WEB+DBシステム（応用編）

第7回(2016年11月10日)

ショッピング・サイトの制作（1/3）

設計・制作の基本方針

※　作った部分が目に見えて、試しながら（ある程度、達成感を感じながら）進める。
（実際の仕事であっても、全く見えないと疲労が倍加する）

※ 「動作検証」を組み込みながら、コーディングの前に仕様を書く。（Rspecを[できる範囲で]実践的に導入する。）

※　メッセージは、基本は英語として多国語化の一つの言語として日本語を入れて行く。

※　画面の修飾は後回しにする。
（WEB Designについては、他に参考資料が多いので、ロジックを作り上げる部分に力点を置く。）

但し、力作であれば加点の対象とします。

※　gitを使ってバックアップを取りながら進める。

これまでに作った部分

これまで導入した部分（説明の都合上、Rspecを先にすべきだが、先に作ってしまった部分がある）

※　ログイン認証（devise）を導入した。

※　商品（merchandises）の登録を導入し、user_rootとした。

※　Welcome画面を作成した。

多国語化は各自の課題

レポート評価の際の「必須」とはしません。

基本は英語で（または全部日本語で）画面が作成されていればそれでOKです。

ですが、機会があったら、コツコツと多国語化を進めてみて下さい。

Gitによるバージョン管理・バックアップと同様に、練習になります。

Merchandises（商品）の登録

:user_rootとして

app/views/merchandises/index.html.erb

を指定した。

この画面では、いきなり商品を表示する。

ここまでの画面テストをどう書くか。

ログイン認証画面の生成

rails generate devise:views

で、認証用の画面を生成する。

（編集可能なファイルを用意する。）

```
[root@cisnote app]# rails generate devise:views
      invoke  Devise::Generators::SharedViewsGenerator
       exist    app/views/devise/shared
      create    app/views/devise/shared/_links.html.erb
      invoke  form_for
       exist    app/views/devise/confirmations
      create    app/views/devise/confirmations/new.html.erb
       exist    app/views/devise/passwords
      create    app/views/devise/passwords/edit.html.erb
      create    app/views/devise/passwords/new.html.erb
       exist    app/views/devise/registrations
      create    app/views/devise/registrations/edit.html.erb
      create    app/views/devise/registrations/new.html.erb
       exist    app/views/devise/sessions
      create    app/views/devise/sessions/new.html.erb
       exist    app/views/devise/unlocks
      create    app/views/devise/unlocks/new.html.erb
      invoke  erb
       exist    app/views/devise/mailer
      create    app/views/devise/mailer/confirmation_instructions.html.erb
      create    app/views/devise/mailer/reset_password_instructions.html.erb
      create    app/views/devise/mailer/unlock_instructions.html.erb
[root@cisnote app]#
```

ログイン認証画面のカスタマイズ

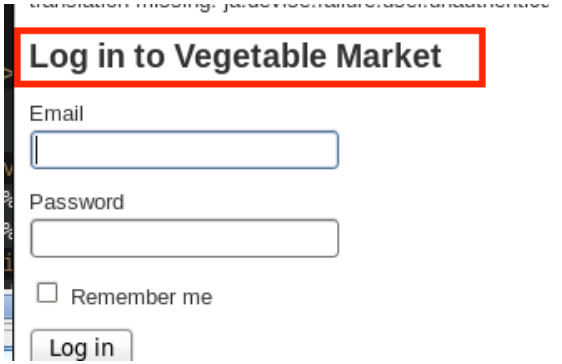
ログイン認証画面を確認するために、見出し表示を編集する。(このファイルを編集したら、この画面が変わったということを確認するためなので、内容は元と違ったものならなんでも良い。)

app/views/devise/sessions/new.html.erb

先頭を

<h2>Log in to Vegetable Market</h2>

にしてみる(後で照合)

