メソッド名やルーティングを記述してテストのグルーピングを行います。

```
# Model の場合
describe "#post" do
  # post メソッドのテスト
end

# Controller の場合
describe "GET #edit" do
  # edit メソッドのテスト
End
```

context は describe のエイリアスですが、こちらは条件を記述します。

```
describe "#post" do
  context "params is invalid" do
    # params に不正な値がある場合
  end
end
```

- it

期待する結果を記述します。テスト結果が期待値であるかの検証しません。

```
describe "POST #create" do
  context "with valid params" do
    it "creates a new User" do
      # create メソッドが実行されると、User モデルにレコードが保存されることを期待する
    end
  end
end
```

- expect

テスト結果が期待値であるかの検証をします。構文は expect(期待値).to マッチャになります。マッチャはテスト対象によってことなります。次の例では利用者が新規作成されていますので、マッチャは次の通りです。

・テーブルの総数は 1 つ増える
change(User, :count).by(1)
・新規作成に成功したら詳細ページに遷移する
redirect_to(User.last)

```
describe "POST #create" do
    context "with valid params" do
      it "creates a new User" do
        expect {
          post :create, params: {user: valid_attributes},
          session: valid_session
        }.to change(User, :count).by(1)
      end

      it "redirects to the created user" do
        post :create, params: {user: valid_attributes},
        session: valid_session
        expect(response).to redirect_to(User.last)
      end
    end
end
```

始めはわかりにくいかもしれませんが、形式に囚われずに「何をテストしたいか」を主軸にテストを作成することをおすすめします。

System テスト

エンドツーエンド (E2E) のテストを行います。

Javascript を利用した画面等のテストを作成します。
spec/system ディレクトリを作成します。

```
mkdir spec/system
```

テストコードを記述する users_spec.rb を作成します。

```
touch spec/system/users_spec.rb
```

SystemSpec は毎回 Chrome を利用しますが、今回の E2E テストは、JavaScript がなくても実行可能なテストなので、Chrome を利用しない設定にします。
spec/rails_helper.rb に下記の設定を記述しましょう。 これで js: true のタグが付いているテストケースだけ Chrome を利用するようになりました。

- spec/rails_helper.rb

```
RSpec.configure do |config|
  #(省略)

  config.before(:each) do |example|
    if example.metadata[:type] == :system
      if example.metadata[:js]
        driven_by :selenium_chrome_headless, screen_size: [1400, 1400]
      else
        driven_by :rack_test
      end
    end
  end
end
```

- spec/system/users_spec.rb

```
require 'rails_helper'

RSpec.describe "Users", type: :system do
```

```
describe "GET /users" do
  it "renders new user link" do
    visit users_path
    assert_text "New User"
  end
end

describe "GET /users/:id" do
  let(:user) { create(:user) }

  it "renders a details of a user" do
    visit user_path(user)

    assert_text "#{user.name}"
    assert_text "#{user.phone_number}"
  end
end

describe "GET /users/new" do
  it "renders a new user form" do
    visit users_path

    click_on "New User"
    fill_in "Name", with: "User_Name"
    fill_in "Phone number", with: "XXX-YYYY-ZZZZ"

    click_on "Create User"
    assert_text "User was successfully created."
  end
end

describe "GET /users/:id/edit" do
  let(:user) { create(:user) }

  it "shows a details of a user" do
    visit edit_user_path(user)

    assert_text "Editing User"

    click_on "Update User"
    assert_text "User was successfully updated."
  end
end

describe "PATCH /users/:id/edit" do
  let(:user) { create(:user) }
  let(:new_name) { Gimei.new.kanji }
  let(:new_phone_number) { "AAA-BBBB-CCCC" }

  it "redirects to '/users/:id'" do
    visit edit_user_path(user)

    fill_in "Name", with: "#{new_name}"
    fill_in "Phone number", with: "#{new_phone_number}"

    click_on "Update User"
```

```
      expect(page).to have_content("User was successfully updated.")
    end
  end

  describe "DELETE /users/:id" do
    let!(:user) { create(:user) }

    it "redirect_to '/users'" do
      visit users_path

      click_on 'Destroy'
      expect(page).to have_content("User was successfully destroyed.")
    end
  end
end
```

7.4 [評価課題]Rails テスト基礎

ここで、Rails テスト基礎で学んできたことのポイントを確認してみましょう。

model spec, system spec などの役割

- model spec

モデルテストは、主に単体のテストを行います。よく利用される gem として、RSpec を Rails で利用するための gem の(Q1:)、テストデータ作成の補助をする gem の(Q2:)、ランダムに日本人の名前を生成する gem の(Q3:)があります。

- system spec

システムテストは(Q4:)のテストを行います。
(Q5:)を使用して、画面などのテストをします。

設定ファイルの作成

```
$ rails generate (Q6:                    )
```

以下のコマンドで、モデルを作成した場合、

```
$ rails generate scaffold User name:string phone_number:string
```

- controllers/users_controller_spec.rb

はコントローラへの(Q7:)を観点にテストをします。

- models/user_spec.rb

はモデルへの単体テストを観点に実装する。仕様書通りに(Q8:)が実装されているか、(Q9:)の結果の検証をするテストをします。

- requests/users_spec.rb

は(Q10:)、生成された HTML がシナリオ通りであるかを検証するテストをします。

Rails のテストの基本的な書き方

以下は、テストが記述されているコードの一部です。 ヒント：Q11 にはグルーピングのキーワード、Q12 には期待する結果を記述する書き出しのキーワード、Q13 には検証したい内容を書き出すためのキーワードです。

```
(略)
(Q11:           ) "POST #create" do
    context "with valid params" do
      (Q12:            ) "creates a new User" do
        expect {
          post :create, params: {user: valid_attributes},
          session: valid_session
        }.to change(User, :count).by(1)
      end

      (Q12:            ) "redirects to the created user" do
        post :create, params: {user: valid_attributes},
        session: valid_session
        (Q13:            )(response).to redirect_to(User.last)
      end
    end
end
(略)
```

```
RSpec.describe "Users", type: :system do

  (略)
  (Q11:           ) "GET /users" do
    (Q12:            ) "renders new user link" do
      (Q14:            ) users_path
      (Q15:            ) "New User"
    end
  end

  (Q11:           ) "GET /users/:id" do
    let(:user) { create(:user) }

    (Q12:            ) "renders a details of a user" do
      (Q14:           ) user_path(user)

      (Q15:            ) "#{user.name}"
      (Q15:            ) "#{user.phone_number}"
    end
  end


  (Q11:           ) "GET /users/new" do
```

```
    (Q12:            ) "renders a new user form" do
      (Q14:            ) users_path

      click_on "New User"
      fill_in "Name", with: "User_Name"
      fill_in "Phone number", with: "XXX-YYYY-ZZZZ"

      click_on "Create User"
      (Q15:            ) "User was successfully created."
    end
  end
  (略)
end
```

第８章　Ruby on Rails EC サイトの開発１

第８章　Ruby on Rails EC サイト開発１

8.1 Ruby on Rails： EC サイトの開発 商品一覧 1

8.1.1 商品一覧の作成

ここからは、要求仕様に従って EC サイトを構築していきます。なるべく Rails らしい機能やデフォルトの設定を使っていきますが、プロジェクトによっていろいろな方法や考え方をするときがあります。これらがすべてではないということを頭の片隅に置いておいてください。

また、git を利用して適切なバージョン管理を心がけてください。git にまだ自信がない人は git コマンドを復習し、まずはコマンドを実行するごと、1 つのファイルを変更するごとにコミットする癖をつけましょう。

8.1.2 EC サイトの概要とタイムゾーンの設定

(a) 解説

本書で構築する EC サイトの要求仕様は次のとおりです。一般的な EC サイトよりは機能が少なめですが、Rails でアプリを作る際に必要なことはほぼ網羅しています。

①　管理画面

1.1 商品の CURD と商品画像のアップロード
1.2 管理者のログイン認証
1.3 注文状態（全体）の確認
1.4 カート状態（全体）の確認

②　ユーザー画面

2.1 商品一覧と商品詳細ページ
2.2 商品の注文
2.3 注文完了後のメール送信
2.4 商品検索

もし、アプリをゼロから作ったことがない方は、要求仕様をもらったあとの構築手順も考えてみましょう。自社サービスの開発などでは必ずしもこの順番になりませんが、やることは同じです。本書のはじめに出てきたアジャイル開発の手法も参考にしてみてください。デザインの適応のタイミングは、プロジェクトの都合で後になることもあります。筆者の経験上、早いうちにデザインを当てていく方がアプリを構築している実感がわき、作業をするときのやる気が全然違います。

トップダウンでのアプリ構築手順（参考）

1. DB 設計
2. 画面遷移の決定
3. プロジェクト、MVC の作成
4. デザインの適用
5. 機能実装（テスト含む）
6. デプロイ

(b) 例題

ここではタイムゾーンの設定を紹介します。機能を実装していく前に必ず行ってください。Rails の時刻のデフォルトは UTC です。このタイムゾーンをきちんと管理しないと、サーバーのタイムゾーンとの整合性が取れなかったり、海外向けサービスを考えている場合は、データ間で時差が生じて大変なことになります。テストを実行する場合にも開発者がハマりやすいポイントですので注意してください。

設定は簡単です。`config/application.rb` の中に次の 1 行を追加してください。

```
module Monka
  class Application < Rails::Application
    config.time_zone = 'Tokyo' # 追加
  end
end
```

ちなみに、日本国内では `Osaka` と `Sapporo` も設定できます。

config フォルダ内の設定を変更したので、サーバーを再起動すれば設定完了です。確認方法としては、`rails console` で Time.zone を呼び出してみて、意図した時刻オフセットがついていれば OK です。

設定前（デフォルト）

```
username:~/workspace (master) $ rails c
Running via Spring preloader in process 37469
Loading development environment (Rails 5.0.0.1)
2.3.0 :001 > Time.zone.now
 => Tue, 13 Dec 2016 10:11:37 UTC +00:00
```

設定後（Tokyo の場合）

```
username:~/workspace (master) $ rails c
Running via Spring preloader in process 37503
Loading development environment (Rails 5.0.0.1)
2.3.0 :001 > Time.zone.now
 => Tue, 13 Dec 2016 19:11:59 JST +09:00
```

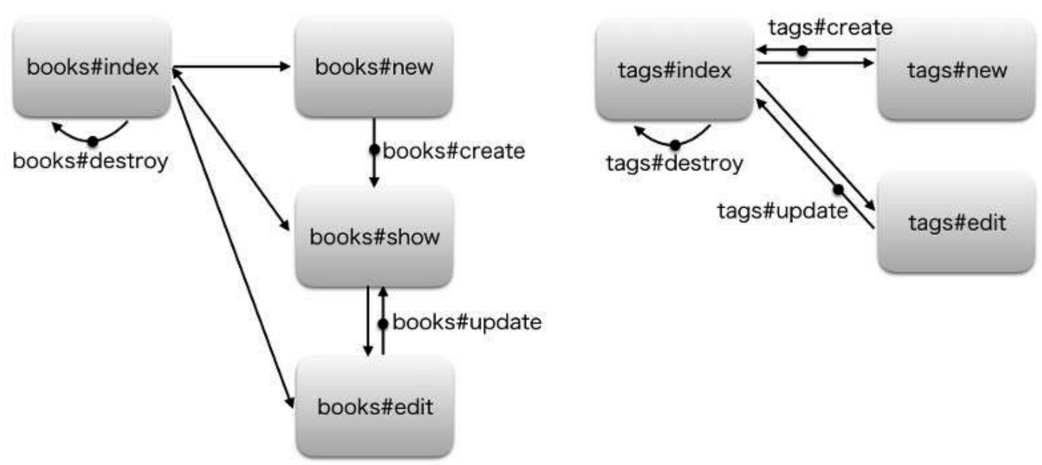
(c) 問題

EC サイトにタイムゾーンを設定しましょう。

8.1.3 画面遷移とルーティングの設定

(a) 解説

画面遷移の検討、決定方法については UI やプロジェクトの進行にも関連し、本書の目的を超えてしまいますのでここでは割愛します。今回は以下のように、scaffold をベースにした遷移を作成してください。このとき、各ページの URL はなるべく Rails のデフォルトに沿ったものを使用しましょう。共通の「レール」に乗った方法を使うことで、後々の仕様変更やバージョンアップに強くなり、開発の引き継ぎなどでもスムーズになります。



まず、scaffold で作成した Controller では、デフォルトのアクションの遷移は次のとおりになっています。

- 登録：new -> create -> show
- 編集：edit -> update -> show

この遷移は Controller で簡単に変更することができます。操作性やデータ量等を考慮し、場合に応じて設定を変更していきましょう。

また、不要になったアクションなどは、面倒がらずに削除するようにしてください。削除しておかないと、後々メンテナンス性の低下につながります。削除対象として注意するものは次のとおりです。

- Controller のアクション
- View の html ファイル
- ルーティング
- テスト

8.2 Ruby on Rails：EC サイトの開発 商品一覧 2

8.2.1 画面遷移とルーティングの実装

(b) 例題

以下は、地域(Area)テーブルを scaffold したあとのファイルです。まず、Controller の redirect_to の部分で、create と edit のあとは直接一覧ページへ遷移するように変更しましょう。

app/controllers/areas_controller.rb

```
# GET /areas/1 # 削除
# GET /areas/1.json # 削除
def show # 削除
end # 削除

def create
  @area = Area.new(area_params)

  respond_to do |format|
    if @area.save
      format.html { redirect_to areas_url,
                    notice: 'Area was successfully created.' } # 編集
      format.json { render :show, status: :created, location: @area }
    else
      format.html { render :new }
      format.json { render json: @area.errors,
                    status: :unprocessable_entity }
    end
  end
end

def update
  respond_to do |format|
    if @area.update(area_params)
      format.html { redirect_to areas_url,
                    notice: 'Area was successfully updated.' } # 編集
      format.json { render :show, status: :ok, location: @area }
    else
      format.html { render :edit }
      format.json { render json: @area.errors,
                    status: :unprocessable_entity }
    end
  end
end
```

また、不要になった詳細ページに関連するView、ルート、テストは全て削除します。

app/views/areas/show.html.erb

```
ファイルを削除
```

config/routes.rb

```
resources :areas, except: :show # 編集
```

test/controllers/areas_controller_test.rb

```
test "should show area" do # 削除
  get area_url(@area) # 削除
  assert_response :success # 削除
end # 削除
```

(c) 問題

商品タグ(Tag)の登録と編集のあと、直接一覧ページへ遷移するように変更しましょう。また、不要になった詳細ページに関連するものは全て削除しましょう。

8.3 Ruby on Rails：EC サイトの開発 商品一覧 3

8.3.1 データベース設計、多対多の関連付け

ここからは、EC サイトプリケーションの開発を例に各実装を解説していきます。 例題で各実装について解説し、問題を解くことにより EC サイトが少しずつ完成していきます。

まず、この章では、EC サイトのデータベース設計の解説を行います。 設計のために、ActiveRecord、データベース設計の基礎、多対多の関連付けについて簡単に説明します。

8.3.2 ActiveRecord とは

Active Record とは、Rails に付属する、重要なライブラリの 1 つで、MVC の M(モデル)に相当します。 Active Record は、ORM (オブジェクトリレーショナルマッピング) で実装されています。 ORM とは、簡潔に説明すると、アプリケーションが持つオブジェクトとリレーショナルデータベース(RDBMS)を繋ぐプログラミング技法です。

また、ActiveRecord では、下記の仕組みが特に重要となっています。

- モデルとそのデータを表す仕組み
- モデル間の関連性を表す仕組み
- 関連するモデルを通した階層の継承を表す仕組み
- DB に保存する前に検証する仕組み
- オブジェクト指向の手法で DB 操作を実行する仕組み

8.3.3 データベース設計の基礎

データベース設計はとても大事で難しい部分もあります。それだけで本が 1 冊書けるぐらいです。ここでは詳しく解説しませんが、データベースの構成次第で、あとの機能の実装がやりにくかったり複雑になってしまうので、くれぐれも注意してください。できるかぎり、サイトで扱うもの、関連のある人、サイトの使われ方などを具体的にイメージしながら考えていきましょう。

データベース設計で決めることは次の 4 つです。

- 作業1：アプリで扱うデータをモレなく書き出す
- 作業2：データの正規化（グループ分け）をする
- 作業3：各データの型（データの種類）を決める
- 作業4：各データのフィールド名（アルファベット）を決める

ここにあげたものは、必ずしも順番にする必要はありません。いろいろ考えていくうちに他に必要なデータが見えてきたり、「こっちの名前がこうなら、あっちの名前はこうしよう」 ということが出てきます。 サイトを利用する場面や利用する人のことを想像しながら、納得するまで考えます。可能であれば、他のプログラミング経験者の意見を聞くことをおすすめします。

8.3.4 多対多の関連付け

まず、多対多の関連付けとは、お互いのテーブルのレコード同士が複数の相手側レコードと関連付けられる関係の事です。
Railsで多対多の関連付けをする方法は以下の2通りがあります。

- has_many :through での関連付け

2つのモデルの間に、互いのモデルのIDを保持している中間テーブル(第3のモデル)を作成して、紐付けします。 例えば、次の[3.3-例題]のように、先生と授業の関係を表します。実際にどのように実装するかは例題を見ていきましょう。

- has_and_belongs_to_many での関連付け

has_many :through ではなく、has_and_belongs_to_many でも多対多の関連付けが可能ですが、 今回は、has_many :through での関連付けで実装していきますので詳細な説明は省きます。 has_and_belongs_to_many は has_many :through と仕様が異なっているため、詳しい仕様が気になる人は調べてみましょう。

8.3.5 ActiveRecord の代表的なメソッド

ここでは、以下のモデルを例に ActiveRecord の代表的なメソッドを簡潔に説明していきます。

モデル User:ユーザー

field名	名称	型
id	ID	integer

name	名前	string
mail_address	メールアドレス	string

ActiveRecordの代表的なメソッド一覧
※表の発行SQLは、わかりやすさのため、テーブル名は省略してカラム名のみ記述しています。

メソッド	説明	使用例	発行SQL
all	全件取得(全カラム)	User.all	SELECT * FROM users
select	全件取得(カラム指定)	User.select(:name)User.select('name, mail_address')	SELECT name FROM usersSELECT name,mail_address FROM users
find	検索(id指定)	User.find(1)	SELECT * FROM users WHERE id = 1
find_by	検索(条件指定)	User.find_by(id:1)User.find_by('id > 1')	SELECT * FROM users WHERE id = 1 LIMIT 1SELECT * FROM users WHERE id > 1 LIMIT 1
where	検索(条件指定)	User.where(id:1)User.where('id > 1')	SELECT * FROM users WHERE id = 1SELECT * FROM users WHERE id > 1
first	最初のデータをとる	User.firstUser.first(2)	SELECT * FROM users ORDER BY id ASC LIMIT 1SELECT * FROM users ORDER BY id ASC LIMIT 2
last	最後のデータをとる	User.lastUser.last(2)	SELECT * FROM users ORDER BY id DESC LIMIT 1SELECT * FROM users ORDER BY id DESC LIMIT 2
order	ソート	User.order(:name)User.order(name: :DESC)	SELECT * FROM users ORDER BY name ASCSELECT * FROM users ORDER BY name DESC
limit	制限	User.limit(2)	SELECT * FROM users LIMIT 2

各メソッドの詳細

- all

レコードを全件取得します

- select

カラムを指定し、レコードを取得します。引数の値がカラムとなります。

- find

指定した id のレコードを取得します。引数の値が指定する id となります。

find は、該当するデータが見つからない場合は例外（RecordNotFound）が発生します。

- find_by

特定のカラムの条件を指定し、該当する 1 件を取得します。引数の値が条件となります。

find_by は該当するデータが見つからない場合は、nil を返します。

- where

特定のカラムの条件を指定し、該当する全件を取得します。引数の値が条件となります。
where は、該当するデータが見つからない場合はからの、ActiveRecord::Relation を返します。

- first

レコードの最初の 1 件を取得します。引数を渡すと最初の n 件と指定することもできます。

- last

レコードの最後の 1 件を取得します。引数を渡すと最後の n 件と指定することもできます。

- order

レコードを引数に指定したカラムで並び変えます。デフォルトの並び順は ASC(昇順)になっています。
降順で並び変える場合は User.order(name: :DESC)とします。

- limit

特定のレコード件数を取得します。引数の値が最大取得行数となります。

etc

- ActiveRecord::Relation について

ActiveRecord::Relation とは、モデルオブジェクトのコレクションです。

ActiveRecord::Relation を返す場合は、メソッドチェーンが可能ですので、

例えば where に続いてさらに ActiveRecord のメソッドを使用する事ができます。

ここでは詳細を説明しませんので、気になる場合は調べてみましょう。

- ?の使い方

条件に変数を使用したい場合等に、?(プレースホルダ)を使用します。

変数でなくても、直接値を指定することも可能です。(冗長ですが例も記載しています。)

```
name = "test"
User.where("name = ?", name)
#SELECT  "users".* FROM "users" WHERE (name = 'test')

User.where("name = ?", 'test2')
#SELECT  "users".* FROM "users" WHERE (name = 'test2')
```

(b) 例題

先生と授業(科目)のテーブルを作成し、先生と授業の多対多で関連付けしていきます。先生は複数の授業(科目)を担当し、授業(科目)からも複数の先生が受け持っているという多対多の関連付けとして実装します。 データベース構成は以下の通りとします。

Teacher : 先生 テーブル

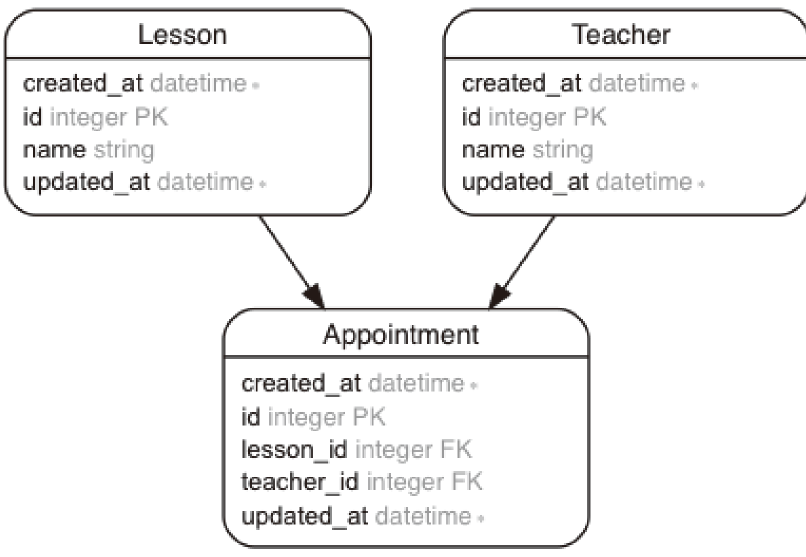
field 名	名称	型
name	名前	string

Lesson : 授業 テーブル

field 名	名称	型
name	授業名	string

Appointment : 出席者 テーブル（中間テーブル）

field 名	名称	型
teacher_id	先生	references
lesson_id	授業	references



① アプリケーションの作成

まずは、例題用の Rails アプリケーションを作成します。

```
$ rails new teacher_sample
```

② Teacher モデルの作成

次に必要なモデルを作成してきます。

```
$ rails generate model Teacher name:string
Running via Spring preloader in process 1718
      invoke  active_record
      create    db/migrate/20170828083314_create_teachers.rb
      create    app/models/teacher.rb
      invoke    test_unit
      create      test/models/teacher_test.rb
      create      test/fixtures/teachers.yml
```

③ Lesson モデルの作成

```
$ rails generate model Lesson name:string
Running via Spring preloader in process 1754
      invoke  active_record
      create    db/migrate/20170828083408_create_lessons.rb
      create    app/models/lesson.rb
      invoke    test_unit
      create      test/models/lesson_test.rb
      create      test/fixtures/lessons.yml
```

④ Appointment モデルの作成

中間テーブルとなるモデルです。 teacher と lesson を参照するように設定して、モデルを生成します。

```
$ rails generate model appointment teacher:references lesson:references
lesson:references
Running via Spring preloader in process 1786
      invoke  active_record
      create    db/migrate/20170828083440_create_appointments.rb
      create    app/models/appointment.rb
      invoke    test_unit
      create      test/models/appointment_test.rb
      create      test/fixtures/appointment.yml
```

teacher、lesson と appointment のモデルが作成できたので、テーブルを作成するためにマイグレーションも実行しましょう。

```
$ rails db:migrate
== 20170828083314 CreateTeachers: migrating ===============================
-- create_table(:teachers)
   -> 0.0014s
== 20170828083314 CreateTeachers: migrated (0.0015s) ======================

== 20170828083408 CreateLessons: migrating ================================
-- create_table(:lessons)
   -> 0.0011s
== 20170828083408 CreateLessons: migrated (0.0012s) =======================

== 20170828083440 CreateAppointments: migrating ===========================
-- create_table(:appointment)
   -> 0.0036s
== 20170828083440 CreateAppointments: migrated (0.0037s) ==================
```

各モデルの関係を設定するために、以下の内容を追記してください。
has_many :through は、appointment をショートカットして、teacher もしくは lesson を参照できるようにします。

```
app/models/teacher.rb
class Teacher < ApplicationRecord
  has_many :appointments
  has_many :lessons,through: :appointments
end
```

app/models/lesson.rb

```
class Lesson < ApplicationRecord
  has_many :appointments
  has_many :teachers,through: :appointments
end
```

appointment モデルは teacher と lesson を参照するように生成したため、既に下記のソースコードとなっています。

app/models/appointment.rb

```
class Appointment < ApplicationRecord
  belongs_to :teacher
  belongs_to :lesson
end
```

(c) 問題

今回の EC サイトのデータベース構成は、以下のようにします。
この構成通りに、モデルを作成してみましょう。本と商品は多対多の関係になることに注意して下さい。
Book : 本 テーブル

field 名	名称	型
title	タイトル	string
author	著者	string
published_on	出版日	date
showing	商品表示	boolean
price	価格	integer

Tag : 商品タグ テーブル

field名	名称	型
name	タグ名	string

Tagging : タグ付け テーブル

field名	名称	型
book_id	本	references
tag_id	商品タグ	references

8.3.6 中間テーブルへのデータ登録設定

(a) 解説

中間テーブルは、これまで出てきたUserやBookなどのように、scaffoldでできるnewやeditのViewを用意してデータを登録・編集することはほとんどありません。最低でも、紐づいたテーブルのどちらかのデータがないと、成り立たないというのが主な理由です。中間テーブルへの登録・編集は、様々な方法があります。ここでは紙面の都合上、Railsの便利なメソッド`collection_check_boxes`を利用してシンプルに実装していきます。このメソッドを利用しない方法については、余裕があればぜひチャレンジしてみてください。

8.4 Ruby on Rails ： EC サイトの開発 商品一覧 4

8.4.1 中間テーブルのデータ表示と保存の処理

(b) 例題

親テーブルにあたるところで一緒にデータを登録・編集できるようにしていきます。例題として、車のメーカと車種(Type)を考えてみましょう。DB 設計の項を参考に、それぞれのテーブルについて CRUD を用意し、Model にリレーションの設定を追加してください。また、Type テーブルに車種のデータを 2 つ以上登録しておいてください。

Maker : 車メーカ テーブル

field 名	名称	型
name	メーカ名	string

Type : 車種 テーブル

field 名	名称	型
name	車種名	string

MakerType : メーカー車種 テーブル

field 名	名称	型
maker	メーカ名	references
type	車種名	references

Maker テーブルの登録・編集で、MakerType テーブルも一緒に編集できるようにしていきます。変更箇所は以下の 3 点です。

form へのチェックボックス追加

collection_check_boxes は、erb タグ 1 行だけで必要な数のチェックボックスを並べてくれます。第 1 引数は参照する親テーブル名、第 2 引数は中間テーブル名に _ids をつけたものを使用します。第 3 引数以降は通常のチェックボックスと同じですので説明は省略します。

app/views/makers/_form.rb

```
<div class="fields">
```

```
  <%= collection_check_boxes :maker, :maker_type_ids, Type.all, :id, :name %>
</div>
```

この *_ids は、ActiveRecord の collection_singular_ids と呼ばれるもので、リレーションの関係があるテーブル間で使用できます。例えば、上の例では Maker.first.type_ids のように、Type テーブルの id をまとめて取得することもできます。

```
[*] pry(main)> Maker.first.type_ids
=> [1, 2]
```

Controller の修正

次は Controller の修正です。フィールドを追加しましたので、ストロングパラメータに追加しておきます。このとき、 collection_check_boxes は、チェックボックスの値が配列として params に追加されるので注意してください。中間テーブル用の save などは追加しなくても、maker_type_ids が自動的にデータを create/edit してくれます。

```
app/controllers/makers_controller.rb
def maker_params
  params.require(:maker).permit(:name, maker_type_ids: []) # 編集
end
```

show への表示追加

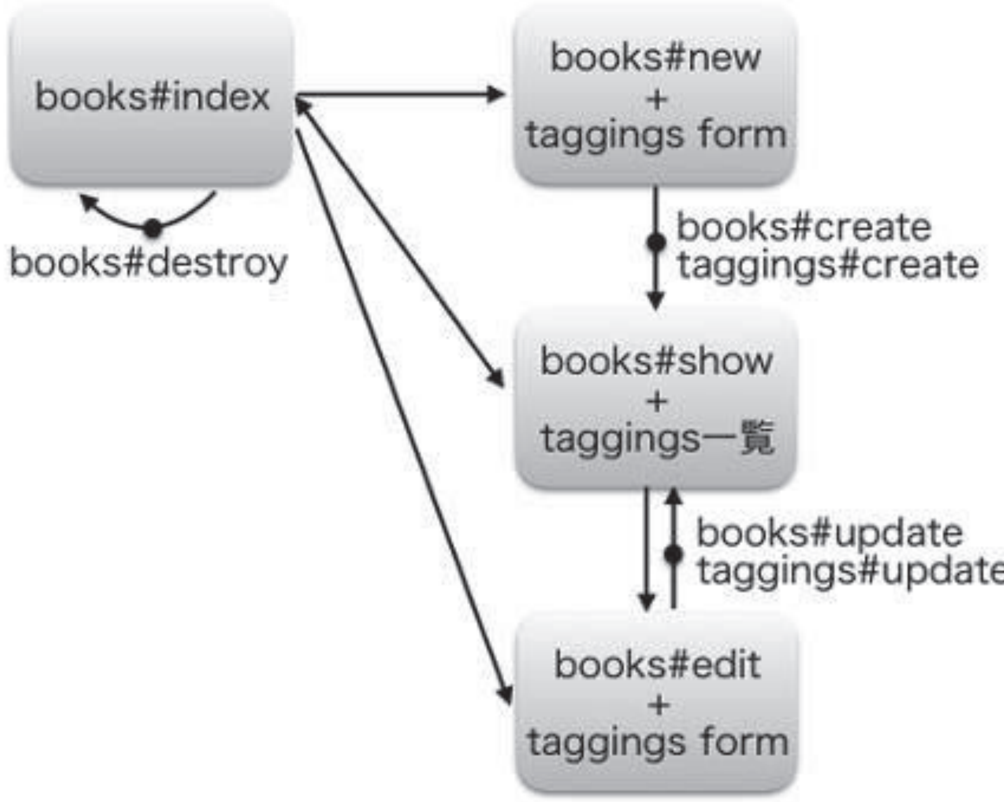
最後に、登録した中間テーブルのデータを表示しましょう。ここではシンプルに車種名を並べておきます。次のデザインテンプレートの項で素敵に編集してみてください。

app/views/makers/show.html.erb

```
<p>
  <strong>タグ</strong>
  <%= @maker.maker_types.map(&:name).join(', ') %>
</p>
```

(c) 問題

以下の図のように、Tagging データの設定を追加しましょう。



コラム：全体を俯瞰することとテストについて

scaffold で作成した Controller の show や edit アクションを見ると、中身は何もありませんがきちんと動作します。ご存知のとおり、フィルタと呼ばれる before_action で設定しています。（ちなみに、フィルタには after_action というものもあります。）また、モデルにはコールバックと呼ばれるものがあります。 before_save, after_save, before_create, after_create など、いろいろなタイミングで評価させるものがあります。さらに、Ruby は他のクラスやモジュールを継承しており、例えば Rails の Controller では application_controller.rb を継承して、そこで何か事前に処理をしているときもあります。筆者の経験からも、Controller の 1 つのアクションだけを見てその機能を把握することは、ほぼ無理です。 一般の書籍やこのようなテキストでプログラミングを学ぶと、どうしても特定の部分だけを注目しがちです。しかし、普段から Rails がどういうふうに動作しているか全体をイメージしながら実装していくと、思わぬ動作になったときに予測がつけられるようになります。また、テストを書くことによって、そいういった仕様や設計者の意図を伝えることにもなります。そうして、全体を俯瞰したりコードで意思疎通ができるようになることが、上級者の第一歩です。

8.5 Ruby on Rails: バリデーションとフォームヘルパー

8.5.1 バリデーション(validation:検証)とは

ここでは、ActiveRecord のバリデーション(検証)機能について説明します。 バリデーション機能とはオブジェクトがデータベースに登録される際に、そのデータが正しいものなのかを検証する仕組みです。文字の入力制限や型制限など Model 単位で設定することができます。

バリデーションがトリガされるタイミングは、オブジェクトに対して以下のメソッドを使用しデータベースに何らかの変更が行われようとしたときです。下記のメソッドが呼び出されるとデータベースに保存する直前にバリデーションを実行します。バリデーションで何らかのエラーが発生した場合はオブジェクトは保存されません。

- create
- create!
- save
- save!
- update
- update!

手動でバリデーションをトリガしてみる

valid?メソッドを使って手動でバリデーションをトリガすることができます。 オブジェクトに対して valid?メソッドを使用しバリデーションを実行し、そのオブジェクトにエラーがなければ「true」エラーが存在すれば「false」を返します。 invalid も用意されており valid?メソッドの逆の動きをします。

8.5.2 バリデーションを設定してみる

ActiveRecord には、バリデーションヘルパーが多数用意されており、これらのヘルパーは、共通のバリデーションルールを提供します。 実際に Model を用意してバリデーションを設定してみましょう。

プロジェクトの作成

```
$ rails _5.1.3_ new validates_sample
```

Model:User を用意します。

User: ユーザー テーブル

field 名	名 称	型
id	ID	integer
name	名前	string
age	年齢	integer
password	パスワード	string
email	メールアドレス	string

User:MVC を作成

後述する confirmation ヘルパーを使用したバリデーションを設定する際、View テンプレートへの記述が必要になるので scaffold コマンドを使用しています。

```
$ rails g scaffold User name age:integer passoword email
```

rails db:migrate で users テーブルを作成しておきましょう。

```
$ rails db:migrate
```

presence

このヘルパーは指定された属性が「空ではない」ことを確認します。値が nil や空文字(空欄やホワイトスペース)だとエラーが発生しデータベースに保存できません。入力必須項目として設定するという認識でよいでしょう。

フィールド「name」「password」「mail_address」にバリデーションを設定してみましょう。

app/models/user.rb

```
class User < ApplicationRecord
  validates :name, :age, :password, presence: true
end
```

length

length ヘルパーは、値の長さを検証します。多数のオプションが用意されており、何文字以上何文字以下など細かくバリデーションを設定することが可能です。

オプション	意味
minimum:	指定した値より小さい値は保存できない
maximum:	指定した値より大きい値は保存できない
in:または within:	指定した値の範囲内でなければならない
is:	指定した値と等しい長さでなければならない

name フィールドに、長さ「3 文字以上」という制限を追加しましょう。
age フィールドには、長さ「3 以内」という制限を追加しましょう。
password フィールドには、長さ「6 文字以上 20 文字以内」という制限を追加しましょう。

app/models/user.rb

```
class User < ApplicationRecord
  validates :name, presence: true, length: { in: 3..10 }
  validates :age, presence: true, length: { maximum: 3 }
  validates :password, presence: true, length: { in: 6..20 }
end
```

numericality

numericality は、数値のみで構成された値かどうかを検証します。デフォルトでは整数と浮動小数点のみ許可し、符号がある場合も許可します。整数のみ許可したい場合は「only_integer: true」を指定します。

age フィールドに整数のみ許可するように設定を追加しましょう。

app/models/user.rb

```
class User < ApplicationRecord
  validates :name, presence: true, length: { in: 3..10 }
  validates :age, presence: true, length: { maximum: 3 }, numericality:
     { only_integer: true }
```

```
  validates :password, presence: true, length: { in: 6..20 }
end
```

uniqueness

このヘルパーは、field の値が一意(unique)であり重複していないことを検証します。
name フィールドと email フィールドに一意性を設定しましょう。

app/models/user.rb

```
class User < ApplicationRecord
  validates :name, presence: true, length: { in: 3..10 }, uniqueness: true
  validates :age, presence: true, length: { maximum: 3 }, numericality:
    { only_integer: true }
  validates :password, presence: true, length: { in: 6..20 }
  validates :email, uniqueness: true
end
```

confirmation

このヘルパーは、2 つの受け取った値が完全に一致する必要がある場合に使用します。
メールアドレスやパスワードなどで使用します。 このバリデーションヘルパーは仮想の属性を作成します。属性の名前は、確認したい属性名に「_confirmation」を追加したものになります。

password に入力確認を設定しましょう。

app/models/user.rb

```
class User < ApplicationRecord
  validates :name, presence: true, length: { in: 3..10 }, uniqueness:   true
  validates :age, presence: true, length: { maximum: 3 }, numericality:
    { only_integer: true }
  validates :password, presence: true, length: { in: 6..20 },
    confirmation: true
  validates :password_confirmation, presence: true
  validates :email, uniqueness: true
end
```

View テンプレートに「パスワード」と「パスワード入力確認用」のテキストフォームを 2 つ用意します。 password_confirmation にも presence: true を指定しているためテキストフィールドに入力しない場合その時点でエラーになります。

app/views/users/_form.html.erb

```
<%= text_field :user, :password %>
<%= text_field :user, :password_confirmation %>
```

format

このヘルパーは、with オプションで指定した正規表現と入力された値がマッチするか検証を行います。

email フィールドにフォーマットを設定しましょう。
:message オプションを使用しエラーだった場合に表示するメッセージを設定しています。

```
validates :name, presence: true, length: { in: 3..10 }, uniqueness: true
validates :age, presence: true, length: { maximum: 3 }, numericality:
  { only_integer: true }
validates :password, presence: true, length: { in: 6..20 }, confirmation:
  true
validates :password_confirmation, presence: true
validates :email, format: { with: /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i,
                            message: "使用できない文字が含まれているか形式に誤りがあり
  ます。" }, uniqueness: true
```

8.5.3 バリデーション実行時の動作を確認してみる

先程作成した Modle:User を使用して動作を確認してみましょう。 まずコンソールを起動します。

```
$ rails c
```

コンソールから User オブジェクトを作成し、変数に格納します。

```
irb(main) > u = User.new
```

User オブジェクトが格納された変数 u に対して valid?メソッドでバリデーションを実行してみましょう。
User オブジェクトが先ほど設定した制約を満たしていないので false が返されます。

```
irb(main) > u.valid? # => false
```

バリデーションエラーの詳細を見る

先ほどバリデーションエラーとなったUserオブジェクトのエラーの内容を確認してみましょう。errors.detailsでエラーの詳細を見ることができます。

```
irb(main) > u.errors.details

irb(main) > u.errors.details
=> {:name=>[{:error=>:blank},
     {:error=>:too_short, :count=>3}], :age=>[{:error=>:blank},
     {:error=>:not_a_number, :value=>nil}], :password=>[{:error=>:blank},
     {:error=>:too_short, :count=>6}], :password_confirmation=>[{:error=>:
     blank}], :email=>[{:error=>:invalid, :value=>nil}]}
```

errors.messages でエラー時に表示されるメッセージを見ることができます。 email に指定したメッセーシに設定されていることが確認できます。

```
irb(main) > u.errors.messages
irb(main):025:0> u.errors.messages
=> {:name=>["can't be blank", "is too short (minimum is 3
     characters)"], :age=>["can't be blank", "is not a
     number"], :password=>["can't be blank", "is too short (minimum is 6
     characters)"], :password_confirmation=>["can't be blank"], :email=>["
     使用できない文字が含まれているか形式に誤りがあります。"], :attribute=>[]}
```

8.5.4 form_with ヘルパーとは

Rails 5.1 から form_for, form_tag の代わりとなる form_with ヘルパーが追加されました。これにより form_for, form_tag ヘルパーは非推奨となりました。 form_with ヘルパーは、View ファイルで簡単にフォームを生成することができるヘルパーです。 まずはプロジェクトを作成し、プログラムがどのようになっているのか確認してみましょう。

```
$ rails new form_sample
$ cd form_sample
$ rails g scaffold User name age:integer
```

コードを確認してみましょう。

app/views/users/_form.html.erb

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

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

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

  <div class="field">
    <%= form.label :age %>
    <%= form.number_field :age %>
  </div>

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

`form_with` ヘルパーのブロック引数 `form` は、`form_with` ヘルパーで受け取ったモデルの情報を持つ FormBuilder オブジェクトです。FormBuilder オブジェクトが知っている `label` メソッドや `text_field` メソッドを使用しフォームを形成します。

`<%= form.label :name %>`でラベルを生成したり`<%= form.text_field :name %>`で、テキストボックスを生成したり、Rails が自動的に渡されたモデルオブジェクト(User)から HTML を生成してくれているのです。

`form_with` ヘルパーで生成されたフォームから入力された値は、Controller から `params[:カラム名]`という 形式で取得することができます。

例: `params[:name], params[:age]`

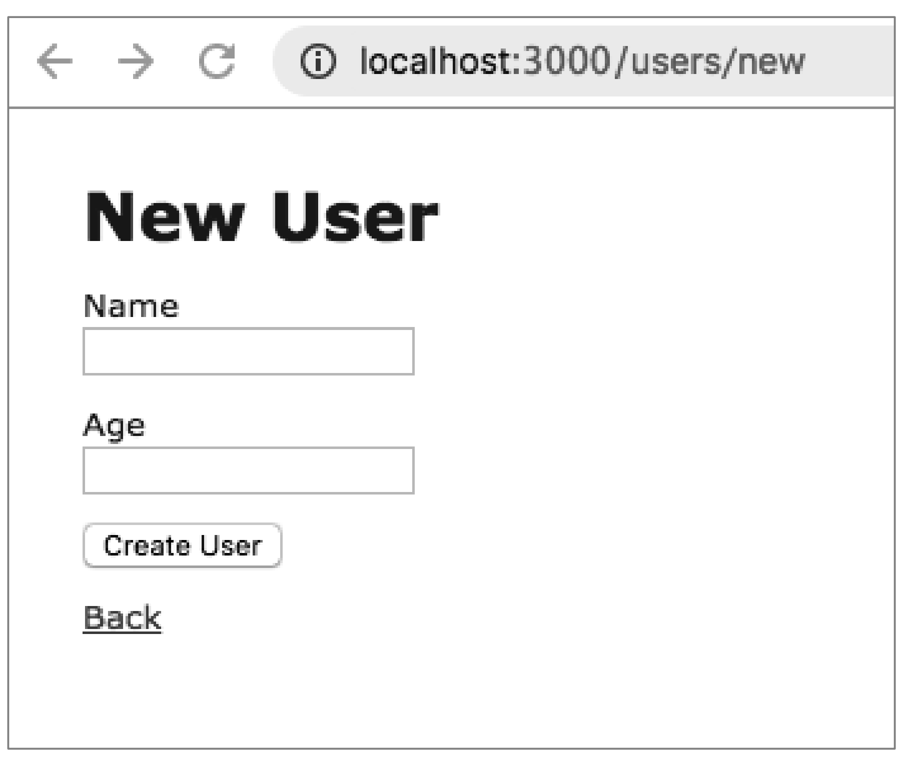
`_form.html.erb` によってどのような HTML が生成されるのか見てみましょう。

```
<form action="/users" accept-charset="UTF-8" method="post"><input type="hidden"
    name="authenticity_token"
```

```
    value="v/nQIldmxgikiUxiDxQ66PjTHUELCU99tfz2jcmyQ5eNLrLup0aprYlPLCaOM56fP/ht
    HiXoW5lbynU17psWkQ==" />

  <div class="field">
    <label for="user_name">Name</label>
    <input type="text" name="user[name]" id="user_name" />
  </div>

  <div class="field">
    <label for="user_age">Age</label>
    <input type="number" name="user[age]" id="user_age" />
  </div>

  <div class="actions">
    <input type="submit" name="commit" value="Create User" data-disable-
    with="Create User" />
  </div>
</form>
```

`$ rails s` サーバーを起動し、`http://localhost:3000/users/new` にアクセスしてみましょう。 以下のような画面が表示されているはずです。

```
$ rails s
```



8.5.5 form_with ヘルパーの使い方

form_with ヘルパーにはさまざまなオプションが用意されてます。 scaffold コマンドを使用して生成されたファイルには form_with ヘルパーに対して:model オプションと:local オプションが指定されていました。

```
form_with(model: user, local: true)
```

:model オプションは、生成するフォームがどのモデルに基づいて作成されるのかを指定するオプションです。 model: user と指定しているので、UsersController に POST リクエストを送信します。

その他にもさまざまなオプションが用意されています。

オプション	概要	デフォルト値
:url	入力された値を送信する送信先を指定します。	nil
:method	HTTP リクエストを指定します。	POST
:fromat	送信するデータ形式を指定します。	text/html
:id	id 属性を指定します。	nil
:class	class 属性を指定します。	nil
:data	data 属性を指定します。	nil
:html	id, class, data 以外の HTML 属性を指定します。	nil

他にも複数オプションが用意されています。 ここでは説明しませんが、他にもモデルに基づかないフォームを生成や form_for と form_with の違いなど調べてみると良いでしょう。

8.6 ログイン認証

ログイン機能を提供する gem を使用しないログイン機能を実装します。

8.5.1 ユーザー登録機能作成

プロジェクトの作成

```
$ rails _5.1.3_ new login_sample
```

パスワードはハッシュ化して保存する

パスワードは大切なので、データベースにそのまま保存してはいけません。
なぜなら、パスワードを暗号化していない状態で保存していて、悪意あるアタックでシステムのセキュリティが突破され、データベースの内容が見える形になると、非常にまずいことが想像できます。パスワードを暗号化していると、パスワードの文字が確認できてもそのまま利用することはできません。

そこで、パスワードをランダムな文字列に変換してデータベースに保存する方法としてハッシュ化を行います。

ハッシュ化された文字列から元のパスワードに復元することは一般的にはかなり難しいですが、同じパスワードからハッシュ化すると同じ文字列に変換されます。 ログインで認証をするときは、画面で入力されたパスワードをハッシュ化した文字列と、データベースに保存されているハッシュ化されたパスワードが一致するか確認することで安全に比較ができます。

（参考：https://www.ipa.go.jp/security/vuln/websecurity.html 「IPA の安全なウェブサイトの作り方」）

パスワードをハッシュ化するために、gem `bcrypt` を使用します。 `Gemfile` の `bcrypt` の部分がコメントになっている場合は、#を削除して有効にします。

`Gemfile`

```
# Use Active Model has_secure_password
gem 'bcrypt', '~> 3.1.7'
```

一度 bundle install を実行します。

```
bundle install
```

User: ユーザーテーブル

field 名	説明	型
id	ID	integer
name	名前	string
password_digest	ハッシュ化したパスワード	string

ログインするためにはユーザーが必要です。

ユーザーを登録できるよう scaffold コマンドで User の MVC をを作成しましょう。
password_digest 属性は、bcrypt の has_secure_password メソッドで利用されるので、他の名前は指定できません。
下記のコマンドを実行します。

```
rails g scaffold User name password_digest
```