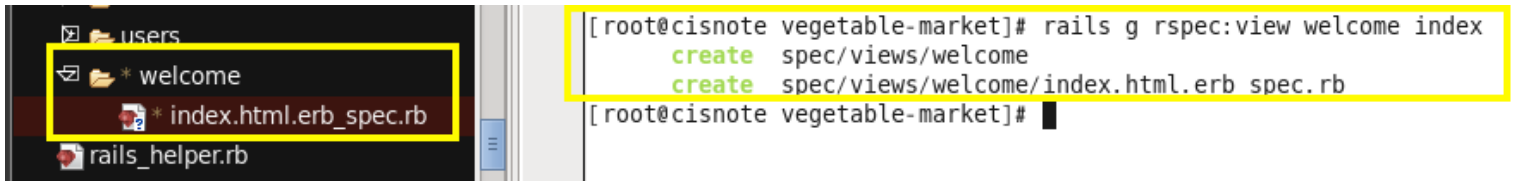


Welcome画面のテストを合成する

Viewのテストの生成は、

rails g rspec:view welcome index

と入力する。（第5回P28参照）



「スタッフ入口」のクリックをテスト

「スタッフ専用入口」→「ここをクリック」の、「ここをクリック」をクリックすると、ログイン認証画面が表示される。

という動作を、テストする。

Spec/views/welcome/index.html.erb_spec.rb

```
require 'rails_helper'

RSpec.describe "welcome/index", type: :view do
  feature 'Welcome画面で' do
    scenario 'Click Hereをクリックすると、ログイン認証画面を開く。' do
      visit root_path    # Welcome画面（root）を開く
     click_on (t :click_here )     #「Click Here」ボタンをクリックすると
     # ログイン認証画面が表示される。
     expect(page).to have_content 'Log in to Vegetable Market'
    end
  end
end
```

Spec/views/welcome/index.html.erb_spec.rb

```
1  require 'rails_helper'
2
3  RSpec.describe "welcome/index", type: :view do
4    feature 'Welcome画面で' do
5      scenario 'Click Hereをクリックすると、ログイン認証画面を開く。' do
6        visit root_path   # Welcome画面（root）を開く
7        click_on (t :click_here )    # 「Click Here」ボタンをクリックすると
8       # ログイン認証画面が表示される。
9        expect(page).to have_content 'Log in to Vegetable Market'
10     end
11   end
12 end
13
```

テストの実行（失敗例）

spec/views/welcome/index.html.erb_spec.rb

のVegetable MarketのVを小文字にして試す。

予想外の画面だと、エラーになる。（テスト記述の誤り）

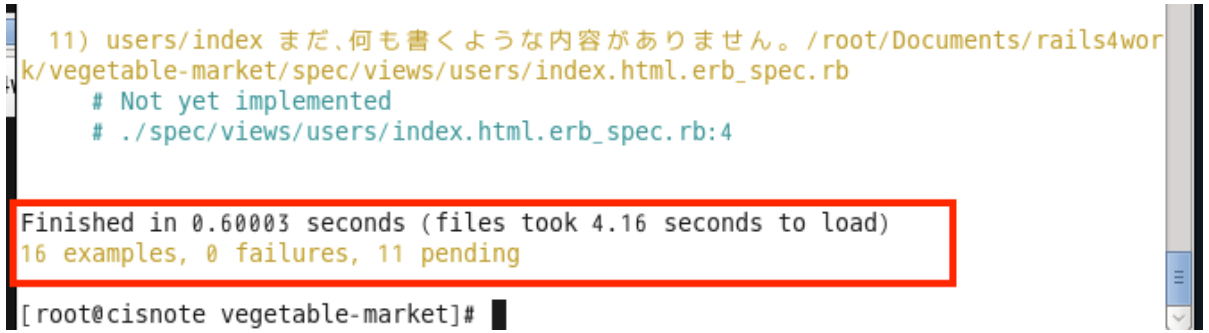
テストの実行（成功例）

spec/views/welcome/index.html.erb_spec.rb

の修正を、元に戻す。（正常動作）エラーがなくなる。

まず「エラー」が起きたことで、テストが動作していることを確認し、プログラムを書いて「No Error」の状態にしながら、プログラムを書いていく。

このテストは今後、「仕様」が変わるまでずっと有効。



ログイン認証のテスト

ログインViewのテストは、

rails g rspec:view devise/sessions new

で生成する。

```
[root@cisnote vegetable-market]# rails g rspec:view devise/sessions new
      create  spec/views/devise/sessions
      create  spec/views/devise/sessions/new.html.erb_spec.rb
[root@cisnote vegetable-market]#
```