下記のコマンドを実行し読み込みテーブルを作成します。

```
rails db:migrate
```

has_secure_password メソッドが使用可能になりました。
has_secure_password はパスワードのハッシュ化だけでなく、password_digest に対して以下のようなバリデーションを追加します。

- パスワードの存在性を検証する
- 入力された「パスワード」と「確認入力用パスワード」が等しいかどうかの有効性を検証する
- パスワードが 73 文字以上の場合のみ許可しない
-

UserModel 内で has_secure_password を使用しましょう。
app/models/user.rb

```
class User < ApplicationRecord
  has_secure_password
end
```

実際にバリデーションが追加されているのかコンソールを使用して確認してみましょう。

```
rails c
```

User オブジェクトを作成します。

```
user = User.new(name: "test_user")
```

無効な User インスタンスを確認してみましょう。
User オブジェクトに対して valid?メソッドで有効性を検証してみましょう。 false が返却されます。これは、UserModel のインスタンスである user はパスワードを保持していないためです。

```
user.valid? # => false
```

有効な User インスタンスを確認してみましょう。
has_secure_password を使用したことで仮想的に password 属性と password_confirmation 属性が作りだされ、password_digest 属性に保存されます。

```
user = User.new(name: "test_user", password: "password",
  password_confirmation: "password")
user.valid? # => true
```

ハッシュ化されているか確認してみましょう。

```
user.password_digest
# => "$2a$12$Ih7SjraevqUwPE.qFbS5DOj/029GAtcXiROaakrDsOec./2QSyLb."
```

パスワードの長さが 73 文字以上または password 属性と password_confirmation 属性の値が異なるパターンも検証してみるとよいでしょう。
has_secure_password だけでなく、以下のバリデーションも追加してします。

app/models/user.rb

```
class User < ApplicationRecord
  validates :name, presence: true, length: { in: 2..10 }
  validates :password, presence: true, length: { minimum: 6 }
  has_secure_password
end
```

次に画面からユーザー登録を行えるように app/views/users/_form.html.erb の内容を以下のように修正します。「パスワード」と「確認用のパスワード」を入力できるようフォームをフォームを修正しましょう。

app/views/users/_form.html.erb

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

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

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

  <div class="field">
    <%= form.label :password %>
    <%= form.password_field :password %>
  </div>

  <div class="field">
    <%= form.label :password_confirmation %>
    <%= form.password_field :password_confirmation %>
  </div>

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

次に、UsersController の StrongParameters で「パスワード」と「確認用パスワード」を受け取るようにを修正します。

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
.
.
.
  private
```

```
    # Never trust parameters from the scary internet, only allow the white list
    through.
    def user_params
      params.require(:user).permit(:name, :password, :password_confirmation)
    end
end
```

これでブラウザからユーザーを登録する準備ができましたので、一度ユーザーを作成します。
コンソールから rails s コマンドでサーバーを起動し、ブラウザから http://localhost:3000/users/new にアクセスし下記のユーザーを登録しましょう。

```
$ rails s
```

ユーザー情報

```
name: test_user
password: password
```

ここで作成したユーザーで次のログイン認証を行います。

8.6.2 SessionController を作成する

セッション情報を管理する SessionsController を作成します。 Sessions#new で下の画像のようなログイン画面を表示します。それぞれのフォームから入力された値を Sessions#create へ送り、有効なユーザーかどうかを検証し、有効なユーザーであればセッションを作成し「そのユーザーでログインした状態」を作成します。 さらに Sessions#destroy でログアウト処理を実装します。

Log in

Name

Password

Password confirmation

Log in

下記のコマンドで SessionsController を作成しましょう。
Sessions#create と Sessions#destory は対応する画面(View)はないので、new だけを指定します。

```
$ rails g controller Sessions new
```

8.6.3 ルーティングを定義する

SessionsController で使用するアクションに対してルーティングを定義します。routes.rb を下記の内容に修正しましょう。

config/routes.rb

```
Rails.application.routes.draw do
  root 'users#index'
  resources :users
  get    'login'   => 'sessions#new'
  post   'login'   => 'sessions#create'
  delete 'logout'  => 'sessions#destroy'
end
```

コンソールから rails routes コマンドでルーティングの定義を確認し画像のように SessionsController に対してルーティングが定義されていれば OK です。

```
   Prefix Verb   URI Pattern                       Controller#Action
     root GET    /                                 users#index
    users GET    /users(.:format)                  users#index
          POST   /users(.:format)                  users#create
 new_user GET    /users/new(.:format)              users#new
edit_user GET    /users/:id/edit(.:format)         users#edit
     user GET    /users/:id(.:format)              users#show
          PATCH  /users/:id(.:format)              users#update
          PUT    /users/:id(.:format)              users#update
          DELETE /users/:id(.:format)              users#destroy
    login GET    /login(.:format)                  sessions#new
          POST   /login(.:format)                  sessions#create
   logout DELETE /logout(.:format)                 sessions#destroy
```

8.6.4 ログインフォームを作る

ログインフォームを作成します。 form_with ヘルパーを使い入力フォームを作成します。通常 form_with は何も指定しなければ HTTP プロトコルは POST リクエストになります。

:url オプションで指定している login_path に POST でリクエストが送信されるので Sessions#create にアクセスします。 app/views/sessions/new.html.erb を下記の内容に修正しましょう。

app/views/sessions/new.html.erb

```
<div>
  <%= flash[:notice] %>
</div>

<h1>Log in</h1>

<%= form_with(scope: :sessions, url: login_path) do |f| %>
  <div>
    <%= f.label :name %>
    <%= f.text_field :name %>
  </div>

  <div>
    <%= f.label :password %>
    <%= f.password_field :password %>
  </div>

  <div>
    <%= f.submit "Log in" %>
  </div>
<% end %>
```

サーバーを停止している場合は、コンソールから rails s コマンドでサーバーを起動します。

```
$ rails s
```

localhost:3000/login にアクセスし、下記のような画面が表示されれば OK です。

Log in

Name

Password

Log in

次に作成するログイン認証で、ログインが成功したかどうか明示的にわかりやすいように、ユーザー一覧画面(users#index)を「ログインしていなければ表示されない」ように設定します。 ブラウザがセッション情報を保持していなければログイン画面へリダイレクトするように users_controller.rb を修正しましょう。

app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def index
    redirect_to login_path if session[:user_id].nil?
    @users = User.all
  end
  .
  .
  .
end
```

8.6.5 入力されたユーザー情報で認証する

ログイン画面から入力された情報を Sessions#create で受け取り、入力されたユーザー情報元にをデータベースから検索します。 ユーザーの存在性が確認され、入力情報の整合性が取れればそのユーザーを認証し、セッションを作成ます。存在しないユーザー情報であればエラーメッセージを表示しログインフォームへリダイレクトするように Sessions#create を実装します。

Sessions#create を下記のように作成します。
users テーブルから find_by メソッドを使用し、送信された値「params[:session][:name]」で検索します。
ユーザーが見つからなければ変数 user には nil が代入され、もう一度ログインフォームが表示されます。
authenticate メソッドは、Model で has_secure_passowrd を宣言すると使用できるメソッドで、入力されたパスワードがデータベースに登録されている password_digest 属性と一致するか検証します。不一致の場合は、ログインフォームが表示されます。
session メソッドは rails が提供するメソッドで、ハッシュのように扱えます。session[:user_id] = user.id とすることで、ブラウザの一時 cookies に暗号化されたユーザーID が保存されます。
session[:user_id]とすることで暗号化されていないユーザーID を取り出すことができます。
session メソッドで作成された cookies は、ブラウザを閉じると有効期限が終了します。
セッションが作成され、ログインが成功すればユーザー一覧画面へ遷移します。

```
app/controllers/sessions_controller.rb
class SessionsController < ApplicationController
  def new
  end

  def create
    user = User.find_by(name: params[:sessions][:name])
    if user && user.authenticate(params[:sessions][:password])
      session[:user_id] = user.id
      redirect_to root_path
    else
      flash[:notice] = 'Invalid name/password combination'
      redirect_to login_path
    end
  end

  def destroy
  end
end
```

8.6.6 ログアウトする

このプログラムではブラウザを閉じればセッション情報も削除され、同時にログアウトされた状態になりますが、ここでは、操作による簡単なログアウト機能について説明します。 Sessions#create 内で作成したセッション情報を、 Sessions#destroy で session メソッドを使用しセッション情報を削除してログアウトされた状態にします。

まず Sessions#destroy へアクセスするリンクを作成します。 destroy アクションには、DELETE リクエストでアクセスします。link_to ヘルパーの method:オプションで DELETE リクエストになるよう設定しましょう。

app/views/users/index.html.erb

```
.

.
.
<%= link_to 'New User', new_user_path %>
<%= link_to 'Log out', logout_path, method: :delete %>
```

次に destroy アクションを実装します。
destroy アクション内でセッション情報を削除し、root_path へリダイレクトするよう sessions_controller.rb に記述します。

```
app/controllers/sessions_controller.rb
class SessionsController < ApplicationController
.
.
.
  def destroy
    session.delete(:user_id)
    redirect_to login_path
  end
end
```

セッション情報が削除されるとユーザー一覧画面は表示されなくなり、ログイン画面へ遷移すればログアウト成功です。
以上で gem を使わず簡単ログイン機能を作ることができました。 その他 SessionHelper を使用したセッション管理やブラウザを閉じてもログイン状態を保持する永続セッションの作成などログイン機能を充実させることが出来るので、調べて実装してみましょう。

8.7 ログイン認証とユーザー管理

8.7.1 ログイン認証

商品一覧ページには特定の人しかアクセスできないようにするために認証機能を追加します。 認証を実現するために Devise という gem を使って実装します。

8.7.2 Devise とは

Devise とは Ruby on Rails でも利用できる認証機能を提供してくれる gem です。 いくつかのモジュールで構成されています。

モジュール名	説明
Database Authenticatable	ログインのためのパスワードをデータベースに暗号化して保存する。
Oauthable	OAuth を利用する。
Confirmable	登録したメールアドレスにメールを送信してメールアドレスの存在を確認
Recoverble	パスワードを忘れた際に、パスワードリセットして再登録用の URL をメールで送信する。
Registerable	ログインするためのアカウントを登録する。
Rememberable	ログイン状態を保持する
Trackable	ログインした回数や日時、IP アドレスなどを保存する。
Timeoutable	一定期間操作がない場合に、自動的にログアウトする。
Validatable	メールアドレスとパスワードのバリデーションを設定する。
Lockable	一定回数ログインに失敗した場合、そのアカウントをロックする。

上記の機能全てを説明できないので、今回説明できなかった機能については下記を参照してください。
https://github.com/plataformatec/devise/wiki

8.7.3 Devise のインストール

`Gemfile` に `devise` を追記しましょう。 追記したら忘れずに `bundle install` を行いましょう。

`Gemfile`

```
gem 'devise'
```

`bundle install` が完了したら下記のコマンドで `Devise` をインストールします。

```
$ rails g devise:install
```

8.7.4 ログイン処理の作成の流れ

まず、ログインに必要な情報を保存するためのモデルと、画面が必要です。
一般的に、アプリケーションやシステムにログインするためには、アカウント、パスワードが必要です。

ここでは、その情報を管理するためにモデルを作成します。
また、ログインを受け付ける画面や、ログインの状態に応じてアクセスできる情報の制御も必要ですから実装します。
ここでは、ログインの受付画面、ログインしてる場合だけアクセスできるマイページの作成を作成します。また、ログインのアカウントにはメールアドレスを利用します。最初にアクセスした場合はログインの認証に必要な情報がないので、アカウントの作成機能も必要です。

アカウント作成をする場合は、一般的に同じパスワード２回入力することが多いです。
これは、入力画面のパスワード欄に文字を入力しても、一般的にはそのまま表示はしないため、同じパスワードを入力することで意図したパスワードが入力できているか確認するためです。

ログインして、画面が変わってもログインの状態を維持できるようにセッション管理機能も必要です。

ログインができたら、ログアウトも必要です。ログイン、マイページができたら、ログアウトの画面と処理を実装します。

8.7.5 例題

認証が必要なマイページを作成します。

ログインするユーザのモデルを作成します。

```
$ rails g devise User
```

作成ができたら、出来上がったモデルを見てみましょう。

app/models/user.rb

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

先ほど説明した Devise を構成しているモジュールが記載されています。 今回利用するものは下記の 4 つです。

- Database Authenticatable
- Registerable
- Trackable
- Validatable

上記のものはコメントアウトを外し、 上記のもの以外はコメントアウトしましょう。
app/models/user.rb

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

次にマイグレーションファイルを見てみましょう。

```
db/migrate/20170824084617_devise_create_users
class DeviseCreateUsers < ActiveRecord::Migration[5.1]
  def change
    create_table :users do |t|
      ## Database authenticatable
      t.string :email,              null: false, default: ""
      t.string :encrypted_password, null: false, default: ""

      ## Recoverable
      t.string   :reset_password_token
      t.datetime :reset_password_sent_at

      ## Rememberable
      t.datetime :remember_created_at

      ## Trackable
      # t.integer  :sign_in_count, default: 0, null: false
      # t.datetime :current_sign_in_at
      # t.datetime :last_sign_in_at
      # t.string   :current_sign_in_ip
      # t.string   :last_sign_in_ip

      ## Confirmable
      # t.string   :confirmation_token
      # t.datetime :confirmed_at
      # t.datetime :confirmation_sent_at
      # t.string   :unconfirmed_email # Only if using reconfirmable

      ## Lockable
      # t.integer  :failed_attempts, default: 0, null: false # Only if lock
    strategy is :failed_attempts
      # t.string   :unlock_token # Only if unlock strategy is :email or :both
      # t.datetime :locked_at


      t.timestamps null: false
    end

    add_index :users, :email,                unique: true
    add_index :users, :reset_password_token, unique: true
    # add_index :users, :confirmation_token,   unique: true
    # add_index :users, :unlock_token,         unique: true
  end
end
```

利用するモジュールに必要なカラムが記載されています。 今回利用する 4 つのモジュールに必要なカラムはコメントアウトを外し、 それ以外のカラムはコメントアウトしましょう。

db/migrate/20170824084617_devise_create_users

```

    create_table :users do |t|
```

```
    ## Database authenticatable
    t.string :email,              null: false, default: ""
    t.string :encrypted_password, null: false, default: ""

    ## Recoverable
    # t.string   :reset_password_token
    # t.datetime :reset_password_sent_at

    ## Rememberable
    # t.datetime :remember_created_at

    ## Trackable
    t.integer  :sign_in_count, default: 0, null: false
    t.datetime :current_sign_in_at
    t.datetime :last_sign_in_at
    t.string   :current_sign_in_ip
    t.string   :last_sign_in_ip

    ## Confirmable
    # t.string   :confirmation_token
    # t.datetime :confirmed_at
    # t.datetime :confirmation_sent_at
    # t.string   :unconfirmed_email # Only if using reconfirmable

    ## Lockable
    # t.integer  :failed_attempts, default: 0, null: false # Only if lock
  strategy is :failed_attempts
    # t.string   :unlock_token # Only if unlock strategy is :email or :both
    # t.datetime :locked_at
```

そのあと、マイグレーションを実行しましょう

```
# rails db:migrate
```

マイページの作成

まずはマイページを作って表示できるようにしてみましょう。 マイページ用のルーティングを追加します。

config/routes.rb

```
get :mypage, to: 'mypage#index'
app/controllers/mypage_controller.rb
class MypageController < ApplicationController
  def index
  end
end
```

app/views/mypage/index.html.erb

```
<h1>マイページ</h1>
<p>ここはマイページです</p>
```

ここまでできたらサーバを起動して確認してみましょう。
rails s でサーバを起動して、https://projectname-username.c9users.io/mypage にアクセスすると、下記のような画面が表示されます。

しかし、この状態だとログインしていない人でもアクセスできてしまいます。 次にこのページに認証をかけて、ログインした人だけがアクセスできるようにします。 認証をかけるには `before_action :authenticate_user!`を使います。

app/controllers/mypage_controller.rb

```
class MypageController < ApplicationController
  before_action :authenticate_user!

  ...
end
```

これでマイページに認証がかかりました。早速アクセスしてみましょう。

Log in

Email

Password

Log in

Sign up

先ほどとは違って、ログイン画面が表示されたかと思います。

次にログインするためのユーザを作成します。 先ほど表示したログイン画面に `sign up` というリンクがあるのでクリックしてみましょう。

ここで Email, Password, Password confirmation を入力することでユーザが作成できます。

デフォルトではユーザを作成したら、ログインした状態になります。
早速マイページにアクセスしてみましょう。
今度はマイページが表示されました。
Devise を使えば、簡単にログイン機能が実装できます。

ログアウトの実装

まずはルーティングを確認してみましょう

```
$ rails routes
```

```
destroy_user_session DELETE /users/sign_out(.:format)
  devise/sessions#destroy
```

`delete destroy_user_session` というルーティングがあります。 Devise のログインは `session` を作成、ログアウトは session を破棄という表現ができます。 delete `destroy_user_session` のルーティングを使ってログアウトします。

マイページの view にログアウト用のリンクを追加してみましょう

app/views/mypage/index.html.erb

```
<h1>マイページ</h1>
<p>マイページ<p>
<p><%= link_to 'ログアウト', destroy_user_session_path, method: :delete %></p>
```

マイページにアクセスすると、ログアウトのリンクが表示されます。 クリックしてログアウトしてみましょう。
Rails の初期画面が表示されました。 これでログアウトができています。
試しにもう一度マイページにアクセスしてみましょう。 ログイン画面が表示されるはずです。

ログイン中のユーザの取得

マイページにはログインしているユーザの名前などの情報を表示することがよくあります。 ここではマイページにログインしているユーザのメールアドレスを表示してみましょう。
Devise ではヘルパーメソッドが利用できます。 ログインしているユーザを取得するには current_user というヘルパーメソッドを利用します。

app/views/mypage/index.html.erb

```
<h1>マイページ</h1>
<p>マイページ<p>
<p>あなたのメールアドレスは<%= current_user.email %>です</p>
<p><%= linkto 'ログアウト' sessions_path, mehtod: :delete %></p>
```

ログインしてからマイページにアクセスして確認してみましょう。 メールアドレスが表示されています。

デザインの変更

今まで見てきたログイン画面や登録画面は Devise のデフォルト画面です。 Devise がインストールされているディレクトリにあるファイルなので修正することはできません。

これらのデザインを変更する方法を見ていきましょう。
下記のコマンドで view ファイルを作成します。

```
$ rails generate devise:views
$ ls app/views/devise/
confirmations/  mailer/  passwords/  registrations/  sessions/  shared/
   unlocks/
```

各機能で利用している view ファイルが作成されました。

まずはログイン画面の view を開いて見ましょう。
ログイン画面の view は app/views/devise/sessions/new.html.erb です
H2 タグの部分を英語から日本語に修正してからログイン画面にアクセスしましょう。

今まで英語で表示されていた Log in が日本語でログインと表示されました。
このように Devise が提供している画面を修正するには devise:views のジェネレータを使って view ファイルを作成し、そのファイルを修正します。

テストの追加

rspec のインストール

rspec がインストールされていない場合はインストールしましょう

Gemfile

```
gem 'rspec-rails'
```

```
$ bundle install
$ rails g rspec:install
Running via Spring preloader in process 35280
      create  .rspec
      create  spec
      create  spec/spec_helper.rb
      create  spec/rails_helper.rb
```

Devise のテストを行うには Request spec を利用します。

ログアウトした状態でのテスト

ログアウトした状態でのテストは特に何も必要ありません。 いつも通り request spec を書けば良いです。

spec/requests/mypage_spec.rb

```
require 'rails_helper'

RSpec.describe 'Mypage', type: :request do
  describe 'GET /mypage' do
    context 'ログインしていない場合' do
      it 'ログイン画面へリダイレクトすること' do
        get mypage_path
        expect(response).to redirect_to(:new_user_session)
      end
    end
  end
end
```

ログインした状態でのテスト

ログインしたい状態のテストを行うにはログインした状態を作ってから request を投げる必要があります。
ログイン状態を作るための設定を行います。
spec/rails_helper.rb

```
  # Devise
  config.include Warden::Test::Helpers
  config.before :suite do
    Warden.test_mode!
  End
```

spec ファイルを修正します。

```
spec/request/mypage_spec.rb
```

```
require 'rails_helper'

RSpec.describe 'Mypage', type: :request do
  describe 'GET /mypage' do
    context 'ログインしていない場合' do
      it 'ログイン画面へリダイレクトすること' do
        get mypage_path
        expect(response).to redirect_to(:new_user_session)
      end
    end

    context 'ログインしている場合' do
      let(:user) { User.create(email: 'test@example.com', password:
    'password') }

      before do
        login_as user
      end

      it 'マイページが表示されること' do
        get mypage_path
        expect(response.body).to include 'ここはマイページです'
      end
    end
  end
end
```

これでログインしている時のマイページのテストとログインしていない時のマイページのテストができました。

8.7.6 問題

それでは、今作成している EC サイトの商品一覧に認証を追加してみましょう。また、新たにユーザの CRUD を作成し認証を追加しましょう。

8.8 セッション管理

アプリケーションでは、セッションを用いてクライアント側で情報を保存することができます。例えば、セッションで保存したユーザー情報で、そのユーザー固有の状態を維持しデータのやりとりを行うことが可能になります。

多くのアプリケーションでは、まず「ユーザー登録」を行いユーザー情報をデータベースに保存します。次回利用する際に、その登録されたユーザーで「ログイン認証」を行いアプリケーションにログインした状態を管理することができます。「ログインした状態」はセッションを用いて実現します。 ショッピングサイトの場合は、現在ログインしているユーザーの「カート機能」もセッションを利用して実現しています。 セッションを使わずにリクエストごとにデータベースへアクセスしユーザー認証を行うのは現実的ではありません。

同時にセッション情報が盗まれるなどのリスクも生まれます。セキュリティに不備があるネットワークでは、セッション ID を他人に盗まれる危険があるので SSL によって暗号化し安全な通信を行う必要があります。

8.8.1 セッションストア

前回のリクエスト情報を再利用するため少量のデータをセッションデータとして保存し、次回のリクエストに使用します。 Rails には、以下のようなセッションストアが用意されておりストレージを選択して、セッション情報を保存することができます。 セッションは Contoller と View で利用できます。 デフォルトでは CookieStore が使われます。 Rails の推奨セッションストアである CookieStore はセッションを利用するための準備が不要であり、データが暗号化されているので 安全に利用できます。ただし CookieStore に保存できるデータ量は約 4KB と少量なので、セッションに大量のデータを保存できません。

セッションストア一覧

ストレージ	概要
ActionDispatch::Session::CookieStore	すべてのセッションをクライアント側のブラウザの cookie に保存する
ActionDispatch::Session::CacheStore	データを Rails のキャッシュに保存する

ActionDispatch::Session::ActiveRecordStore	Active Record を用いてデータベースに保存する（activerecord-session_store gem が必要）
ActionDispatch::Session::MemCacheStore	データを memcached クラスタに保存する（この実装は古いので CacheStore を検討すべき）

8.8.2 セッションにアクセスする

セッションへのアクセスは session メソッドを使用します。

セッションにデータを保存する

セッションには、ハッシュのように「キー」と「値」がペアで保存されます。

```
def create
  current_user = User.find_by(name: params[:name], password: params[:password])
  session[:user_id] = current_user.id
end
```

セッションのデータを削除する

delete メソッドを使用し引数にキーを指定します。

```
def destroy
  session.delete[:user_id]
end
```

セッションの内容を全てリセットする

`reset_session` でセッション全体をリセットします。

8.8.3 Flash を使用する

Rails では、`flash` を使用し画面上部に赤色でエラーメッセージを表示させる場合があります。 この `flash` もセッションの一部です。`flash` は、リクエストごとにクリアされるので処理直後のリクエストでのみ使用するデータを使用するのみ便利です。 ユーザー登録処理やログイン/ログアウト処理などさまざまなアクションで使用することで、次のリクエストに表示するメッセージを送ることができます。 ログアウトの処理を例に見てみ

ましょう。 ログアウトが成功すれば、次の画面で「ログアウトしました」というメッセージが表示されるというわけです。

```
class SessionController < ApplicationController
  def destroy
    session.delete[:current_user_id]
    flash[:notice] = "ログアウトしました"
    redirect_to root_url
  end
end
```

コラム：セキュリティの注意点

セキュリティに関しては、新しい欠陥（セキュリティホール）が見つかったり、それを利用して秘密情報を盗む手順が考えられたりします。プログラムの作り方によって、思いも寄らないセキュリティホールを作ってしまったり、セッションの中に秘密情報を記録してしまったりすることがあります。Rails でのセキュリティに対する記述の仕方、考え方を理解するためや、自分の知識を再確認するためにも以下のサイトを一読されることをお勧めします。

Rails セキュリティガイド
https://railsguides.jp/security.html

8.9 Ruby on Rails：画像アップロード

8.9.1 画像アップロード

前項のデザインテンプレートに商品の写真を載せる部分がありますので、画像アップロード機能を追加しましょう。まず、アプリの管理者が商品の画像データをアップロードします。そのアップロードされた画像データをデザインテンプレートへ表示できるように変更していきます。

ここでは、ブラウザから HTTP(HTTPS も含む)で送られてきたデータが、Rails の内部でどのようにして画像ファイルとして扱われるかを少し説明します。そのあと、画像アップロードの機能は便利な gem がありますので、そちらを利用して簡単に実装してしまいましょう。

8.9.2 Rails における画像アップロードの概要

(a) 解説

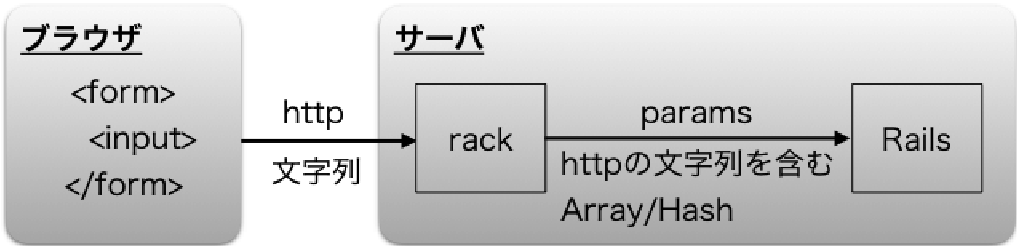
ブラウザ ～ サーバ

ブラウザは、html の form タグと input タグを組み合わせて利用することで、様々な文字情報をサーバーに送ることができます。画像ファイルも同じです。input タグの type を file にすると、ファイルを選択できるようになります。そこで選択されたファイルは、バイナリデータとしてサーバーへ送られます。このとき、データは HTTP のプロトコルに乗っ取った表現になっていますが、あくまでそれらは大きな文字列のかたまりです。例えば、html として "2017-12-31" という部分があったとしても、 それを Ruby として判断すると String であることに気をつけてください。

サーバ ～ Rails のコントローラ

サーバーに到着した HTTP のデータは、Rails アプリへ直接渡されるのではなく、rack というミドルウェアを経由します。Rails アプリは rack に依存しており、rack はどこでも必ず必要です。rack の役割はいろいろありますが、HTTP データについては、HTTP の文字列のかたまりを、要素別に Ruby の Array または Hash に整形してくれます。これが、コントローラで扱う params の元ネタです。 rack で Array や Hash に整形されたあとは、ActionDispatch::Request::Utils でさらに分析され、ファイルの場合は ActionDispatch::Http::UploadedFile でファイル名やファイルのバイナリ、ファイルサイズなどを取り出しやすいように file オブジェクトへ格納します。 そうして、HTTP か

らのデータをコントローラで受け取るときには、params という扱いやすい形式になっています。画像データも同様に、file オブジェクトに格納されたものから保存したいものを取り出し、同様に .save すれば、ファイルアップロードの機能を実現することができます。



8.9.3 Active Storage を利用した画像アップロード(Rails5.2 から)

Active Storage は Rails 5.2 から利用できる gem です。
この課題を始める前に、Rails 5.2 をインストールします。

```
gem install rails -v 5.2.3
```

Active Storage

1. Active Storage を導入するために新規プロジェクトを作成します

```
bash     $ rails _5.2.3_ new storage
```

2. マイグレーションファイルを作成します

```
bash     $ cd storage     $ rails active_storage:install
```

・active_storage_blobs テーブルと active_storage_attachments テーブルのマイグレーションファイルが作成されます

3. マイグレーションファイルの内容をデータベースに反映する

```
bash    $ rails db:migrate
```

4. 雛形を作成します

User モデルに name カラムを設定

```
$ rails generate scaffold user name:string
```

5. User モデルをデータベースに反映します

```
bash    $ rails db:migrate
```

6. テーブル同士の関連付けを行います

- _app/models/user.rb

```
class User < ApplicationRecord
    has_one_attached :photo
end
```

7. Strong Parameter を設定します

- _app/controllers/users_controller.rb

```
def user_params
    params.require(:user).permit(:name, :photo)
end
```

8. view ファイルに画像をアップロードする項目を追加します

- app/views/users/_form.html.erb

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

  <div class="field">
    <%= form.file_field :photo %>
  </div>
```

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

9. ユーザーの詳細画面にアップロードした画像を表示するよう設定します

- app/views/users/show.html.erb

```
<p>
  <strong>Name:</strong>
  <%= @user.name %>
</p>

<p>
  <strong>Photo:</strong>
  <% if @user.photo.attached? %>
    <%= image_tag @user.photo %>
  <% end %>
</p>
```

※実際に画像をアップロードして確認してみましょう。

RubyLogo

8.9.3 画像のリサイズ

Active Storage を利用して表示する画像のリサイズをするために ImageMagick を利用します。
ImageMagick は、画像のリサイズや画像フォーマットの変換、画像の加工、GIF アニメーションの作成などもできる画像処理のコマンドです。

Cloud9 環境に ImageMagick をインストール

ここでは、Cloud9 環境に ImageMagick をインストールする手順を説明します。

他の環境への ImageMagick のインストール手順は、
https://imagemagick.org/script/download.php を参考にしてください。

```
$ sudo yum -y install libjpeg-devel libpng-devel ImageMagick-devel
    ImageMagick
```

gem の追加

Gemfile に mini_magic を追加しましょう。コメントになっている場合は、コメントを外しましょう。

MiniMagick は Rails から ImageMagick を利用できるようにする gem です。
Gemfile の変更ができたら、bundle install を実行しましょう。

```
$ bundle install
```

モデルにリサイズするメソッドを定義

1. 大きめの画像がアップロードされた場合でも、一定の大きさで表示されるようにリサイズします。

Active Storage では、アップロードされた画像はそのまま保存し、表示する時にリサイズをします。 画像の大きさを縦横 300x300 ピクセルに収まる画像として表示するようにモデルにメソッドを追加します。
app/models/user.rb

```
(省略)
def thumbnail
  photo.variant(resize: '300x300')
end
(省略)
```

リサイズされた画像の確認

1. リサイズした画像を表示できるように View ファイルを修正します。

追加したリサイズのメソッドを利用して画像を表示するようビューを修正します。 ここでは、元の画像と比較できるように、`@user.thumbnail` を利用して表示できるように処理を追加しましょう。

`app/views/users/show.html.erb`

```
(省略)
<p>
  <strong>Photo:</strong>
  <% if @user.photo.attached? %>
    <%= image_tag @user.photo %>
    <%= image_tag @user.thumbnail %>
  <% end %>
</p>
(省略)
```

2. ブラウザで show をクリックして画像を確認します。

アップロードした画像が指定のサイズ(300x300)で表示されれば OK です。

8.9.4 アップロード機能の自動テスト

1. アップロードできるファイルの種類と大きさを制限して、その制限が機能しているかテストを作成しましょう。

まずは、アップロードされる画像に対してのバリデーション条件を記述しましょう。

`app/models/user.rb`

```
class User < ApplicationRecord
  has_one_attached :photo

  # (1)photo に対するバリデーションを適用
  validate :validate_photo

  # (2)photo に対するファイルサイズの制限と、ファイルの種類の制限
  def validate_photo
    return unless photo.attached?
    if photo.blob.byte_size > 2.megabytes
      photo.purge
      errors.add(:photo, 'File too large.')
    elsif !allowed_image_type?
      photo.purge
      errors.add(:photo, 'File type not allowed.')
    end
  end

  private

    # (3)ここでは JPEG 形式の画像のみを受け入れ可能とします
    def allowed_image_type?
      %w[image/jpg image/jpeg].include?(photo.blob.content_type)
    end
end
```

2. バリデーションの確認

実際に、2MB 以上の画像をアップロードしたり、PNG 形式などの JPEG 形式とは異なる画像をアップロードしてバリデーションが機能していることを確認しましょう。

3. テストの実装

Gemfile に RSpec に必要な gem を追加します。

```
group :development, :test do
  # Use RSpec
  gem 'rspec-rails', '~> 3.6'
  # Use FactoryBot
  gem 'factory_bot_rails'
end
```

bundle install も忘れずに実行します。

```
$ bundle install
```

RSpec のインストールも忘れずに実行します。

```
$ rails generate rspec:install
```

config/application.rb に RSpec を利用する設定を記述します。

```
module RspecMockups
  class Application < Rails::Application
    # Don't generate system test files.
    config.generators.system_tests = nil
  end
end
```

config/initializers/generators.rb を新規作成して、以下の内容を記述します。

```
Rails.application.config.generators do |g|
 g.test_framework :rspec
 g.view_specs false
 g.routing_specs false
 g.helper_specs false
 g.fixture_replacement :factory_bot, dir: 'spec/factories'
end
```

User モデルに対する spec ファイルを作成します。

```
$ rails generate rspec:model user
```

spec/models/user_spec.rb にテストを記述します。

```
require 'rails_helper'

RSpec.describe User, type: :model do
  it "is valid content_type with jpg, jpeg" do
    formats = %w(jpg jpeg)
    formats.each do |format|
      user = User.new( name: "test" )
      user.photo.attach(io: File.open("spec/fixtures/photo.#{format}"),
                        filename: "photo.#{format}", content_type:
   'image/#{format}')
      expect(user).to be_valid
    end
  end

  it "is invalid content_type with png" do
    formats = %w(png)
    formats.each do |format|
      user = User.new( name: "test" )
      user.photo.attach(io: File.open("spec/fixtures/photo.#{format}"),
                        filename: "photo.#{format}", content_type:
   'image/#{format}')
      expect(user).not_to be_valid
    end
  end

  it "is invalid to large file" do
    user = User.new( name: "test" )
```

```
    user.photo.attach(io: File.open("spec/fixtures/large-photo.jpg"),
                      filename: "large-photo.jpg", content_type:
   'image/jpg')
    expect(user).not_to be_valid
  end
end
```

テストに必要なファイルを準備します。

```
spec/fixtures/photo.jpg (ファイルサイズが 2MB 以下の JPEG 形式画像)
spec/fixtures/photo.jpeg (ファイルサイズが 2MB 以下の JPEG 形式画像,photo.jpg の
   コピーでも OK)
spec/fixtures/photo.png (許可されていない PNG 形式画像)
spec/fixtures/large-photo.jpg (ファイルサイズが 2MB より大きな JPEG 形式画像)
```

テストを実行します。

```
$ rspec spec/models/user_spec.rb
2019-11-20 13:45:56 WARN Selenium [DEPRECATION]
   Selenium::WebDriver::Chrome#driver_path= is deprecated.
Use Selenium::WebDriver::Chrome::Service#driver_path= instead.
...

Finished in 0.55947 seconds (files took 5.28 seconds to load)
3 examples, 0 failures
```

とエラーがゼロ件なら、テストは成功です。
今度は、ファイルサイズの制限を large-photo.jpg の画像のファイルサイズより大きく(例:10MB)、また、受け入れ可能な画像形式に image/png を追加してみましょう。

app/models/user.rb

```
class User < ApplicationRecord
  (省略)
  def validate_photo
    return unless photo.attached?
    if photo.blob.byte_size > 10.megabytes
      photo.purge
      errors.add(:photo, 'File too large.')
    elsif !allowed_image_type?
      photo.purge
      errors.add(:photo, 'File type not allowed.')
    end
  end

  private

    def allowed_image_type?
```

```
        %w[image/jpg image/jpeg
      image/png].include?(photo.blob.content_type)
      end
    end
```

テストを実行します。

```
$ rspec spec/models/user_spec.rb
2019-11-20 13:49:43 WARN Selenium [DEPRECATION]
    Selenium::WebDriver::Chrome#driver_path= is deprecated.
Use Selenium::WebDriver::Chrome::Service#driver_path= instead.
.FF

Failures:

  1) User is invalid with png
    Failure/Error: expect(user).not_to be_valid
      expected #<User id: nil, name: "test", created_at: nil, updated_at: nil>
    not to be valid
    # ./spec/models/user_spec.rb:18:in block (3 levels) in <top (required)>'
    # ./spec/models/user_spec.rb:15:in each'
    # ./spec/models/user_spec.rb:15:in block (2 levels) in <top (required)>'

  2) User is invalid to large file
    Failure/Error: expect(user).not_to be_valid
      expected #<User id: nil, name: "test", created_at: nil, updated_at: nil>
    not to be valid
    # ./spec/models/user_spec.rb:25:in block (2 levels) in <top (required)>'

Finished in 0.71279 seconds (files took 6.11 seconds to load)
3 examples, 2 failures

Failed examples:

rspec ./spec/models/user_spec.rb:13 # User is invalid with png
rspec ./spec/models/user_spec.rb:22 # User is invalid to large file
```

と、２件のエラーが検出されると期待通りの動作です。ですが、コードのバランスがとれていませんので、修正した箇所は、元に戻してもかまいませんし、`app/models/user.rb` の変更をそのままにするのであれば、テストが正常に終了するように `spec/models/user_spec.rb` を修正しておきましょう。

8.10 [評価課題]EC サイトの開発 1 の課題実施の評価

ここで、CRUD とルーティングで学んできたことのポイントを確認してみましょう。

Rails アプリケーションで CRUD とは、

データを新規作成する(Q1:　　　　　　　　　　)、表示する(Q2:　　　　　　　　　　)、更新する(Q3:　　　　　　　　　　)、削除の(Q4:　　　　　　　　　　)をいいます。

以下のコントローラを完成させてください。

```
app/controllers/users_controller.rb
  class UsersController < ApplicationController
    before_action :set_user, only: [:show, :edit, :update, (Q5:         )]

    def index
      (Q6:             )
    end

    def show
    end

    def new
      (Q7:             )
    end

    def edit
    end

    def create
      @user = User.new(Q8:             )

      respond_to do |format|
        if @user.(Q9:             )
          format.html { redirect_to @user, notice: t('.success') }
          format.json { render :show, status: :created, location: @user }
        else
          format.html { render :new }
          format.json { render json: @user.errors,
    status: :unprocessable_entity }
        end
      end
    end

    def update
      respond_to do |format|
        if @user.(Q10:             )
          format.html { redirect_to @user, notice: t('.success') }
          format.json { render :show, status: :ok, location: @user }
```

```
      else
        format.html { render :edit }
        format.json { render json: @user.errors,
  status: :unprocessable_entity }
      end
    end
  end

  def destroy
    @user.(Q11:                 )
    respond_to do |format|
      format.html { redirect_to users_url, notice: t('.success') }
      format.json { head :no_content }
    end
  end

  # クラス内でのみ使用します
  (Q12:                 )
    # Use callbacks to share common setup or constraints between actions.
    def set_user
      @user = User.find(params[:id])
    end

    # Never trust parameters from the scary internet, only allow the white
  list through.
    # 安全に User.new に利用できる属性を制限します
    def user_params
      params.require(:user).(Q13:                )(:name, :phone_number)
    end
end
```

アプリケーションで利用できるルーティングは

```
$ rails (Q14:                     )
```

を実行して確認します。
以下に表示されているルーティングでは

```
   Prefix Verb   URI Pattern                Controller#Action
    users GET    /users(.:format)           users#index
          POST   /users(.:format)           users#create
 new_user GET    /users/new(.:format)       users#new
edit_user GET    /users/:id/edit(.:format)  users#edit
     user GET    /users/:id(.:format)       users#show
          PATCH  /users/:id(.:format)       users#update
          PUT    /users/:id(.:format)       users#update
          DELETE /users/:id(.:format)       users#destroy
```

user を新規に作成する画面への URI Pattern は(Q15:　　　　　　)、新規登録への URI Pattern は(Q16:　　　　　　)、user を一覧表示する Prefix は(Q17:　　　　　　)、id

が 1 の user を編集する URI は(Q18:)、id が 2 の user を削除する URI は (Q19:)で、method に(Q20:)を指定してアクセスします。

第９章　Ruby on Rails ECサイトの開発２

第９章　Ruby on Rails EC サイト開発２

9.1 Ruby on Rails： EC サイトの開発 注文

9.1.1 確認画面

通常、入力画面と完了画面の間には、入力した内容を確認する画面があります。 しかし、Rails の scaffold で作成したものには確認画面は存在しません。

ここでは確認画面の作成方法を説明します。

9.1.2 例題

scaffold を利用せずに記事入力、確認、完了の流れを作成します。

ルーティングの作成

下記のようにルーティングを追加します。

今回は登録、確認、完了と一覧、詳細です。

Ruby on Rails のルーティングは、'resouces'メソッドを使うことで `index` `show` `new` `create` `edit` `update` `destroy` の７つの基本のルーティングが定義されます。 しかし、今回の確認画面のように、基本の７つ以外のルーティングが必要な場合があります。

その際に使うのが `member` と `collection` です。

`member` はある１つのリソースに対しての操作を行う場合に利用します。 collection は特定のリソースが決まっていない場合に利用します。

member の例

```
resources :user do
  member do
    get :avatar
```

```
  end
end
```

このように定義すると/users/(:id)/avatarという URL 定義され UsersController#avatar アクションが呼びだされます。 その際 params[:id]にはアクセスした URL で指定した(:id)の値がセットされます。

```
 @user = User.find(params[:id])
```

このようにして、特定のリソースに対して処理を行います。

collection の例

```
resources :user do
  collection do
    get :search
  end
end
```

このように定義すると/users/searchという URL が定義され UsersController#search アクションが呼び出されます。
最初は、member は id が必要、collection は id が不要と覚えておきましょう。
今回必要な確認画面は、まだ注文は確定していないため id は持っていません。 そのため、member ではなく collection を使いルーティングを定義します。

config/routes.rb

```
resources :articles, only: [:index, :new, :create, :show] do
  collection do
    post :confirm
  end
end
```

余談ですが、変更の確認画面が必要になった場合は、member を利用します。

モデルの作成

モデルを作成するのは Rails のジェネレータを利用します。

```
$ rails g model Article title:string content:text
```

上記のコマンドを実行すると、モデルに必要なファイルが作成されます。 ここでマイグレーションを実行します。

```
$ rails db:migrate
```

これでモデルの作成は完了です。

入力画面の作成

入力画面を作成します。

コントローラに入力画面のアクションを追加します。
app/controllers/articles_controller.rb

```
class ArticlesController < ApplicationController
  def new
    @article = Article.new
  end
end
```

次に view を作成します。
app/views/articles/new.html.erb

```
<h1>New Article</h1>

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

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

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

    <div class="field">
      <%= form.label :content %>
```

```
      <%= form.text_area :content, id: :article_content %>
    </div>

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

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

scaffold で作成した場合の `form_with` は自動的に完了画面に遷移するようになっていました。今回は確認画面に遷移させたいので `url: confirm_articles_path` を追加します。

確認画面の作成

コントローラに確認画面のようのアクションを追加します。

`app/controllers/articles_controller.rb`

```
class ArticlesController < ApplicationController
  def confirm
    @article = Article.new(article_params)
  end

  private

  def article_params
    params.require(:article).permit(:title, :content)
  end
end
```

view を作成します。

```
<h1>Confirm New Article</h1>

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

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

    <div class="field">
```

```
      <p><%= form.label :title %></p>
      <%= @article.title %>
      <%= form.hidden_field :title, id: :article_title %>
    </div>

    <div class="field">
      <p><%= form.label :content %></p>
      <%= @article.content %>
      <%= form.hidden_field :content, id: :article_content %>
    </div>

    <div class="actions">
      <%= form.submit '戻る', name: :back %>
      <%= form.submit '登録' %>
    </div>
<% end %>


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

確認画面の view では 2 つの submit ボタンを表示します。 片方の name 属性を submit に、もう一方の name 属性を back に設定します。 次の完了の処理でこの name 属性の値をみて、確定ボタンがクリックされたのか、戻るボタンがクリックされたのかを判断するために利用します。

完了画面の作成

コントローラに完了用のアクションを追加します。

app/controllers/articles_controller

```
class ArticlesController < ApplicationController
  def create
    @article = Article.new(article_params)

    if params[:back].present?
      render :new
    else
      if @article.save
        redirect_to articles_path, notice: '登録が完了しました。'
      else
        render :new
      end
    end
  end
end
```

ここでは確認画面で設定した submit ボタンの名前によって処理を分岐させます。 name 属性に値がセットされている場合は戻るボタンがクリックされたと判断出来ます。 その場合は new を render します。 そうでない場合は、確定ボタンがクリックされたので登録処理を行います。

一覧画面の作成

最後に登録した Article を表示する一覧画面を作成しましょう

app/controllers/articles_controller.rb

```
class ArticlesController < ApplicationController
  def index
    @articles = Article.all
  end
end
app/views/articles/index.html.erb
<p id="notice"><%= notice %></p>

<h1>Articles</h1>

<table>
  <thead>
  <tr>
    <th>Title</th>
    <th>Content</th>
  </tr>
  </thead>

  <tbody>
  <% @articles.each do |article| %>
      <tr>
        <td><%= article.title %></td>
        <td><%= article.content %></td>
      </tr>
  <% end %>
  </tbody>
</table>

<br>

<%= link_to 'New Article', new_article_path %>
```

これで入力、確認、登録、一覧の流れができました。 実際に操作して確認してみましょう。

レイアウトの変更

今作成中の EC サイトには、すでに管理者がアクセスできる商品一覧があります。

それとは別に、買い物をする利用者がアクセスするための商品一覧も必要になります。

しかし、今までどおり scaffold で作成すると、管理側の商品一覧も利用者側の商品一覧も同じレイアウトになってしまいます。 通常 EC サイトでは利用者側のページは管理者側よりも見栄えの良いものになります。

そのためには利用者側は別のレイアウトを使う必要があります。

レイアウトを変更する方法はいくつかあります。順番に見ていきましょう

アクションごとにレイアウトを変更する

アクションごとに変更する場合は render でレイアウト指定します。

```
def index
  @users = User.all
  render layout: 'hoge'
end
```

このように指定すると `/app/views/layout/hoge.html.erb` のレイアウトを使用します。
コントローラごとにレイアウトを変更する
コントローラごとにレイアウトを変更する場合は、規約で決められたファイル名のファイルを `/app/views/layout` に置くだけです。
下記の順番に優先度が高くなっています。

- コントローラと同じファイル名
- コントローラが継承しているコントローラと同じファイル名
- ApplicationController と同じファイル名

layout/application.html.erb が使われているのは、アクションごとにもコントローラごとにもレイアウトが指定されていないので継承元である ApplicationController と同じファイル名の application.html.erb が使われていることになります。