Viewテストの書き方：new...rspec.rb

```
require 'rails_helper'

RSpec.describe "devise/sessions/new", type: :view do
  feature 'ログインとログアウト' do
    background do
      # ユーザを作成する
     User.create!(email: 'foo@example.com', password: '12345678')
    end
    scenario 'ログインする' do
      # ログイン画面を開く
      visit new_user_session_path
      # ログインフォームにEmailとパスワードを入力する
     fill_in 'em', with: 'foo@example.com'
      fill_in 'pw', with: '12345678'
       # ログインボタンをクリックする
      click_on "Log in"
      # ログインに成功したことを検証する
      expect(page).to have_content 'Listing Merchandises'
    end
  end
end
```

画面側で、入力BoxにIDを追加

テスト側で指定しているのt同じIDを、それぞれの入力Boxに追記しておく。

```
<h2>Log in to Vegetable Market</h2>

<%= form_for(resource, as: resource_name, url: session_path(resource_name)) do |f|
  <div class="field">
    <%= f.label :email %><br />
    <%= f.email_field :email, autofocus: true, id: :em %>
  </div>

  <div class="field">
    <%= f.label :password %><br />
    <%= f.password_field :password, autocomplete: "off", id: :pw %>
  </div>

  <% if devise_mapping.rememberable? -%>
    <div class="field">
      <%= f.check_box :remember_me %>
```

Viewのテスト:

```ruby
require 'rails_helper'

RSpec.describe "devise/sessions/new", type: :view do
  feature 'ログインとログアウト' do
    background do
      # ユーザを作成する
      User.create!(email: 'foo@example.com', password: '12345678')
    end
    scenario 'ログインする' do
      # ログイン画面を開く
      visit new_user_session_path
      # ログインフォームにEmailとパスワードを入力する
      fill_in 'em', with: 'foo@example.com'
      fill_in 'pw', with: '12345678'
      # ログインボタンをクリックする
      click_on "Log in"
      # ログインに成功したことを検証する
      expect(page).to have_content 'Listing Merchandises'
    end
  end
end
```

Viewのテストと結果

rspec spec/*/*_spec.rb spec/views/*/*_spec.rb spec/views/*/*/*_spec.rb

Deviseの下は、pathが深いことに注意

パスワードや、IDが間違っていたら、ログインできないこともテストすると良い、

```
Finished in 0.73383 seconds (files took 4.35 seconds to load)
17 examples, 0 failures, 11 pending

[root@cignete vegetable_market]# AC
```

入力テストのその他のメソッド

click_link_or_button
click_link　（リンクまたはボタン）
choose　（ラジオボタン）
check　（チェックボックス）
uncheck　（チェックボックス）
select　（ドロップダウンリストなど）
unselect（選択リスト）
attach_file　（ファイルのアップロード）
(Capybara/actions.rb)