```
class BaseController < ApplicationController
end

class UsersController < BaseController
  def index
  end
end
```

上記のような継承関係のコントローラの場合、 `/app/views/layout/users.html.erb` のレイアウトが使われます。
なければ、 `/app/views/layout/base.html.erb` のレイアウトが使われます。
それもなければ、 `/app/views/layout/application.html.erb` が使われます。

9.1.3 問題

注文画面の実装

EC サイトに下記の操作ができるように注文機能を実装しましょう。 管理者以外のユーザが閲覧するページを products コントローラ、注文を保存するテーブルを Order という名称で作成してください。

- 利用者が見る商品一覧画面と詳細画面を作成
- 商品詳細画面に注文ボタンを作成
- 注文ボタンをクリックすると、個数、届け先の入力フォーム、次へ進むのボタンを表示
- 次へ進むをクリックすると、注文した商品、入力した内容、注文確定ボタンを表示
- 注文確定ボタンをクリックすると、注文データの作成し、注文完了画面を表示

9.1.4 テストの追加

ここでは、

- 商品詳細画面に購入ボタンが表示されている
- 購入ボタンをクリックすると、届け先の入力フォームや購入確定ボタン等が表示される
- 購入確定ボタンをクリックすると購入完了画面が表示される

という動作をする注文画面について、

- 購入ボタンをクリックすると、購入情報入力画面が表示できる

● 購入確定ボタンをクリックすると、購入完了画面が表示できる

ことを確認する E2E のテストを作成していきます。

E2E テストの準備

テストコードを記述する products_spec.rb を作成します。

```
touch spec/system/products_spec.rb
```

それから、今回のテストで使用するテストデータを FactoryBot に用意しておきます。

spec/factories/products.rb

```
FactoryBot.define do
  factory :product do
    type { "Book" }
    title { "テスト用本" }
    author { "テスト太郎" }
    published_on { "1981-09-01" }
    showing { true }
    price { 1020 }
  end
end
```

購入ボタンをクリックすると、購入情報入力画面が表示できる

それでは、実際に画面の作成と、テストを記述していきましょう。

まずは、一覧画面とそのテストから記述します。

views/products/index.html.erb

```
<div class="row marketing">
  <h2>BOOK</h2>
  <div class="col-lg-12">
    <% @books.each do |book| %>
      <h3><%= book.title %></h3>
      <p><%= book.author %></p>
      <p><%= book.published_on %></p>
      <p>
```

```
        <%= number_to_currency(book.try(:price), precision: 0, unit: "円") %>
        <%= link_to '購入', new_order_path(product_id: book.id), class: 'btn
    btn-default' %>
      </p>
    <% end %>
  </div>
</div>
```

views/orders/new.html.erb

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

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

  <h2 class="sub-header">商品</h2>

  <p>
    <strong>タイトル：</strong>
    <%= product.title %>
  </p>

  <p>
    <strong>金額：</strong>
     <%= number_to_currency(product.try(:price), precision: 0, unit: "円") %>
  </p>

  <div class="form-group">
    <%= f.label :shipping_address %>
    <%= f.text_field :shipping_address, class: "form-control" %>
  </div>

  <%= f.hidden_field :product_id, value: product.id %>

  <div class="actions">
    <%= f.submit '購入確定', class: "btn btn-default" %>
  </div>
<% end %>
...
```

spec/system/products_spec.rb

```
require 'rails_helper'

RSpec.describe "Products", type: :system do
  describe "GET /products/index" do
    it "renders order_new" do
      book = FactoryBot.build(:product)
      visit products_index_path
      click_link "購入"
      assert_text "#{book.title}"
      assert_text "商品購入"
    end
  end
end
```

購入確定ボタンをクリックすると、購入完了画面が表示できる

続いて、購入情報入力画面とそのテストを記述します。

app/controllers/orders_controller.rb

```
...
  def create
    @order = Order.new(product_id: order_params[:product_id], shipping_address:
   order_params[:shipping_address]).save

    respond_to do |format|
      format.html { redirect_to products_index_url, notice: 'Order was
   successfully created.' }
      format.json { render :show, status: :created, location: @order }
    end
  rescue
    respond_to do |format|
      format.html { render :new }
      format.json { render json: @order.errors, status: :unprocessable_entity }
    end
  end

  private
    def order_params
      params.require(:order).permit(:product_id, :shipping_address)
    end
...
```

spec/system/products_spec.rb

```
require 'rails_helper'
```

```
RSpec.describe "Products", type: :system do
  ...

  describe "GET /orders/new" do
    it "renders order_new" do
      book = FactoryBot.build(:product)
      visit products_index_path
      click_on "購入"
      fill_in "order_shipping_address", with: "大阪市"
      click_on "購入確定"
      assert_text "Order was successfully created."
    end
  end
end
```

9.1.5 まとめ

- 基本の７つ以外のルーティングが必要な場合は member や collection を利用しましょう。
- form_with に url を指定することで、submit 時に任意の URL に遷移できます。
- render でレイアウトを指定することで、レイアウトを変更できます。

9.2 Ruby on Rails：EC サイトの開発 メール送信

9.2.1 メール送信のプロトコルの概要

メールを送信するためには、一般的には SMTP を利用します。
SMTP(Simple Mail Transfer Protocol)の動作はおおよそ以下の流れになります。

- メールソフトや Rails がメールの送信者、受信者、タイトル、本文、添付ファイルなどのデータを組み立てます。
- メールソフトや Rails が組み立てたメールの情報を設定されているメールサーバに送信します。
- 送信側のメールサーバは、メールの送信者を認証してからメールを受け取ります。
- 送信側のメールサーバは、受け取ったメールの受信者のアカウントがある受信側のメールサーバに転送します。
- 受信側のメールサーバは、受け取ったメールの受信者がメールサーバに登録されているアカウントであるか確認し、有効であればアカウントのポストに保存します。

ここでは、送信しか取り扱いませんが、メールを受信するためには、一般的には POP(Post Office Protocol)もしくは IMAP(Internet Message Access Protocol)を利用します。

POP の動作はおおよそ以下の流れになります。

- メールソフトや Rails が設定されているメールサーバのポストに接続を要求します。
- メールサーバは要求されたアカウントとパスワードが有効であれば、ログインを許可します。
- メールサーバは要求してきたアカウントのポストに保存されているメールの件数やメールを返信します。

9.2.2 メール送信

ここでは、購入後にメールを送信する機能を実装していきます。 簡単に ActionMailer について説明し、Rails アプリケーションでのメール送信機能を実装します。

9.2.3 ActionMailer とは

ActionMailer とは、Rails のデフォルトの gem で、メールの送信や受信を行うことが可能となっています。 今回は、この ActionMailer でのメール送信について説明していきます。 ActionMailer を使用することで、Mailer クラスと View でメールを送信することができます。 これは、今まで作成してきた画面での Controller クラスと View の関係に似ています。

なお、メールの送信や受信には、別途メールサーバが必要になります。

最終的に送信先にメールを配送するのはメールサーバの担当になります。この動作は、一般的なメールソフトの動作と同じです。

ActionMailer では、送信するメールの形式として HTML 形式とテキスト形式が利用できますが、ここでは HTML 形式でメールを送信するよう実装します。

9.2.4 例題

簡単な ToDo リストアプリケーションを作成し、ToDo の登録時に、メールを送信できるように実装します。

① スケジュール登録アプリケーションの作成

まず最初に、例題用の Rails アプリケーションを作成します。
その後、generator の scaffold コマンドを使って、ToDo を保存する CURD を作成します。

```
$ rails new MyToDoTask
$ rails generate scaffold ToDoTask title:string
description:text alert_mail_address:string
Running via Spring preloader in process 2183
      invoke  active_record
      create    db/migrate/20170829090051_create_to_do_tasks.rb
      create    app/models/to_do_task.rb
      invoke    test_unit
      create      test/models/to_do_task_test.rb
      create      test/fixtures/to_do_tasks.yml
      invoke  resource_route
       route    resources :to_do_tasks
      invoke  scaffold_controller
      create    app/controllers/to_do_tasks_controller.rb
      invoke    erb
```

```
      create      app/views/to_do_tasks
      create      app/views/to_do_tasks/index.html.erb
      create      app/views/to_do_tasks/edit.html.erb
      create      app/views/to_do_tasks/show.html.erb
      create      app/views/to_do_tasks/new.html.erb
      create      app/views/to_do_tasks/_form.html.erb
      invoke    test_unit
      create      test/controllers/to_do_tasks_controller_test.rb
      invoke    helper
      create      app/helpers/to_do_tasks_helper.rb
      invoke      test_unit
      invoke    jbuilder
      create      app/views/to_do_tasks/index.json.jbuilder
      create      app/views/to_do_tasks/show.json.jbuilder
      create      app/views/to_do_tasks/_to_do_task.json.jbuilder
      invoke  test_unit
      create    test/system/to_do_tasks_test.rb
      invoke  assets
      invoke    coffee
      create      app/assets/javascripts/to_do_tasks.coffee
      invoke    scss
      create      app/assets/stylesheets/to_do_tasks.scss
      invoke  scss
      create    app/assets/stylesheets/scaffolds.scss
```

```
$ rails db:migrate
== 20170829090051 CreateToDoTasks: migrating ==============================
-- create_table(:to_do_tasks)
   -> 0.0008s
== 20170829090051 CreateToDoTasks: migrated (0.0009s) =====================
```

② Action Mailer クラスとメイラービューの作成

次に、Mailer クラスと View を作成していきます。今まで通り、rails generate コマンドを使用して作成できます。

```
$ rails generate mailer TodoTaskMailer registration_mail
Running via Spring preloader in process 2439
      create  app/mailers/todo_mailer.rb
      invoke  erb
      create    app/views/todo_mailer
      create    app/views/todo_mailer/registration_mail.text.erb
      create    app/views/todo_mailer/registration_mail.html.erb
      invoke  test_unit
      create    test/mailers/todo_mailer_test.rb
      create    test/mailers/previews/todo_mailer_preview.rb
```

コマンドの最後に指定した registration_mail が ToDo 登録時にメールを送信するメソッド名になります。

そして、registration_mail メソッドから呼び出されるビューが app/views/todo_mailer/registration_mail.text.erb と app/views/order_mailer/registration_mail.html.erb になります。 今までの Controller と View の関係と同じように、メソッド名と同じ View ファイルが作成されています。

ファイルが 2 種類作成されているのは、HTML 形式用とテキスト形式用の 2 つのフォーマットです。

③ Mailer クラスの編集

Mailer クラスに送信処理を記述していきます。 registration_mail メソッド内に最初から記述されている不要な処理は、削除してください。

app/mailers/todo_task_mailer.rb

```
def registration_mail(todotask)
  @todotask = todotask
  mail to: todotask.alert_mail_address, subject: 'ToDo 登録のお知らせ'
end
```

mail メソッドの to が送信先アドレス、subject が件名になります。

④ View の編集

View は実際に送信するメール本文になります。 ここでは、ToDo のタイトルと内容を表示するようにしています。

```
app/views/todo_task_mailer/registration_mail.html.erb
<!DOCTYPE html>
<html>
  <head>
    <meta content='text/html; charset=UTF-8' http-equiv='Content-Type' />
  </head>
  <body>
    <h1><%= @todotask.title %></h1>
      <p>
        <%= @todotask.description %>
      </p>
  </body>
</html>
```

app/views/todo_task_mailer/registration_mail.text.erb

```
<%= @todotask.title %>
===============================================
<%= @todotask.description %>
```

ActionMailer は 2 種類(HTML/テキスト)のフォーマットが存在すると、multipart/alternative 形式のメールを作成します。multipart/alternative についての説明は省略しますが、HTML 形式で表示できない環境の場合はテキスト形式で表示されるようになります。

⑤ Mailer の呼び出し

Mailer クラスとメール本文(View)が作成できました。実際にメールを送るために、registration_mail メソッドを呼び出す処理を追加しましょう。

ToDo 登録後に送信するメールなので、コントローラの Create アクションで、DB の登録完了後にメール送信するようにします。

app/controllers/to_do_tasks_controller.rb

```
# POST /to_do_tasks
# POST /to_do_tasks.json
def create
  @to_do_task = ToDoTask.new(to_do_task_params)

  respond_to do |format|
    if @to_do_task.save
      #以下の処理を追加
      TodoTaskMailer.registration_mail(@to_do_task).deliver

      format.html { redirect_to @to_do_task,
      notice: 'To do task was successfully created.' }
      format.json { render :show, status: :created, location: @to_do_task }
    else
      format.html { render :new }
      format.json { render json: @to_do_task.errors,
      status: :unprocessable_entity }
    end
  end
end
```

⑥ メールのプレビュー

Action Mailer Previews の機能で、メールの本文を確認する事が可能です。 以下のようにソースコードを変更し、下記のアドレスにアクセスし確認してみましょう。

http://localhost:3000/rails/mailers/todo_task_mailer/registration_mail

test/mailers/previews/todo_task_mailer_preview.rb

```
# Preview all emails at http://localhost:3000/rails/mailers/todo_task_mailer
class TodoTaskMailerPreview < ActionMailer::Preview

  # Preview this email at
    http://localhost:3000/rails/mailers/todo_task_mailer/registration_mail
  def registration_mail
    #以下のように変更
    TodoTaskMailer.registration_mail(ToDoTask.new(title:"テスト",description:"プレビュー テスト"))
  end

end
```

⑦ ActionMailer の設定

最後に ActionMailer の設定を行います。

各環境ファイルごとに設定が必要になります。 下記に記載しているのは、development 環境になります。本番環境で使用するには config/environments/production.rb の編集が必要です。 今回はメール送信に、Gmail を使用しています。もし、設定を編集してもメールが届かない場合は、Gmail 側の設定を確認してください。Gmail はセキュリティ対策のため、安全性が低いとみなしたアプリからのアクセスを拒否するようにデフォルトで設定されています。
（参考：https://support.google.com/a/answer/6260879?hl=ja 「安全性の低いアプリからのアカウントへのアクセスを許可、拒否する」）

```
config/environments/development.rb
config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = {
  enable_starttls_auto: true,
  address: 'smtp.gmail.com',
  port: '587',
  domain: 'smtp.gmail.com',
  authentication: 'plain',
  user_name: ENV['MAIL_USER_NAME'],
  password: ENV['MAIL_PASSWORD'],
```

```
}
```

user_name と password の部分が、ENV['MAIL_USER_NAME']や ENV['MAIL_PASSWORD'] になっています。 これはユーザ名やパスワードをそのまま設定しているとセキュリティ的に危険なため、環境変数に設定して使用します。
開発環境(Cloud9)

```
$ export MAIL_USER_NAME="monkaec@gmail.com"
$ export MAIL_PASSWORD="monkaec123"
```

本番環境(Heroku)

```
$ heroku config:add MAIL_USER_NAME="monkaec@gmail.com"
Setting MAIL_USER_NAME and restarting ● sample-monka-ec2... done, v3
MAIL_USER_NAME: monkaec@gmail.com
$ heroku config:add MAIL_PASSWORD="monkaec123"
Setting MAIL_PASSWORD and restarting ● sample-monka-ec2... done, v4
MAIL_PASSWORD: monkaec123
```

これで、メール送信できるようになったはずです。ToDo の登録を通知するメールアドレスを自分のアドレスなどに変更して、メールが届くか確認してみましょう。

9.2.5 問題

例題を参考に、商品の購入後にメールを送信する機能を実装してみましょう。

9.3 [評価課題]EC サイトのメール送信課題

ここで、メール送信で学んできたことのポイントを確認してみましょう。

メール送信に利用するプロトコルは(Q1:)です。
以下のコマンドでメイラーを作成する場合、
$ rails (Q2:) (Q3:) TodoTaskMailer
とコマンドを入力します。
以下のメソッドで、メールアドレス todotask.alert_mail_address に件名'お知らせ'を送信する場合のコードを完成しましょう(コロンも忘れずに)。

```
app/mailers/todo_task_mailer.rb
def registration_mail(todotask)
  @todotask = todotask
  (Q4:              ) (Q5:                ) todotask.alert_mail_address,
    (Q6:              ) 'お知らせ'
end
```

以下の HTML 形式のメール本文を完成しましょう(セミコロンも忘れずに)。

```
app/views/todo_task_mailer/registration_mail.html.erb
<!DOCTYPE html>
<html>
  <head>
    <meta content='(Q7:               ) charset=UTF-8' http-equiv='Content-
    Type' />
  </head>
  <body>
    <h1><%= @todotask.title %></h1>
      <p>
        <%= @todotask.description %>
      </p>
  </body>
</html>
```

以下のコントローラで、Create アクションで DB への登録が成功した後、メールを送信する場合に
TodoTaskMailer.registration_mail(@to_do_task).deliver
の処理を追加する場所は以下の(A)から(D)のうち(Q8:)

```
app/controllers/to_do_tasks_controller.rb
# POST /to_do_tasks
# POST /to_do_tasks.json
def create
  @to_do_task = ToDoTask.new(to_do_task_params)

  respond_to do |format|
    if @to_do_task.save
      (A)
      format.html { redirect_to @to_do_task,
      notice: 'To do task was successfully created.' }
      format.json { render :show, status: :created, location: @to_do_task }
      (B)
    else
      (C)
```

```
      format.html { render :new }
      format.json { render json: @to_do_task.errors,
      status: :unprocessable_entity }
    end
    (D)
  end
end
```

以下のメール送信に必要な設定ファイルを完成しましょう。ただし、メール送信に必要なユーザー名と、パスワードは OS の環境変数に MAIL_USER_NAME と MAIL_PASSWORD に設定されているものとします。

```
config/environments/development.rb
config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = {
  enable_starttls_auto: true,
  address: 'smtp.gmail.com',
  port: '587',
  domain: 'smtp.gmail.com',
  authentication: 'plain',
  user_name: (Q9:              ),
  password: (Q10:            ),
}
```

第１０章　Ruby on Rails ECサイトの開発３

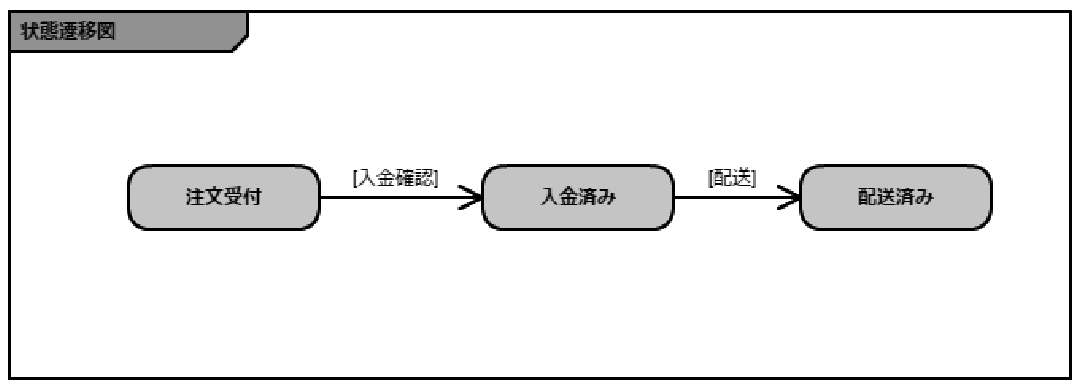
第 10 章　Ruby on Rails EC サイト開発３

10.1 Ruby on Rails：EC サイトの開発 enum/状態遷移 1

ここでは、状態遷移と、ActiveRecord::Enum の機能について説明していきます。

10.1.1 状態遷移とは

状態遷移とは、ある状態から特定のイベントで別の状態に切り替わることをいいます。



上の図は、EC サイトでの注文ステータスの遷移を表した簡単な図になります。 注文受付/入金済み/配送済みが状態で、入金確認/配送がイベントになります。
例えば、注文受付の状態から入金確認というイベントが発生することで、入金済みという状態に切り替わっています。
Rails には列挙型を表す ActiveRecord::Enum があります。列挙型はプログラミングの可読性を高めるためにとても役に立ちます。言語を問わずプログラミングでは数値はときに意味を持ちます。意味はコメントに記載されることは多いですが、長く運用されている場合はその意味が薄れていきます。
状態遷移を管理する場合、1 や 2 などの数字を直接記述する代わりに STACK や RENTED などの定数を利用して記述することで意味をつけることができますが、ActiveRecord::Enum を利用することで、意味にグループを持たせることができます。

10.1.2 例題

① ActiveRecord::Enum の実装

次の例では、Book クラスでは本の貸出状態（status）をインスタンス変数で持ち、「１」は書庫、「２」は貸出中を表します。

```
class Book
  def initialize
    @status = 1 # 「１」は書庫、「２」は貸出中。初期値は「１」
  end

  def status
    case @status
    when 1
      'stack' # 書庫であれば、'stack'
    when 2
      'rented' # 貸出中であれば、'rented'
    end
  end

  def rent!
    @status = 2 # 貸出中にする
  end

  def stack!
    @status = 1 # 書庫に入れる
  end
end
```

予約済を表したい場合はどのようにすればよいでしょうか？予約済を表す数字、対応する文字列、更新するメソッドなど考えることが多いですが、その効果は薄いです。例題では、ActiveRecord::Enum を利用して本の貸出状態を実装してみましょう。このモジュールは enum メソッドを提供し、状態遷移を簡単に実装することができます。また、ここでは RSpec を利用してテストも行うので、Gemfile に'rspec-rails'と'factory_bot_rails'を追加してください。

```
$ rails generate model Book title:string status:integer
```

貸出状態には貸出中（rented）、書庫（stack）の 2 つの状態があるとすると、book.rb は次のようになります。

```
class Book < ApplicationRecord
  enum status: [:rented, :stack]
end
```

ActiveRecord::Enum で状態遷移を定義することにより、status は型が Integer ですがプログラム上では文字列で意味を表現できます。さらに、プレフィックスが?のメソッドを提供してくれるので、プログラムの可読性が向上します。

```
book = Book.new
book.status = 'rented' # 貸出中

puts book.rented? #=> true
puts book.stack?  #=> false
```

貸出中（rented)、書庫（stack）には、0 と 1 がそれぞれ割り当てられるので RSpec のテストで確認してみましょう。

```
spec/models/book_spec.rb
  require 'rails_helper'

  RSpec.describe Book, type: :model do
    describe 'testing for book status' do
      context 'assign primitive value' do
        it "expects 0 to be mapped to 'rented'" do
          book = Book.create(status: 0)
          expect(book.status).to eq('rented')
        end
        it "expects 1 to be mapped to 'stack'" do
          book = Book.create(status: 1)
          expect(book.status).to eq('stack')
        end
      end

      context 'assign enum value' do
        it "expects 'rented' to be mapped to 0" do
          book = Book.create(status: 'rented')
          expect(Book.statuses[book.status]).to eq(0)
        end
        it "expects 'stack' to be mapped to 1" do
          book = Book.create(status: 'stack')
          expect(Book.statuses[book.status]).to eq(1)
        end
      end
    end
  end
```

また、検索クエリも状態名で実行できるようにメソッドが定義されます。RSpec を利用して動作を確認してみましょう。

```
spec/factories/books.rb
  FactoryBot.define do
    factory :rented_book, class: Book do
      sequence(:title) do |n|
        "坊っちゃん 第#{n}巻"
      end
      status { 0 }
    end
    factory :stack_book, class: Book do
      sequence(:title) do |n|
        "吾輩は猫である 第#{n}巻"
      end
      status { 1 }
    end
  end
spec/models/book_spec.rb
  require 'rails_helper'

  RSpec.describe Book, type: :model do
    describe 'testing for book status' do
      describe 'enum attributes are available for where clause' do
        before do
          FactoryBot.create_list(:rented_book, 7)
          FactoryBot.create_list(:stack_book, 11)
        end

        it "expects the number of 'rented' books is 7" do
          expect(Book.rented.count).to eq(7)
        end

        it "expects the number of 'stack' book is 11" do
          expect(Book.stack.count).to eq(11)
        end
      end
    end
  end
```

Rails アプリケーション開発を進めるにつれ、要件が変わっていくことはよくあります。本の貸出状態に予約済（reserved）を増やしてみましょう。book.rb は次のようになります。

```
  class Book < ApplicationRecord
    enum status: [:reserved, :rented, :stack]
  end
```