ここまでで、カピバラは終わり。

Merchandises（商品）の登録

:user_rootとして

app/views/merchandises/index.html.erb

を指定した。

この画面では、いきなり商品を表示する。

この画面でデータがどう処理されるか。

Index.html.erb

application.rb | en.yml | ja.yml | index.html.erb

```
<h1><%= t :list_merchandises %></h1>

<table>
  <thead>
    <tr>
      <th><%= t 'category_name' %></th>
      <th><%= t 'merchandise_name'%></th>
      <th><%= t 'merchandise_price'%></th>
      <th><%= t 'merchandise_max_order'%></th>
      <th colspan="3"></th>
    </tr>
  </thead>

  <tbody>
    <% @merchandises.each do |merchandise| %>
      <tr>
        <td><%= merchandise.category %></td>
        <td><%= merchandise.name %></td>
        <td><%= merchandise.price %></td>
        <td><%= merchandise.maxOrder %></td>
        <td><%= link_to 'Show', merchandise %></td>
        <td><%= link_to 'Edit', edit_merchandise_path(merchandise) %></td>
        <td><%= link_to 'Destroy', merchandise, method: :delete, data: { confirm: 'Are you sure?' } %></td>
      </tr>
    <% end %>
  </tbody>
</table>

<br>
<%= link_to('Sign out', destroy_user_session_path, :method => :delete) %>
<br />
<%= link_to 'New Merchandise', new_merchandise_path %>

```

Index.html.erb 先頭部分

- <h1>見出し</h1>
  - このページの見出しが記載されている。
- <table><thead><tr>…</tr></thead>
  - 私の例では、14行目まで。
  - 設定したい項目の、「列見出し」が列挙されている。

- 今後、「多国語化」で<%= %>のrubyのタグが入って来る。

一覧表示のための@merchandises

私の例では17行目

```
<% @merchandises.each do |merchandise| %>
```

@merchandisesに、コントローラからデータが渡される。

ここにデータを渡すのは、

app/controllers/merchandises_controller.rb

の、indexメソッド。

merchandisesの画面制御は、このControllerに集められている。

ViewとControllerは名前で対応

- Views/merchandisesの下には、
  - index.html.erb, show.html.erb,
    new.html.erb, edit.html.erb などがあった。
- このファイル名は、MerchandisesControllerの中の
  メソッド名に、直接対応している。
  - うっかり片方だけ変えてしまうと、うまく動かなくなる。
- 今、index.html.erbを調べている ➔
  - merchandises_controller.rbの中の、def indexの記述を調べる。
  - def indexは、indexというメソッドを定義する、という記述で、endまでがこの記述。

Controllerからの流れ

MerchandisesControllerのindexメソッドの

@merchandises = Merchandise.all

で、@merchandises（ハッシュ配列）に
Merchandiseクラスの全データが渡される。

```
6  def index
7    if current_user
8      if !current_user.role_id
9        current_user.role_id = 3
10       current_user.handle = "No Name"
11       current_user.save
12       redirect_to :controller=>"users", :action=>"edit",
13               :id=> current_user.id
14       return
15     end
16   end
17   @merchandises = Merchandise.all
18 end
```

イテレータによる反復（17行目）

<% @merchandises.each do |merchandise| %>

- 私の例では17行目

- この先頭にある、@merchandisesはMerchandiseから .allで取り出された「ハッシュの配列」が渡される。
  - 「ハッシュ」とは「連想配列」
- .eachはイテレータで、この1行は「@merchandisesという配列から、レコードを一つずつ取り出してmerchandiseという呼び名で示し、25行目のendまでに記された内容を反復する」という処理を意味する。

HTMLの記述を覗いてみる

```
<tbody>
  <% @merchandises.each do |merchandise| %>
    <tr>
      <td><%= merchandise.category %></td>
      <td><%= merchandise.name %></td>
      <td><%= merchandise.price %></td>
      <td><%= merchandise.maxOrder %></td>
      <td><%= link_to 'Show', merchandise %></td>
      <td><%= link_to 'Edit', edit_merchandise_path(merchandise) %></td>
      <td><%= link_to 'Destroy', merchandise, method: :delete, data: { confirm: 'Are you sure?' } %></td>
    </tr>
  <% end %>
 </tbody>
</table>

<br>
<%= link_to('Sign out', destroy_user_session_path, :method => :
<br />
<%= link_to 'New Merchandise', new_merchandise_path %>
```