② テストの実行

ここで既存の機能に影響がないか、RSpec テストを実行して確認してみましょう。

```
$ bin/rake

Book
  testing for book status
    assign primitive value
      expects 0 to be mapped to 'rented' (FAILED - 1)
      expects 1 to be mapped to 'stack' (FAILED - 2)
    assign enum value
      expects 'rented' to be mapped to 0 (FAILED - 3)
      expects 'stack' to be mapped to 1 (FAILED - 4)
    enum attributes are available for where clause
      expects the number of 'rented' books is 7 (FAILED - 5)
      expects the number of 'stack' book is 11 (FAILED - 6)

Failures:

  1) Book testing for book status assign primitive
  value expects 0 to be mapped to 'rented'
     Failure/Error: expect(book.status).to eq('rented')

       expected: "rented"
            got: "reserved"

       (compared using ==)
     # ./spec/models/book_spec.rb:8:in block (4 levels) in <top (required)>'

  2) Book testing for book status assign primitive
  value expects 1 to be mapped to 'stack'
     Failure/Error: expect(book.status).to eq('stack')

       expected: "stack"
            got: "rented"

       (compared using ==)
     # ./spec/models/book_spec.rb:12:in block (4 levels) in <top (required)>'

  3) Book testing for book status assign enum value
  expects 'rented' to be mapped to 0
     Failure/Error: expect(Book.statuses[book.status]).to eq(0)

       expected: 0
            got: 1

       (compared using ==)
     # ./spec/models/book_spec.rb:19:in block (4 levels) in <top (required)>'

  4) Book testing for book status assign enum
  value expects 'stack' to be mapped to 1
     Failure/Error: expect(Book.statuses[book.status]).to eq(1)

       expected: 1
            got: 2
```

```
      (compared using ==)
    # ./spec/models/book_spec.rb:23:in block (4 levels) in <top (required)>'

  5) Book testing for book status enum attributes are available for
  where clause expects the number of 'rented' books is 7
     Failure/Error: expect(Book.rented.count).to eq(7)

       expected: 7
            got: 11

       (compared using ==)
     # ./spec/models/book_spec.rb:34:in block (4 levels) in <top (required)>'

  6) Book testing for book status enum attributes are available for
  where clause expects the number of 'stack' book is 11
     Failure/Error: expect(Book.stack.count).to eq(11)

       expected: 11
            got: 0

    (compared using ==)
   # ./spec/models/book_spec.rb:38:in block (4 levels) in <top (required)>'

Finished in 0.106 seconds (files took 1.63 seconds to load)
6 examples, 6 failures

Failed examples:

rspec ./spec/models/book_spec.rb:6 # Book testing for book status assign
primitive value expects 0 to be mapped to 'rented'
rspec ./spec/models/book_spec.rb:10 # Book testing for book status assign
primitive value expects 1 to be mapped to 'stack'
rspec ./spec/models/book_spec.rb:17 # Book testing for book status assign
enum value expects 'rented' to be mapped to 0
rspec ./spec/models/book_spec.rb:21 # Book testing for book status assign
enum value expects 'stack' to be mapped to 1
rspec ./spec/models/book_spec.rb:33 # Book testing for book status enum
attributes are available for where
clause expects the number of 'rented' books is 7
rspec ./spec/models/book_spec.rb:37 # Book testing for book status enum
attributes are available for where
clause expects the number of 'stack' book is 11
```

いままで苦労して作ったテストがすべてダメになりました。これはなぜでしょうか？ `ActiveRecord::Enum` にバグがあるわけではありません。`enum` の追加の方法に問題があり

ます。状態名を配列にした場合、対応する値は添字になるためです。なので、先頭に予約済（reserved）を追加した場合に結果が変わってしまいました。
追加する状態によらず、今まで提供してきた機能を崩さないためにはハッシュで状態を定義することが多いです。book.rb を次のように変更してテストを再度実行してみましょう。

```
class Book < ApplicationRecord
  enum status: {rented: 0, stack: 1, reserved: 2}
end
```

10.1.3 問題

例題を参考に、商品の注文ステータスを管理するモデルを作成し、enum で状態遷移を実装してみましょう。

10.2 Ruby on Rails ： EC サイトの開発 セッションと複数商品の注文

商品がそろってきたので買い物カートを追加し、ユーザーが複数の商品を一括で購入できるようにし、管理画面も複数商品購入の管理ができるように、変更しましょう。 今回は、買い物カートを実現するためにセッションを利用します。まずは、セッションの機能と注意点から見ていきましょう。

10.2.1 セッション機能について

(a) 解説

セッションとは何か

アプリケーションでは、特定のユーザーがどのような状態にあるかを観測したり識別したりすることがあります。例えば、ショッピングサイトの買い物カゴや、現在ログインしているユーザーの情報などです。 しかし、HTTP は基本的にステートレスプロトコルなので、カート情報やユーザ情報を次の HTTP リクエストまで保持しておくことができません。HTTP のリクエストを発行するたびにカートが空になってしまったり、ログインを行わなければならなくなります。 このため、これらの有効な情報を保持してステートフルを実現するため、Web アプリケーションのセッションを利用します。デフォルトではセッション情報はブラウザのクッキーに保存されます。アプリケーション層のセッションデータとブラウザのセッションデータの照合をすることで、ステートフルを実現しています。

Rails でのセッションの機能について

Rails は、ユーザーがアプリケーションに新しくアクセスするときに自動的にセッションを作成します。ユーザーが既にアプリケーションを使用中であれば、既存のセッションを読み込みます。そして、コントローラーで session インスタンスを使うことで、セッションに値を設定/取得といったアクセスができます。 なお、Rails4.0 以降では内容が暗号化されていますので、ブラウザ上でクッキーの値からセッションの内容を簡単に確認することはできません。暗号化に使われた秘密鍵の値を知っていれば、復号化して、セッションの内容を確認することが可能となります。

セッションを使うときの注意点

秘密情報を保持しない：デフォルトのセッションデータは、クッキーに保存されます。そのため、暗号化で改ざん防止機構はついていますが、内容を確認することができてしまうため、秘密情報は保持しないようにしましょう。
セッションには巨大なオブジェクトを格納しない：クッキーは 4KB 弱という容量制限があります。Rails のセッションの使われ方として、セッションに入れるデータはユーザーの ID と flash 程度にとどめることが推奨されています。
マーシャルできないオブジェクトは入れられない：セッションにデータを格納すると、保存対象のオブジェクトは Marshal#dump によってシリアライズされて保存されます。これは、IO オブジェクトや Proc オブジェクトなど、シリアライズできないオブジェクトはセッションに格納することができないという意味です。

10.2.2 セッションを利用したカート機能の実装

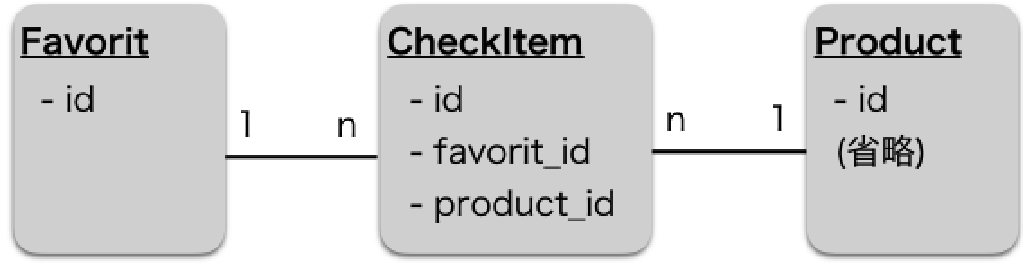
(a) 解説

セッションの利用例として「お気に入り登録」機能を実装し、商品名の横にお気に入りマークを表示するようにしましょう。 ユーザーが「お気に入り」ボタンをクリックしたら商品情報を取得し、セッション ID とともに DB へ保存します。このとき、トップページには、DB に保存された詳細ページの一覧を表示するだけです。

(b) 例題

① セッション管理テーブルの準備

まず、セッション用のテーブルを追加していきましょう。 セッション ID（クッキーに保存する値）を管理するテーブルとして Favorite テーブルと、チェックした商品明細を管理する CheckItem テーブルを準備します。その後、モデルのリレーション設定、セッション登録・管理を行うメソッドを追加、チェックした商品が閲覧できる画面を作成していきます。
Favorite モデルと CheckItem モデルの関係性は下の図のようになります。



- Favorite テーブルの追加

セッション ID を管理するためだけのテーブルですので、id (integer) のみを用意します。

```
$ rails generate scaffold Favorite
```

- CheckItem テーブルの追加

CheckItem テーブルは中間テーブルの位置づけですので、Favorite テーブルと Product テーブルの id を references にしておきましょう。

```
$ rails generate scaffold CheckItem product:references favorite:references
Favorite テーブル・CheckItem テーブルの作成をデータベースに反映しましょう。
$ rails db:migrate
```

- モデルの関連付け

Favorite、CheckItem、Product モデルの関連付けを行い、Favorite 削除時は同時に CheckItem の削除を行えるようにしておきましょう。

```
app/models/favorite.rb
class Favorite < ApplicationRecord
  has_many :check_items, dependent: :destroy
end
```

`dependent: :destroy` オプションを追加することで、favorite レコードを destroy メソッドで削除したとき、その favorite に紐づいていた check_item を Rails が全て削除してくれます。

`app/models/product.rb`

```
class Product < ApplicationRecord
  has_many :check_items
end
app/models/check_item.rb
class CheckItem < ApplicationRecord
  belongs_to :product
  belongs_to :favorite
end
```

②セッション登録機能の追加

現在の Favorite（お気に入り）を取得する処理を current_favorite メソッドに実装します。処理の流れは以下のとおりです。共通コントローラである ApplicationController に作成しましょう。

1. セッションから取得した favorite_id を元に Favorite テーブルからお気に入り情報を取得（存在しない場合、Favorite 作成）
2. 取得した Favorite 情報より ID を取得し、セッションに設定
3. Favorite 情報を返却

app/controller/application_controller.rb

```
class ApplicationController < ActionController::Base

  (省略)

private

  def current_favorite
    favorite = Favorite.find_or_create_by(id: session[:favorite_id])
    session[:favorite_id] = favorite.id
    favorite
  end
end
```

- Favorite に入れる機能

Favorite モデルに CheckItem を登録する add_product を実装しましょう。
商品が既にあれば取得、なければ新規で CheckItem オブジェクトを生成し、CheckItem オブジェクトを返してください。

```
app/models/favorite.rb
class Favorite < ApplicationRecord
  has_many :check_items, dependent: :destroy

  def add_product(product_id)
    check_items.find_or_initialize_by(product_id: product_id)
  end
end
```

セッションから Favorite 情報を取得し、check_items コントローラにお気に入りの商品を登録できるように create メソッドを実装しましょう。

Favorite と Product を取得し、CheckItem を追加してトップ画面へ遷移させてください。

app/controllers/check_items_controller.rb

```
class CheckItemsController < ApplicationController
  (省略)

  # POST /check_items
  # POST /check_items.json
  def create
    favorite = current_favorite
    product  = Product.find(params[:product_id])
    @check_item = favorite.add_product(product.id)

    respond_to do |format|
      if @check_item.save
        format.html { redirect_to root }
        format.json { render :show, status: :created, location: @check_item }
      else
        format.html { redirect_to products_index_url,
                      notice: 'Unprocessable entity.' }
        format.json { render json: @check_item.errors,
                      status: :unprocessable_entity }
      end
    end
  end

  (省略)
end
```

最後にトップ画面へ「お気に入り」ボタンとお気に入り表示を設置しましょう。お気に入りに入っているかどうかの判定は、ヘルパーメソッドとして実装します。

app/helpers/products_helper.rb 変更後

```
module ProductsHelper
  def current_favorite?(product, favorite)
    favorite.check_items.map{|i| i.product_id}.include?(product.id)
  end
end
```

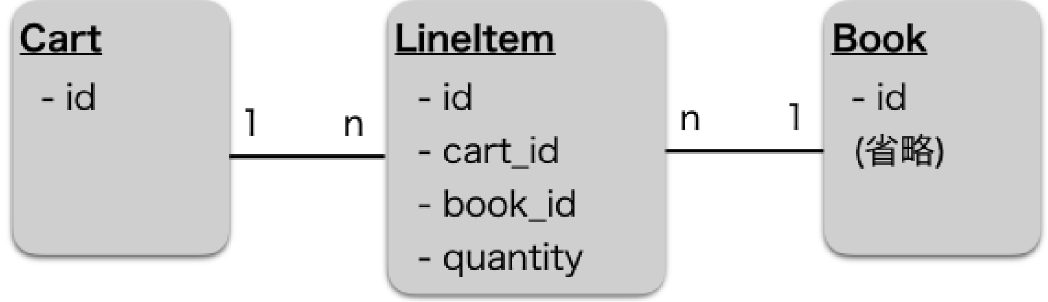
app/views/products/index.html.erb 変更後

```
(省略)
<div class="row marketing">
  <h2>BOOK</h2>
  <div class="col-lg-12">
    <% @books.each do |book| %>
      <h3><%= book.title %></h3>
      <p><%= book.author %></p>
      <p><%= book.published_on %></p>
      <p>
        <%= number_to_currency(book.try(:price), precision: 0, unit: "円") %>
        <% if current_favorite?(book, @favorite) %>
            <%= '☆' %>
        <% else %>
          <%= link_to 'お気に入り', check_items_path(product_id: book.id),
              method: :post, class: 'btn btn-default' %>
        <% end %>
        <%= link_to '購入', new_order_path(product_id: book.id),
            class: 'btn btn-default' %>
      </p>
    <% end %>
  </div>
</div>
```

(c) 問題

では、実際にカート(買い物かご)の機能を実装していきます。ユーザーがカートを利用して複数商品を一括購入できるようにしてください。セッション ID を管理するテーブルとして Cart テーブルとカート内の商品明細を管理する LineItem テーブルを用意しましょう。

Cart モデルと LineItem モデルの関係性は下の図のようになります。LineItem テーブルの quantity は、購入個数（integer）を表します。



実装方法がイメージできない場合は、以下のステップを参考に実装してみてください。

ステップ 1) セッション管理テーブルの準備

- Cart テーブルの追加
- LineItem テーブルの追加（個数については、初期値を設定）
- 追加したモデルの関連付けを設定

ステップ 2) セッション登録機能の追加

- ApplicationController に current_cart を実装

ステップ 3) カート登録機能の追加

- Cart モデルに商品を登録する add_product を実装
- LineItems コントローラの create メソッドで、セッションからカート情報を取得し、Cart#add_product を利用してカートに注文明細を登録
- トップ画面の「購入する」ボタンを「買い物かごに入れる」ボタンにしてリクエストを line_items#create に変更

ステップ 4) カート（買い物かご）一覧画面の編集

- LineItem モデルに book ごとの小計金額を表示するための total_price を実装
- Cart モデルにカート合計金額を表示するための total_price、合計個数を表示するための total_number を実装
- LineItem#total_price, Cart#total_price,total_number を利用してカート一覧画面（carts#show）を編集

ステップ 5) カート削除機能の追加

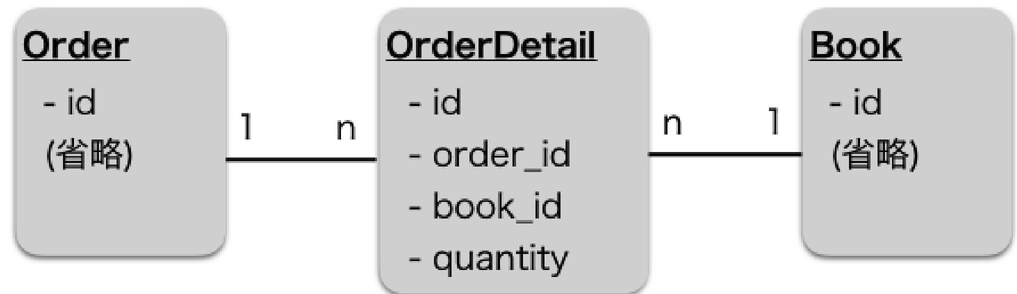
- Carts コントローラにセッションを削除できるように destroy メソッドを実装
- カート一覧画面に「買い物かごを空にする」ボタンを追加

10.2.3 複数明細の購入・管理への変更

ここまではセッションを利用してカート機能を実装しました。この章では、複数商品へ対応した購入処理と管理画面の修正を行います。

ユーザーがカートに入った複数商品を一括購入できる機能を実装しましょう。カートはセッションを参照していますので、それらを注文(Order)テーブルに置き換える必要があります

Order モデルはすでにあるものを利用します。新しく追加する OrderDetail モデルとの関係性は下の図のとおりです。



実装方法がイメージできない場合は、以下のステップを参考に実装してみてください。

(c) 問題

ステップ 1) 複数明細の購入処理テーブルの準備

一括で購入登録ができるように、OrderDetail モデルに個数管理を行うカラム（integer、初期値 0）を追加

ステップ 2) 複数明細の一括購入処理への変更

Orders コントローラの new,confirm,create メソッドで単一購入から複数購入できるように変更
カート一覧画面に「購入する」ボタンを追加してリクエストを orders#new にする
Order モデルに合計金額を表示するための total_price、合計個数を表示するための total_number を実装
OrderDetail モデルに商品の小計金額を表示するための total_price を実装
購入登録画面（orders#new）で複数の商品の情報が扱えるように修正
購入登録結果メールで複数の商品の情報が扱えるように修正

10.2.4 複数明細の管理

最後に、管理画面について変更します。

(c) 問題

注文管理画面で複数の商品の情報が扱えるように変更しましょう。

10.3 [評価課題]セッション管理

ここで、セッション管理で学んできたことのポイントを確認してみましょう。

HTTP は基本的にステート(Q1:　　　　　　　)プロトコルなので、カート情報やユーザ情報を次の HTTP リクエストを保持しておくことが(Q2:　　　　　　　)。ステート(Q3:　　　　　　　)を実現するため、Web アプリケーションのセッションを利用します。デフォルトではセッション情報はブラウザの(Q4:　　　　　　　)に保存されます。

セッションを使う時には、(Q5:　　　　　　　)を保持しない、巨大な(Q6:　　　　　　　)を格納しない、(Q7:　　　　　　　)できないオブジェクトは入れられないことに注意しましょう。

以下のコードを完成させましょう。

app/controllers/products_controller.rb

```
… def new @product = Product.new # セッションがあれば、name を復元 if
    (Q8:        )[:params].present? @product.name = (Q9:        ):params
    (Q11:        )[:params] = nil end end
def conf (Q12:        )[:params] = product_params end
private def product_params params.fetch(:product, {}) end …

app/views/products/new.html.rb
<%= form_with(model: @product, url: conf_products_path, local: true) do |f| %>
    <%= f.text_field :name, value: @product.name %> <%= f.submit '確認' %> <%
    end %>
```

app/views/products/conf.html.rb

```
<%= link_to '戻る', new_product_path %>
```

以下の①～④のコードの中で実行した際にエラーが発生するものは(Q13:　　　　)

①session[:hoge] = String.new

②session[:hoge] = Proc.new

③session[:hoge] = Array.new

④session[:hoge] = Hash.new

10.4 Ruby on Rails： gem を使わない検索

10.4.1 検索フォームの作成

ここでは商品一覧や注文一覧からタイトルや著者、購入者、状態から検索が行える機能を追加します。 Web アプリケーション開発では、検索機能を実装することが多くあります。以降の章で検索機能を簡単に実装できる ransack という gem を使って実装していきますが、まずは ransack を使わない検索を作ってみます。

検索条件は画面から入力します。下記のような顧客の検索画面を実装します。

プロジェクトの作成

```
$ rails _5.1.3_ new search_sample
```

Customers

Search

Name

Age

Search

Result

Employee	Name	Age			
岸部一樹(31)	猿谷夏希	35	Show	Edit	Destroy
岸部一樹(31)	小国香	45	Show	Edit	Destroy
岸部一樹(31)	菅尾大地	22	Show	Edit	Destroy
崎谷雄大(25)	針谷優太	16	Show	Edit	Destroy
崎谷雄大(25)	井上涼	75	Show	Edit	Destroy
崎谷雄大(25)	清宮義雄	32	Show	Edit	Destroy
北出小百合(55)	福江桐子	49	Show	Edit	Destroy
北出小百合(55)	川嶋和也	33	Show	Edit	Destroy
北出小百合(55)	片原昭夫	29	Show	Edit	Destroy

New Customer

scaffold を使った一覧画面の作成

モデルは下記の 2 つを準備します。 以降の章で検索の gemransack を使用して検索画面を実装しますが、モデルとデータは同じものを使用します。

社員 (Employee)

項目名	カラム名	型
名前	name	string
年齢	age	integer

顧客 (Customer)

項目名	カラム名	型
担当者	employee_id	integer
名前	name	string
年齢	age	integer

Scaffold を使って一覧画面を準備しましょう

```
$ rails g scaffold employee name age:integer
$ rails g scaffold customer employee:references name age:integer
```

generator で生成されたコードはその時点で一度コミットする癖を付けましょう。
migrate します。

```
$ rails db:migrate
```

リレーションの設定を行います。
app/models/employee.rb

```
class Employee < ApplicationRecord
  has_many :customers # 追加
end
```

次にサンプルデータを登録します。
db/seeds.rb に下記コードを追記しましょう

db/seeds.rb

```
employee1 = Employee.create(name: '岸部一樹', age: 31)
employee2 = Employee.create(name: '崎谷雄大', age: 25)
employee3 = Employee.create(name: '北出小百合', age: 55)
```

```
employee1.customers.create(name: '猿谷夏希', age: 35)
employee1.customers.create(name: '小国香', age: 45)
employee1.customers.create(name: '菅尾大地', age: 22)
employee2.customers.create(name: '針谷優太', age: 16)
employee2.customers.create(name: '井上涼', age: 75)
employee2.customers.create(name: '清宮義雄', age: 32)
employee3.customers.create(name: '福江桐子', age: 49)
employee3.customers.create(name: '川嶋和也', age: 33)
employee3.customers.create(name: '片原昭夫', age: 29)
```

サンプルデータを登録します。
`rails db:seed`
app/models/customer.rb は scaffold で references を指定したので自動生成されています。

検索フォームの作成

ここでは、顧客の検索をつくるので app/views/customers/index.html.erb に検索フォームを追加します。(1)

また、検索結果をわかりやすくるために、担当者（Employee）の名前と年齢が表示されるようにしましょう。(2)

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

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

<h1>Customers</h1>

<% # (1)検索フォーム ここから %>
<h2>Search</h2>
<%= form_with url: '/customers', local: true, method: :get do |f| %>
  <table>
    <tr>
      <th>Name</th>
      <td><%= f.text_field :name, value: params[:name] %></td>
    </tr>
    <tr>
      <th>Age</th>
      <td><%= f.number_field :age, value: params[:age] %></td>
    </tr>
  </table>
  <%= f.submit 'Search' %>
```

```
<% end %>
<h2>Result</h2>
<% # (1)検索フォーム ここまで追加 %>
<table>
  <thead>
    <tr>
      <th>Employee</th>
      <th>Name</th>
      <th>Age</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @customers.each do |customer| %>
      <tr>
        <% # 修正(2) 担当者の名前に(年齢)を追加 %>
        <td><%= customer.employee.name %>(<%= customer.employee.age %>)</td>
        <td><%= customer.name %></td>
        <td><%= customer.age %></td>
        <td><%= link_to 'Show', customer %></td>
        <td><%= link_to 'Edit', edit_customer_path(customer) %></td>
        <td><%= link_to 'Destroy', customer, method: :delete, data: { confirm:
    'Are you sure?' } %></td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>

<%= link_to 'New Customer', new_customer_path %>
```

ここでは、検索フォームは form_with を使用して作成します。
入力欄は text_field と number_field を利用します。 これらのメソッドに渡す第一引数には、画面の項目名、第二引数には初期値(最初に表示したい値)を設定します。

10.4.2 検索処理の作成

画面で入力した内容を受け取って検索を実行します。

app/controllers/customers_controller.rb

```
# 変更前
def index
  @customers = Customer.all
end

# 変更後
def index
  return @customers = Customer.all if params[:name].blank? &&
  params[:age].blank?

  # 条件が 2 つとも指定されている場合
  if params[:name].present? && params[:age].present?
    @customers = Customer.where(["name like ? and age = ?",
                                 "%#{params[:name]}%",
                                 "#{params[:age]}"]);
  else
    # name の条件だけ指定されている場合
    if params[:name].present?
      @customers = Customer.where(["name like ?", "%#{params[:name]}%"]);
    # age の条件だけ指定されている場合
    else
      @customers = Customer.where(["age = ?", "#{params[:age]}"]);
    end
  end
end
```

検索画面の検索条件には、Customer に対する name と、Customer に対する age の検索条件を配置しました。
検索画面で入力した検索条件は、画面の submit ボタンをクリックすると、CustomersController に検索条件が渡されます。
CustomersController は、渡された検索条件 params に含まれる条件:name と:age を取り出して検索条件とします。
present?メソッドを使用して、検索条件が入力されているか判断します。
Customer の name に対しては、like 構文で%を使用して部分一致検索をするように記述しているので、name の検索条件に'岸辺'と入力すると like '%岸辺%'と検索のための構文を作成します。

2 つの項目を使って、検索する組み合わせは

name	age	検索対象
条件なし	条件なし	検索条件なし(全て対象)
条件あり	条件あり	name に部分一致、かつ age に一致
条件あり	条件なし	name に部分一致
条件なし	条件あり	age に一致

になります。

rails s でサーバを起動して http://localhost:3000/customers にアクセスしてみましょう。
検索フォームと検索結果が表示されていると思います。
条件を入力して検索してみましょう。
また、条件が多い場合は、以下のように記述することもできます。

```
# 変更後
def index
  @customers = Customer.all

  if params[:name].present?
    @customers = @customers.where("name like ?", "%#{params[:name]}%")
  end

  if params[:age].present?
    @customers = @customers.where(age: params[:age])
  end
end
```

検索項目の組み合わせを考慮せず、検索項目が入力された項目だけを SQL の where 句として組み立てます。

複数の条件がある場合でも、@customers に where を継ぎ足していくことで、検索の条件を追加できます。

ポイント

上記の例では、直接 SQL 文を組み立てていませんが、直接組み立てる場合は検索条件に入力
される条件によっては全く予期しない SQL 文になる場合があります。
例えば、name の欄に' or 1=1'と入力して、search ボタンをクリックすると、Rails のログには

```
Started GET
   "/customers?utf8=%E2%9C%93&name=%27+or+1%3D1%27&age=&commit=Search"
   for 127.0.0.1 at 2019-11-16 12:03:12 +0900
Processing by CustomersController#index as HTML
  (略)
  Customer Load (0.2ms)  SELECT "customers".* FROM "customers" WHERE  (name
like '%'' or 1=1''%')
  (略)
Completed 200 OK in 54ms (Views: 51.5ms | ActiveRecord: 0.2ms)
```

のように実行結果が表示されます。name に対する検索条件が(と)でくくられ、入力した条
件がそのまま文字列として検索条件になるように' or 1=1'が'' or 1=1''と置換され、SQL
で文字列や時刻の値をくくるための記号'が''に置換され、無害な文字列'と解釈されるよう
になります。

考え方の一例ですが、SQL 文を組み立てる場合に、name like '%' + 条件 + '%'のように記
述して、検索すると

```
SELECT customers.* FROM customers WHERE name like '%' or 1=1;
```

と、name に対する条件以外に、or 1=1 など、常に条件が成立する SQL 文が含まれ、本来は表示できてはいけないデータが表示されることも考えられますので、Rails では、

```
@customers.where("name like ?", "%#{params[:name]}%")
```

のように記述するようにしましょう。
このように、入力された条件で期待している結果が取得できるテストをすることはもちろんですが、入力された条件によって期待している結果以外の動作をしないこともテストをして問題がないことを確認するようにしましょう。

10.5 Ruby on Rails：EC サイトの開発 検索

ここでは商品一覧や注文一覧からタイトルや著者、購入者、状態から検索が行える機能を追加します。 Web アプリケーション開発では、このような検索機能を実装することが多くあります。そこで検索機能を簡単に実装するために ransack という gem を使って実装していきます。

10.5.1 ransack のしくみ

ransack は、検索条件を入力するフォームに機能を追加します。

たとえば、ransack を使用せずに検索機能を作成する場合、フォームには検索に利用する条件を設定し、実際に検索のアクションが実行されたときに、コントローラが検索条件の情報を利用して、条件と一致する、条件より大きい、条件と不一致、条件を含むなどをロジックで記述します。ransack を使用することで、フォームには検索項目と検索条件を同時に記述することができ、コントローラでは、その検索条件を処理する文を追加するだけで、検索条件を組み立てるプログラムを記述する必要がなくなり、プログラムの可読性や保守性を高めることができます。

10.5.2 ransack の使い方

ransack をインストールすると各種モデルに ransack というメソッドが追加されます。検索条件を引数にして ransack メソッドを呼び出すと Ransack::Search オブジェクトが返ってきます。 そのオブジェクトに対して result メソッドを呼び出すことで検索結果を取得できます。

```
def index
  @q = User.ransack params[:q]
  @users = @q.result
end
```

検索条件は画面から入力します。 その画面のフォームを作成するために ransack では search_form_for という ViewHelper が準備されています。

```
<%= search_form_for @q do |f| %>
  <%= f.text_field :name_cont %>
  <%= f.submit %>
<% end %>
```

(b) 例題

下記のような顧客の検索画面を実装します。

scaffold を使った一覧画面の作成

モデルは下記の 2 つを準備します。

社員 (Employee)

項目名	カラム名	型
名前	name	string
年齢	age	integer

顧客 (Customer)

項目名	カラム名	型
担当者	employee_id	integer
名前	name	string
年齢	age	integer

Gemfile への追加とインストールをしましょう

Gemfile

```
gem 'ransack'
```

インストール

```
$ bundle install
```

Scaffold を使って一覧画面を準備しましょう

```
$ rails g scaffold employee name age:integer
$ rails g scaffold customer employee:references name age:integer
```

generator で生成されたコードはその時点で一度コミットする癖を付けましょう。
migrate します。

```
$ rails db:migrate
```

リレーションの設定を行います。

app/models/employee.rb

```
class Employee < ApplicationRecord
  has_many :customers # 追加
end
```

次にサンプルデータを登録します。

db/seeds.rb に下記コードを追記しましょう

db/seeds.rb

```
employee1 = Employee.create(name: '岸部一樹', age: 31)
employee2 = Employee.create(name: '崎谷雄大', age: 25)
employee3 = Employee.create(name: '北出小百合', age: 55)

employee1.customers.create(name: '猿谷夏希', age: 35)
employee1.customers.create(name: '小国香', age: 45)
employee1.customers.create(name: '菅尾大地', age: 22)
employee2.customers.create(name: '針谷優太', age: 16)
employee2.customers.create(name: '井上涼', age: 75)
employee2.customers.create(name: '清宮義雄', age: 32)
employee3.customers.create(name: '福江桐子', age: 49)
employee3.customers.create(name: '川嶋和也', age: 33)
employee3.customers.create(name: '片原昭夫', age: 29)
```

サンプルデータを登録します。

```
rails db:seed
```

app/models/customer.rb は scaffold で references を指定したので自動生成されています。

検索フォームの作成

ここでは、顧客の検索をつくるので app/views/customers/index.html.erb に検索フォームを追加します。(1)

また、検索結果をわかりやすくるために、担当者(Employee)の名前と年齢が表示されるようにしましょう。(2)

app/views/customers/index.html.erb

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

<h1>Customers</h1>

<% # 修正(1) 検索フォーム ここから %>
<h2>Search</h2>
<%= search_form_for @q do |f| %>
  <table>
    <tr>
```

```
      <th>Name</th>
      <td><%= f.text_field :name_cont %></td>
    </tr>
    <tr>
      <th>Age</th>
      <td><%= f.number_field :age_gteq %> 〜 <%=
    f.number_field :age_lteq %></td>
    </tr>
    <tr>
      <th>Employee</th>
      <td><%= f.collection_select :employee_id_eq,
    Employee.all, :id, :name, include_blank: true %></td>
    </tr>
    <tr>
      <th>Emplyee Age</th>
      <td><%= f.number_field :employee_age_gteq %> 〜 <%=
    f.number_field :employee_age_lteq %></td>
    </tr>
  </table>
  <%= f.submit %>
<% end %>
<h2>Result</h2>
<% # 修正(1) 検索フォーム ここまで追加 %>
<table>
  <thead>
    <tr>
      <th>Employee</th>
      <th>Name</th>
      <th>Age</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @customers.each do |customer| %>
      <tr>
        <% # 修正(2) 担当者の名前に(年齢)を追加 %>
        <td><%= customer.employee.name %>(<%=
    customer.employee.age %>)</td>
        <td><%= customer.name %></td>
        <td><%= customer.age %></td>
        <td><%= link_to 'Show', customer %></td>
        <td><%= link_to 'Edit', edit_customer_path(customer) %></td>
        <td><%= link_to 'Destroy', customer, method: :delete, data:
    { confirm: 'Are you sure?' } %></td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>

<%= link_to 'New Customer', new_customer_path %>
```

検索フォームを作成するにはransackが提供するViewHelperのsearch_form_forを利用します。 引数にはRansack::Searchオブジェクトを渡します。
それぞれの入力欄はform_forと同様に#text_fieldや#number_fieldを利用します。
これらのメソッドに渡す第一引数によって、どのカラムからどのような条件で検索するかが決まります。

例えばf.text_field :name_contの場合nameというカラムに入力した内容が含まれているという条件になります。 他にも下記のような条件が設定できます。

引数	条件
カラム名_cont	入力した内容が含まれる
カラム名_start	入力した内容から始まる
カラム名_eq	入力した内容と同じ
カラム名_gt	入力した内容より大きい
カラム名_lt	入力した内容より小さい
カラム名_gteq	入力した内容以上
カラム名_lteq	入力した内容以下

Customerモデルはbelongs_to :employeeが設定されているのでカラム名をemployee_ageとすることで 担当者の年齢から検索することも可能です。

検索処理の作成

画面で入力した内容を受け取って検索を実行します。

```
app/controllers/customers_controller.rb
  # 変更前
  def index
    @customers = Customer.all
  end

  # 変更後
  def index
    @q = Customer.ransack params[:q]
    @customers = @q.result
  End
```

最初に説明した通りCustomerモデルにransackメソッドが追加されているので、画面で入力した検索条件を引数にして呼び出します。

ransack メソッドは Ransack::Search オブジェクトを返します。このオブジェクトは view で利用するためインスタンス変数にセットしておきます。

また、このオブジェクトに対して result メソッドを呼び出すことで検索を実行して結果が取得できます。

rails s でサーバを起動して http://localhost:3000/customers にアクセスしてみましょう。 検索フォームと検索結果が表示されていると思います。

条件を入力して検索してみましょう。

ソート機能の追加

ransack が提供する ViewHelper の sort_link を利用することで簡単にソート機能を実装することが出来ます。

app/views/customers/index.html.erb

```
# 変更前
<table>
  <thead>
    <tr>
      <th>Employee</th>
      <th>Name</th>
      <th>Age</th>
      <th colspan="3"></th>
    </tr>
  </thead>


# 変更後
<table>
  <thead>
    <tr>
      <th><%= sort_link(@q, :employee_name, 'Employee') %></th>
      <th><%= sort_link(@q, :name, 'Name') %></th>
      <th><%= sort_link(@q, :age, 'Age') %></th>
      <th colspan="3"></th>
    </tr>
  </thead>
```

変更したら、検索結果のテーブルのヘッダ部分をクリックしてみましょう。 クリックした項目でソートされるようになっています。

(c) 問題

EC サイトの商品一覧に下記の項目で絞り込み検索ができるように機能を追加しましょう

- タイトル
- 著者
- 価格

10.6 [評価課題]EC サイトの検索機能

ここで、検索で学んできたことのポイントを確認してみましょう。

ransack を使用せずに検索が行えるように、以下のコードを完成させましょう。

```
def index
   @user = User.all

   # 検索結果を取得

   # params[:name]が存在する場合は name の部分一致検索を実行する
   if params[:name].present?
     @user = (Q1:       ).where("users.name like ?", (Q2:       ))
   end

   # params[:age]が存在する場合は age の一致検索を実行する
   if params[:age].present?
     @user = (Q3:       ).where((Q4:       ): params[:age])
   end

 end
```

```
<%# 検索フォームを作成 %>
<%= (Q5:       ) url: users_path, local: true, method: :get do |f| %>
  <table>
    <tr>
      <th>Name</th>
      <td><%= f.text_field :name, value: params[:name] %></td>
      <th>Age</th>
      <td><%= f.number_field :age, value: params[:age] %></td>
    </tr>
  </table>

  <%= f.submit '検索' %>
<% end %>
```

ransack を使用せずに検索が行えるように、以下のコードを完成させましょう。

```
 def index
   return @user = User.all if params[:name].blank? && params[:age].blank?

   # 検索結果を取得
   # params[:name]が存在する場合は name の部分一致検索を、
```

```
    # params[:age]が存在する場合はageの一致検索を実行する
    # どちらも存在する場合は両方とも実行する。
    if params[:name].present? && params[:age].present?
      @user = (Q6:       ).where("(Q7:       )", (Q8:       ))
    elsif params[:name].present?
      @user = (Q9:       ).where("(Q10:       )", (Q11:       ))
    else
      @user = (Q12:       ).where("(Q13:       )", (Q14:       ))
    end
  end
```

ransackを利用して検索が行えるように、以下のコードを完成させましょう。

```
  def index
    # Ransack::(Q15:       )オブジェクトを取得
    @q = User.(Q16:       )(params[:q])
    # 検索結果を取得
    @user = @q.(Q17:       )
  end
<%# 検索フォームを作成 %>
<%= (Q18:       ) @q do |f| %>
  <table>
    <tr>
      <th>Name</th>
      <td><%= f.text_field :name_cont %></td>
      <th>Age</th>
      <td><%= f.number_field :age_eq %></td>
      <th>Employee</th>
      <td><%= f.collection_select :employee_id_eq, Employee.all, :id, :name,
    include_blank: true %></td>
    </tr>
  </table>

  <%= f.submit '検索' %>
<% end %>
```

ransackを利用して以下の条件を満たした検索が行えるように、以下のコードを完成させてください。

①入力値を含むauthorを検索したい

```
<%= f.text_field :author_(Q19:       ) %>
```

②入力値より大きいpriceを検索したい

```
<%= f.number_field :price_(Q20:       ) %>
```

③入力値より小さい price を検索したい

```
<%= f.number_field :price_(Q21:       ) %>
```

④入力値から始まる tag を検索したい

```
<%= f.text_field :tag_(Q22:       ) %>
```

⑤入力値以下の fee を検索したい

```
<%= f.number_field :fee_(Q23:       ) %>
```

⑥入力値以上の fee を検索したい

```
<%= f.number_field :fee_(Q24:       ) %>
```

⑦入力値と等しい order_number を検索したい

```
<%= f.number_field :order_number_(Q25:       ) %>
```

ransack を利用してソートが行えるように、以下のコードを完成させましょう。

```
<%# 検索結果ヘッダ部分 %>
<table>
  <thead>
    <tr>
      <%# <th>Name</th>にソート機能を追加 %>
      <th><%= (Q26:       )(@q, :name, 'Name') %></th>
    </tr>
  </thead>
```

10.7 Ruby on Rails ： EC サイトの開発 Heroku へのデプロイ

10.7.1 Heroku について

アプリケーションを公開するには、サーバーにプログラムファイルを配置し、それにセキュリティなどを考慮して動く状態にする必要があります。これらの作業をデプロイといいます。
今までのデプロイは、サーバー用の PC を用意し、OS や使用する言語をインストールし、セキュリティーなどの設定を追加し、そのうえでプログラムファイルを設置して動作確認をするという、とても手間がかかるものでした。

しかし、近年のクラウドサービスの進歩は目覚ましいものがあり、ブラウザから管理画面にログインして表示されるボタンをいくつかクリックしたり、コマンドを数行実行したりするだけでデプロイができてしまうというものが主流になりつつあります。

その中で Heroku は、PaaS（Platform as a Service）と呼ばれるクラウドサービスで、ローカルのターミナルから直接 Git コマンドでデプロイを行ったり、Add-ons と呼ばれるいろいろな拡張機能をクリックするだけでそれらの機能が追加できるという特徴があります。Heroku が用意した Git リポジトリに `git push` することで、Dyno と呼ばれる Linux コンテナが作られアプリケーションをインストールします。

デフォルトのままであれば、そのとき同時に DB コンテナも用意されます。Dyno はアプリケーションの規模によって拡張され、同時に課金の対象となります。

Dyno とは

個々のアプリケーションが動作するための独立した環境です。アプリケーションを動作させるためには OS が必要です。Heroku は PaaS ですから、OS はあらかじめ用意されているものから選択します。
（参考：https://www.heroku.com/dynos）

スタックとは

dyno などで動作する OS のイメージです。2019 年 12 月現在では、以下のイメージがあります。

Stack Version	Base Technology
Heroku-18 (default)	Ubuntu 18.04
Heroku-16	Ubuntu 16.04
Container	Docker

（参考：https://devcenter.heroku.com/articles/stack）

気をつけなければいけないのは、この Dyno はデプロイごとに作り直されてしまうということです。例えば、Rails のプロジェクトディレクトリ内に画像データなどを蓄積するものがありますが、デプロイすると削除されてしまいますので、その場合は Amazon S3 など外部サービスと連携させる必要があります。

一方、DB コンテナは一番最初にアプリケーションを Heroku に置いたときに作られますが、自動的に Rails の migraion ファイルを適応しません。手動でコマンドを実行することになります。ただ、DB コンテナはアプリケーションを削除するまで消されることはありません。

10.7.2 Heroku へのリリース準備

Heroku のアカウントを持っていない人は、以下のサイトの Sign up からアカウントを作成してください。アカウントの作成は無料です。今回のアプリケーションのレベルでは課金されることはありませんのでご安心ください。また、ログイン ID とパスワードはデプロイのときに必要ですのですぐわかるようにしておいてください。

https://www.heroku.com/

では、Heroku へのリリースの準備をしましょう。

（参考：https://devcenter.heroku.com/articles/getting-started-with-rails5）

データベースの変更をします。Heroku では、デフォルトのデータベースは PostgreSQL です。

config/datagase.yml を開いて、production 環境のデータベース指定を変更してください。

```
production:
  adapter: postgresql
  database: heroku_production # わかりやすい名前で自由につけてかまいません
```

それから、PostgreSQLを使用するためのgemを追加しなければいけません。ただし、developmentモードとtestモードではこのままSQLiteを利用しますので、gem sqlite3はdevelopment/test専用とし、gem pgはproduction専用になるように変更してください。

また、Gemfileの先頭でRubyのversionも指定しましょう。今回は、2.4.2で記載していますが、ここのversionは開発しているversionを指定してください。versionを指定しない場合は、herokuのデフォルトのバージョンになります。何か問題が生じた場合の切り分けや、開発環境の確認のためにも明示的に記載したほうが良いでしょう。

Gemfile

```
ruby '2.4.2'
     .
     .
group :development, :test do # 追加
  gem 'sqlite3'
end # 追加

group :production do # 追加
  gem 'pg', "~> 0.21.0" # 追加
end # 追加
```

このあと、 bundle installを忘れずに実行してコミットまでしておいてください。

10.7.3 Herokuへの公開

(a) Herokuコマンドの紹介

Herokuへのデプロイはとてもシンプルです。コマンドを順番に実行するだけでデプロイができてしまいます。以下の通りにターミナルへ入力していってください。（参考：https://devcenter.heroku.com/articles/getting-started-with-rails5）

①Herokuへのログイン

ターミナルからHerokuへログインしておきます。これであとのHerokuコマンドが利用できるようになります。

```
username:~/workspace (master) $ heroku login -i
Enter your Heroku credentials.
Email: your@email.com
Password (typing will be hidden):
Logged in as your@email.com
```

② アプリケーション公開の初期設定

app-name の部分にアプリケーションの名前を入力してください。これは、Heroku のダッシュボードで識別するためのもので、Heroku が用意する url にも反映されます。rails のプロジェクト名と同じである必要はありません。自由に名前をつけることができます。もし他のプロジェクトと名前が重複している場合はメッセージが出ますので、その場合は違う名前で実行しなおしてください。

```
usernamey:~/workspace (master) $ heroku create app-name
Creating ⬢ app-name... done
https://app-name.herokuapp.com/ | https://git.heroku.com/app-name.git
```

この中の https://app-name.herokuapp.com がこのアプリケーションに割り当てられた URL です。 これが終わると、git のリモートリポジトリとしての Heroku が追加されます。

```
usernamey:~/workspace (master) $ git remote -v
heroku  https://git.heroku.com/app-name.git (fetch)
heroku  https://git.heroku.com/app-name.git (push)
```

③ Heroku へのデプロイ

デプロイは、git コマンドで Heroku のリモートリポジトリへ push するだけです。push 先のブランチ名は必ず master にしてください。

```
usernamey:~/workspace (master) $ git push heroku master
Counting objects: 478, done.
Delta compression using up to 8 threads.
Compressing objects: 100% (440/440), done.
Writing objects: 100% (478/478), 105.43 KiB | 0 bytes/s, done.
Total 478 (delta 207), reused 0 (delta 0)
remote: Compressing source files... done.
remote: Building source:
remote:
remote: -----> Ruby app detected
remote: -----> Compiling Ruby/Rails
remote: -----> Using Ruby version: ruby-2.2.6
remote: -----> Installing dependencies using bundler 1.13.6
remote:        Running: bundle install
                --without development:test
                --path vendor/bundle
                --binstubs vendor/bundle/bin -j4 --deployment
remote:        Fetching gem metadata from https://rubygems.org/.........
remote:        Fetching version metadata from https://rubygems.org/..
remote:        Fetching dependency metadata from https://rubygems.org/.
remote:        Installing i18n 0.7.0
remote:        Installing rake 12.0.0
(中略)
remote: -----> Discovering process types
remote:        Procfile declares types     -> (none)
remote:        Default types for buildpack -> console, rake, web, worker
remote:
remote: -----> Compressing...
```

```
remote:        Done: 30M
remote: -----> Launching...
remote:        Released v5
remote:        https://app-name.herokuapp.com/ deployed to Heroku
remote:
remote: Verifying deploy... done.
To https://git.heroku.com/app-name.git
 * [new branch]      master -> master
```

④ アプリケーションのデータベース設定

データベースの反映は自動的には行われません。 db:migrate を実行しましょう。2 回目以降、アプリケーションの修正などでデータベースの反映がなければ、これを実行する必要はありません。

```
usernamey:~/workspace (master) $ heroku run rails db:migrate
Running rake db:migrate on ⬢ app-name... up, run.4419 (Free)
== 20161210052143 CreateBooks: migrating =================================
-- create_table(:books)
   -> 0.0085s
== 20161210052143 CreateBooks: migrated (0.0086s) ========================

== 20161210052334 CreateTags: migrating ==================================
-- create_table(:tags)
   -> 0.0078s
== 20161210052334 CreateTags: migrated (0.0079s) =========================

== 20161210052429 CreateTaggings: migrating ==============================
-- create_table(:taggings)
   -> 0.0227s
== 20161210052429 CreateTaggings: migrated (0.0228s) =====================

== 20161219102255 DeviseCreateUsers: migrating ===========================
-- create_table(:users)
   -> 0.0114s
-- add_index(:users, :email, {:unique=>true})
   -> 0.0078s
-- add_index(:users, :reset_password_token, {:unique=>true})
   -> 0.0085s
== 20161219102255 DeviseCreateUsers: migrated (0.0280s) ==================

== 20161221085504 AddColumnsToUser: migrating ============================
-- add_column(:users, :name, :string)
   -> 0.0016s
-- add_column(:users, :role, :string)
   -> 0.0014s
== 20161221085504 AddColumnsToUser: migrated (0.0033s) ===================
```

これで Heorku でのデプロイが完成しました。これで Heroku に割り当てられた URL からアプリケーションにアクセスすることができるかどうか、確認してください。

期待した画面が表示されていれば確認できたらデプロイ完了です。おめでとうございます！

(b) Heroku のその他機能の紹介

ここでは、運用、デバッグしていく際によく使う Heroku コマンドをご紹介します。コマンドのオプションなどが変更される可能性もありますので、最新情報は Heroku の開発者向けホームページを参照ください。
https://devcenter.heroku.com

① 環境変数の一覧

デフォルトでの一覧は以下のとおりです。 heroku config:set などで、ここに任意の環境変数を追加することもできます。詳しくは heroku help config を見てください。

```
username:~/workspace (master) $ heroku config
=== app-name Config Vars
DATABASE_URL:              postgres://(何かの羅列).amazonaws.com:(何かの羅列)
LANG:                      en_US.UTF-8
RACK_ENV:                  production
RAILS_ENV:                 production
RAILS_LOG_TO_STDOUT:       enabled
RAILS_SERVE_STATIC_FILES:  enabled
SECRET_KEY_BASE:           (何かの羅列)
```

② ログ閲覧

デフォルトでは直近から 1500 行のログが保存されています。それ以上保存したい場合は、Add-ons で拡張機能を利用してください。 もしくは、ブラウザの管理画面でもログを見ることができます。--tail オプションをつけるとアプリケーションへのアクセスログがインクリメンタルに見ることができます。

```
https://dashboard.heroku.com/apps/app-name/logs
username:~/workspace (master) $ heroku logs --tail
2016-12-24T08:14:44.612193+00:00 app[api]:
  Initial release by user test@example.com
2016-12-24T08:14:44.841327+00:00 app[api]:
  Enable Logplex by user test@example.com
2016-12-24T08:14:44.612193+00:00 app[api]:
  Release v1 created by user test@example.com
2016-12-24T08:14:44.841327+00:00 app[api]:
  Release v2 created by user test@example.com
2016-12-24T08:16:17.196667+00:00 app[api]:
  Set LANG, RACK_ENV, RAILS_ENV, RAILS_LOG_TO_STDOUT,
  RAILS_SERVE_STATIC_FILES,
  SECRET_KEY_BASE config vars by user test@example.com
2016-12-24T08:16:17.196667+00:00 app[api]:
  Release v3 created by user test@example.com
2016-12-24T08:16:18.396611+00:00 app[api]:
  Release v4 created by user test@example.com
2016-12-24T08:16:18.396611+00:00 app[api]:
  Attach DATABASE (@ref:postgresql-rugged-32907) by user test@example.com
```

```
2016-12-24T08:16:19.247083+00:00 heroku[slug-compiler]:
  Slug compilation started
2016-12-24T08:16:19.247093+00:00 heroku[slug-compiler]:
  Slug compilation finished
2016-12-24T08:16:18.927856+00:00 app[api]:
  Deploy 69acfa3 by user test@example.com
2016-12-24T08:16:18.927856+00:00 app[api]:
  Release v5 created by user test@example.com
2016-12-24T08:16:22.171682+00:00 heroku[web.1]:
  Starting process with command bin/rails server -p 27764 -e production
....
```

③ アプリケーションのリスタート

Dyno（Linux コンテナ）をリスタートします。

```
username:~/workspace (master) $ heroku restart
Restarting dynos on ⬢ app-name... done
```

④ rails コンソールの起動

Dyno（Linux コンテナ）でアプリケーションのコンソールを起動します。

```
usernmae:~/workspace (master) $ heroku run rails console
Running rails console on ⬢ app-name... up, run.1384 (Free)
Loading production environment (Rails 5.0.0.1)
irb(main):001:0>
```

⑤ bash の起動

Dyno（Linux コンテナ）でターミナルを起動します

```
username:~/workspace (master) $ heroku run bash
Running bash on ⬢ app-name... up, run.6008 (Free)
~ $ ls
app  config  db  Gemfile.lock  log  Rakefile  README.rdoc  test
vendor  bin  config.ru  Gemfile  lib  public  README.md  spec tmp
```

⑥ データバックアップ

PostgreSQL を使用した場合のデータのバックアップ方法です。 まず、データセットを作成します。

```
username:~/workspace (master) $ heroku pg:backups capture
Use Ctrl-C at any time to stop monitoring progress; the backup
will continue running. Use heroku pg:backups info to check progress.
Stop a running backup with heroku pg:backups cancel.

DATABASE ---backup---> b001
```

Backup completed

そうすると、下記のように ID が付けられたバックアップファイルが作成されます。バックアップファイルは最大で 2 つまで保存され、古いものから順に削除されていきます。

```
username:~/workspace (master) $ heroku pg:backups
=== Backups
ID    Backup Time                Status
----  -------------------------  ----------------------------------
b001  2016-12-27 07:41:11 +0000  Completed 2016-12-27 07:41:15 +0000

Size    Database
------  --------
14.8kB  DATABASE

=== Restores
No restores found. Use heroku pg:backups restore to restore a backup

=== Copies
No copies found. Use heroku pg:copy to copy a database to another
```

ローカルで Heroku コマンドを使用している場合、ここで heroku pg:backups:download とすれば勝手にダンプファイルのダウンロードができます。しかし、Cloud9 では制限されているため、バックアップ保存先の url を取得し、そこに直接アクセスすることになります。

```
username:~/workspace (master) $ heroku pg:backups public-url b001
The following URL will expire at 2016-12-27 09:35:29 +0000:
  "https://xxxx.s3.amazonaws.com/xxxxx"
```

https で始まる Amazon S3 の長い URL が表示されます。そこにアクセスするとダンプファイルをダウンロードすることができます。

⑦ データリストア

データのリストアでも制限があり、バックアップ ID がついたものを直接リストアすることしかできません。ですので、Heroku で運用する場合は、すべて Heroku 上で解決する方法が必要です。リストアのコマンドは以下の通りです。

```
username:~/workspace (master) $ heroku pg:backups restore b003 DATABASE_URL

 !    WARNING: Destructive Action
 !    This command will affect the app: app-name
 !    To proceed, type "app-name" or re-run this command with
 !    --confirm app-name

> app-name
Use Ctrl-C at any time to stop monitoring progress; the backup
will continue restoring. Use heroku pg:backups to check progress.
Stop a running restore with heroku pg:backups cancel.

r005 ---restore---> DATABASE
```

```
Restore completed
```

restoreのあとに、バックアップIDと、環境変数のDATABASE_URLまたは該当のデータベースのURLを指定します。ここでのDATABASE_URLは、先のheroku configで確認できるURLです。

Cloud9やHerokuなどのいわゆるクラウドサービスは進化・変化も早く、ここにあげた内容がすぐに陳腐化する可能性もあります。常に最新の情報を元に開発を進めるようにしてください。

10.8 Ruby on Rails：EC サイトの開発 ～スプリント＃１ふりかえり～

ECサイト開発

～スプリント＃１のふりかえり～

「アジャイルの各種プラクティス」に記載
されている手法を使って、グループごとに
ふりかえりをしてみましょう

質問の例

1. ECサイト開発はうまくやれましたか？
2. 開発をすすめていく上で、
   なにか問題はありますか？
3. より学習効率を高めるために
   どんなことをすればよいと思いますか？

決定したカイゼンアクションについて
みんなの前で発表しましょう

ECサイト開発

～スプリント＃２のふりかえり～

「アジャイルの各種プラクティス」に記載されている手法（前回使用したものと別の手法）を使って、グループごとにふりかえりをしてみましょう

質問の例

1. 前回のふりかえりで決めたカイゼン
   アクションは実行できましたか？
2. その結果、目指していた状態に
   近づけましたか？
3. どんな問題が発生しましたか？
4. よりよい状態にするために
   どんなことをすればよいと思いますか？

決定したカイゼンアクションについて
みんなの前で発表しましょう

アジャイルソフトウェア開発

まとめ

アジャイル開発について

- アジャイル開発では、顧客から素早くフィードバックを受け取ることで、プロダクトの価値を高めることを目的としている。
- 代表的なアジャイル開発手法としては、リーン、XP、スクラムがあり、近年では導入企業も増えてきている。
- アジャイルソフトウェア開発宣言では、左側に書かれている項目（ドキュメントやツールなど）に価値がないと言っているわけではないことに注意。
- アジャイル開発手法を取り入れただけでプロジェクトが成功するわけではない。円滑なコミュニケーションや技術者のスキル向上は必須。
- アジャイルチームは自己組織化されたチームである。

- 他のチームでうまくいっているやり方を真似ても、
  同じようにうまくいくとは限らない。
  チームの状態・状況に合わせて、適切なプラクティスを導入する必要がある。
- 一定のタイムボックスごとにふりかえりを実施し、継続的に改善していく。
- 身近なところから始め、徐々に組織に広げていく。

スクラムについて

- 理解は容易だが、習得は非常に難しい。
- 定義されている型通りに実行できるようになるまで続けること。
  安易なカスタマイズは、かえって状況を悪化させてしまうことが多い。
- プロダクトバックログアイテムが完成したか、チームが自己判断できる
  ようにするため、「完成」の状態を定義・共有しておく必要がある。
- ロール（プロダクトオーナー、スクラムマスター、開発チーム）は
  兼任してはいけない。

プラクティスについて

- むやみにプラクティスを導入してもうまくいかない。
  過去に発生した（または現在発生している）問題に対処するための
  プラクティスを選定すること。
- 要求の変更に適応するための技術的なプラクティス（コーディングルール、
  テスト駆動開発、YAGNI、継続的インテグレーションなど）は積極的に
  活用するとよい。

10.9 Ruby on Rails：まとめ

ここでは、Ruby on Rails の各種コマンドと、Ruby on Rails のプロジェクトのディレクトリ構成についてまとめていきます。

10.9.1 Ruby on Rails のコマンド

コマンドまとめ

コマンド	説明
rails new	アプリ（プロジェクト）を新規作成します
rails generate scaffold	リソースを一括で作成します
rails db:create	DB を作成します
rails db:migrate	migrattion ファイルを利用して、DB を操作します
rails server	Rails アプリを起動します

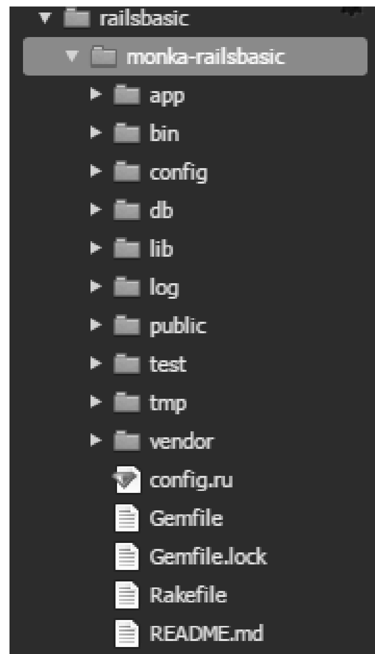
補足説明

アプリを新規作成してから scaffold を作り、マイグレーションを実行してからアプリを立ち上げるまでの手順は以下となります。

```
$ rails new （アプリケーション名）
$ rails generate scaffold User name:string email:string
$ rails db:create
$ rails db:migrate
$ rails server -b 0.0.0.0 -p 8080
```

10.9.2 Ruby on Rails のプロジェクトのディレクトリ構成

ディレクトリ構成まとめ



補足説明

- app

アプリケーション本体が格納されます

- assets

WEB ページ内で使用する image ファイルや、ページのレイアウトに使用する css、ページの動きを制御する js 等が格納されます

- controllers

ユーザアクションを基にアプリを制御する controller が格納されます
MVC の“C”に相当します

- helpers

「ヘルパーメソッド」と呼ばれるメソッドをまとめたファイルが格納されます
「ヘルパーメソッド」とは、主に view を記述する際に役立てるメソッドで、 フォーム要素の生成、文字列や数値の整形するメソッド等、view でよく利用する操作がデフォルトで用意されています 例えば、link_to メソッドでは、与えられた引数を元にハイパーリンクを生成することができます ここでは、独自に使用するヘルパーメソッドを定義します

- models

データの処理全般を管理する model が格納されます
MVC の“M”に相当します

- views

画面に表示する部分の view が格納されます
MVC の“V”に相当します

- config

Rails アプリの設定に関するファイルが格納されます
ルーティングを制御する routes.rb 等が格納されています
ルーティングとは、ブラウザからのリクエスト(URL)をサーバ側の Rails と結びつける仕組みです

- db

DB 関係のファイルが格納されます
DB のテーブルをアプリ側から操作できるようにした migrattion ファイルや、
DB の初期投入データを管理できる seeds.rb 等が格納されています

- test

テスト関係のファイルが格納されます

- Gemfile

アプリで使用する gem をまとめたファイルです
gem の種類だけでなく、バージョンや使う環境を限定できます

10.9.3 ルーティングの設定方法

ルーティングまとめ

- `config/routes.rb` ファイルで、ルーティングの設定ができます。

補足説明

`config/routes.rb` ファイルは以下のようになります。

```
Rails.application.routes.draw do
  # resources だけで、index show new create edit update destroy の 7 つのルーティングが設
    定できるようになっています。
  resources :book
  resources :user do
    member do
      # /users/(:id)/avatar という URL が定義され
      # UsersController#avatar アクションが呼びだされます。
      get :avatar
```

```
    end
    collection do
      # /users/search という URL が定義され
      # UsersController#search アクションが呼びだされます。
      get :search
    end
  end

  # index show new create confirm の 5 つのルーティングを設定しています。
  resources :articles, only: [:index, :new, :create, :show] do
    collection do
      post :confirm
    end
  end

  # /mypage という URL が定義され
  # MypageController#index アクションが呼び出されます。
  get :mypage, to: 'mypage#index'
end
```

10.9.4 ActionController

ActionController まとめ
app/controllers/の配下にあるファイルです
MVC の C(Controller)に相当します

補足説明

ActionController は以下のようになります。

```
class UsersController < ApplicationController
before_action :set_user, only: [:show, :edit, :update, :destroy]

  # GET /users
  # GET /users.json
  def index
    @users = User.all
  end

  # GET /users/1
  # GET /users/1.json
```

```
def show
end

# GET /users/new
def new
  @user = User.new
end

# GET /users/1/edit
def edit
end

# POST /users
# POST /users.json
def create
  @user = User.new(user_params)

  respond_to do |format|
    if @user.save
      format.html { redirect_to @user,
      notice: 'User was successfully created.' }
      format.json { render :show, status: :created, location: @user }
    else
      format.html { render :new }
      format.json { render json: @user.errors,
  status: :unprocessable_entity }
    end
  end
end

# PATCH/PUT /users/1
# PATCH/PUT /users/1.json
def update
  respond_to do |format|
    if @user.update(user_params)
      format.html { redirect_to @user,
      notice: 'User was successfully updated.' }
      format.json { render :show, status: :ok, location: @user }
    else
      format.html { render :edit }
      format.json { render json: @user.errors,
  status: :unprocessable_entity }
    end
  end
end

# DELETE /users/1
# DELETE /users/1.json
def destroy
  @user.destroy
  respond_to do |format|
    format.html { redirect_to users_url,
    notice: 'User was successfully destroyed.' }
    format.json { head :no_content }
  end
end
```

```
  private
    # Use callbacks to share common setup or constraints between actions.
    def set_user
      @user = User.find(params[:id])
    end

    # Never trust parameters from the scary internet, only allow the white list
   through.
    def user_params
      params.require(:user).permit(:name, :email)
    end
end
```

10.9.5 ActiveRecord

ActiveRecord まとめ
Rails に付属する、重要なライブラリの 1 つです
MVC の M(Model)に相当します

補足説明

代表的なメソッドとしては、以下のものがあります。

- all

レコードを全件取得します

- select

カラムを指定し、レコードを取得します。引数の値がカラムとなります。

- find

指定した id のレコードを取得します。引数の値が指定する id となります。
find は、該当するデータが見つからない場合は例外（RecordNotFound）が発生します。

- find_by

特定のカラムの条件を指定し、該当する 1 件を取得します。引数の値が条件となります。
find_by は該当するデータが見つからない場合は、nil を返します。

- where

特定のカラムの条件を指定し、該当する全件を取得します。引数の値が条件となります。
where は、該当するデータが見つからない場合は空の ActiveRecord::Relation を返します。

- first

レコードの最初の 1 件を取得します。引数を渡すと最初の n 件と指定することもできます。

- last

レコードの最後の 1 件を取得します。引数を渡すと最後の n 件と指定することもできます。

- order

レコードを引数に指定したカラムで並び変えます。デフォルトの並び順は ASC(昇順)になっています。
降順で並び変える場合は User.order(name: :DESC)とします。

- limit

特定のレコード件数を取得します。引数の値が最大取得行数となります。

10.9.6 ビュー

ビューまとめ
画面を表示する際に使用されます
MVC の V(View)に相当します

補足説明

ビューは以下のようになります。
<%= %>や<% %>を使うことで HTML の中に Ruby のコードが書けるようになっています。

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

<h1>Users</h1>

<table>
  <thead>
    <tr>
      <th>Name</th>
      <th>Email</th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @users.each do |user| %>
      <tr>
        <td><%= user.name %></td>
        <td><%= user.email %></td>
        <td><%= link_to 'Show', user %></td>
        <td><%= link_to 'Edit', edit_user_path(user) %></td>
```

```
        <td><%= link_to 'Destroy', user,
        method: :delete, data: { confirm: 'Are you sure?' } %></td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>

<%= link_to 'New User', new_user_path %>
```

10.9.7 Heroku へのデプロイ方法

コマンドまとめ

コマンド	説明
heroku login	Heroku へログインします
heroku create	Heroku 上でのアプリケーションの名前を登録します
heroku run	Heroku 上でコマンドを実行します。

補足説明

Heroku へデプロイするまでの手順は以下となります。
なお、Heroku では、デフォルトのデータベースが PostgreSQL となっているため、デプロイ前に対応が必要です。

```
$ heroku login
$ heroku create （アプリケーションの名前）
$ git push heroku master
$ heroku run rails db:migrate
```

10.10 [評価課題]Ruby on Rails　まとめ

ここで、Ruby on Rails のポイントを確認してみましょう。

rails コマンド

myapp という名前の Rails アプリケーションを作るコマンドは rails (Q1: 　　　　　　　　)です。
Rails サーバーを起動するコマンドは rails (Q2: 　　　　　　　　)です。

rails generate コマンドを使うことで、ユーザアクションを基にアプリを制御する(Q3: 　　　　　　　　)、データの処理を管理する(Q4: 　　　　　　　　)などのソースコードを生成することができます。

string 型の name と email カラムと、date 型の born_on カラムを持つリソースを作るには、以下のようなコマンドを実行します。

```
rails generate scaffold User (Q5:                         )
```

rails generate scaffold コマンドを実行すると、ルーティングの設定を持つ config/(Q6: 　　　　　　　　)ファイルも更新され、以下の行が追加されます。

```
Rails.application.routes.draw do
  (Q7:        ) :users # 追加される行
End
```

また、マイグレーションファイルも生成されます。このマイグレーションをテスト環境のデータベースに適用するには、次のコマンドを実行します。

```
rails (Q8:        ) (Q9:        )
```

コントローラ

rails generate コマンドで生成されたコントローラは、デフォルトでは以下の 7 つのアクションを持っています。

- リソースを一覧を表示する(Q10:　　　　　　　　)
- 個別のリソースを表示する(Q11:　　　　　　　　)
- リソースの新規作成画面を表示する(Q12:　　　　　　　　)
- リソースの編集画面を表示する(Q13:　　　　　　　　)
- リソースの作成リクエストを受け取る(Q14:　　　　　　　　)
- リソースの更新リクエストを受け取る(Q15:　　　　　　　　)
- リソースの削除リクエストを受け取る(Q16:　　　　　　　　)

モデル

users テーブルに対するモデルは次のようにして定義することができます。

```
class User < (Q17:          )
end
```

users テーブルから name カラムに田中太郎という値を持つレコードをすべて取得して、born_on カラムで降順に並び替えるには、次のようなコードを実行します。

User.【Q18:　　　　　　　　】(name: '田中太郎').order(【Q19:　　　　　　　　】)

ビュー

user 変数に User クラスのインスタンスが代入されているとき、ERB テンプレートで以下のように書くとユーザーの born_on を表示することができます。

(Q20:　　　　　　　　) user.born_on (Q21:　　　　　　　　)

ユーザーリソースの一覧画面へのリンクは次のコードで作ることができます。

(Q20) (Q22:　　　　　　　　) 'User Index', users_path (Q21)

Heroku へのデプロイ

Heroku は、PaaS（Platform as a Service）と呼ばれるクラウドサービスです。Web サービスを簡単な手順でデプロイすることができます。

myapp というアプリケーションを Heroku に登録して、git を使ってデプロイする手順は次のようになります。

```
$ heroku login
$ heroku (Q23:         ) myapp
$ git (Q24:         ) heroku master
$ heroku (Q25:         ) rails db:migrate
```

2019 年度「専修学校による地域産業中核的人材養成事業」
札幌（北海道）をモデルとした地域創生のための IT 人材育成と企業連携推進事業

■実施委員会

◎橋本　直樹	吉田学園情報ビジネス専門学校　副校長
谷口　英司	日本電子専門学校 情報ビジネスライセンス科科長
北原　聡	麻生情報ビジネス専門学校　校長代行
小幡　忠信	一般社団法人 Ruby ビジネス推進協議会　理事長
岡山 保美	株式会社ユニバーサル・サポート・システムズ　取締役
宇野　哲哉	株式会社サンクレエ　取締役　開発グループ　マネージャー
森　正人	一般社団法人北海道 IT 推進協会　会長
飯塚　正成	一般社団法人全国専門学校情報教育協会　専務理事
小塚　隆	経済産業省　北海道経済産業局　地域経済部　参事官（情報産業・情報化推進担当）

■事業実施分科会

◎岡山 保美	株式会社ユニバーサル・サポート・システムズ　取締役
菅野　崇行	吉田学園情報ビジネス専門学校　情報システム学科
村岡　好久	名古屋工学院専門学校／一般社団法人 TokurouneMono 振興協会　代表理事
谷口　英司	日本電子専門学校　情報ビジネスライセンス科科長
北原　聡	麻生情報ビジネス専門学校　校長代行
宇野　哲哉	株式会社サンクレエ取締役　開発グループ　マネージャー
森　正人	一般社団法人北海道 IT 推進協会　会長
大園　博美	有限会社Ａｒｉｅｓ 代表
井上　浩	一般財団法人 Ruby アソシエーション　副理事長
高畑　道子	株式会社ＦＭ．Ｂｅｅ　代表取締役社長 ／一般社団法人 Ruby ビジネス推進協議会　副理事長
川端　光義	株式会社アジャイルウェア　代表取締役 ／一般社団法人 Ruby ビジネス推進協議会　理事
吉岡　正勝	一般社団法人全国専門学校情報教育協会

■評価委員会

◎飯塚　正成	一般社団法人全国専門学校情報教育協会　専務理事
北原　聡	麻生情報ビジネス専門学校　校長代行
高畑　道子	株式会社ＦＭ．Ｂｅｅ　代表取締役社長 ／一般社団法人 Ruby ビジネス推進協議会　副理事長

2019 年度「専修学校による地域産業中核的人材養成事業」
札幌（北海道）をモデルとした地域創生のための IT 人材育成と企業連携推進事業

Ruby on Rails を利用したアジャイル型システム開発教材

令和 2 年 2 月

学校法人吉田学園（吉田学園情報ビジネス専門学校）
〒060-0063　北海道札幌市中央区南 3 条西 1 丁目
TEL 011-272-6070　　FAX 011-272-6075

●本書の内容を無断で転記、掲載することは禁じます。