Rubyでは,printやput, pで
文字列を表示するが、不要

<%= %>で、=が入っている。
<% @merchandises.each %>や、
<% end% >には、=がない。

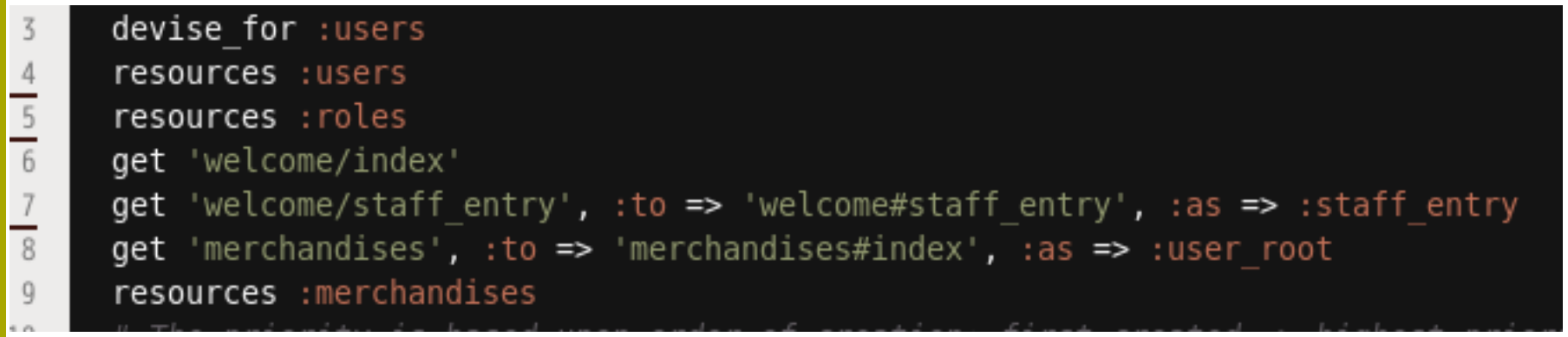
各行の最後に「リンク」を表示する、
という処理を、イテレータeachによって
endまで、1レコードごとに
反復している。

Link先のパスはどう読むか。

config/routes.rb

で記述されている。ここからパスに展開される。

```
devise_for :users
 resources :users
 resources :roles
 get 'welcome/index'
 get 'welcome/staff_entry', :to => 'welcome#staff_entry', :as => :staff_entry
 get 'merchandises', :to => 'merchandises#index', :as => :user_root
 resources :merchandises
 # The priority is based upon
```



Routesの名称

コマンドプロンプトで、

rake routes

と入力する。

```
[root@cisnote vegetable-market]# rake routes
                  Prefix Verb   URI Pattern                       Controller#Action
        new_user_session GET    /users/sign_in(.:format)          devise/sessions#new
            user_session POST   /users/sign_in(.:format)          devise/sessions#create
    destroy_user_session DELETE /users/sign_out(.:format)         devise/sessions#destroy
           user_password POST   /users/password(.:format)         devise/passwords#create
       new_user_password GET    /users/password/new(.:format)     devise/passwords#new
      edit_user_password GET    /users/password/edit(.:format)    devise/passwords#edit
                         PATCH  /users/password(.:format)         devise/passwords#update
                         PUT    /users/password(.:format)         devise/passwords#update
cancel_user_registration GET    /users/cancel(.:format)           devise/registrations#cancel
       user_registration POST   /users(.:format)                  devise/registrations#create
   new_user_registration GET    /users/sign_up(.:format)          devise/registrations#new
  edit_user_registration GET    /users/edit(.:format)             devise/registrations#edit
                         PATCH  /users(.:format)                  devise/registrations#update
                         PUT    /users(.:format)                  devise/registrations#update
                         DELETE /users(.:format)                  devise/registrations#destroy
                   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
                   roles GET    /roles(.:format)                  roles#index
                         POST   /roles(.:format)                  roles#create
                new_role GET    /roles/new(.:format)              roles#new
               edit_role GET    /roles/:id/edit(.:format)         roles#edit
                    role GET    /roles/:id(.:format)              roles#show
                         PATCH  /roles/:id(.:format)              roles#update
                         PUT    /roles/:id(.:format)              roles#update
                         DELETE /roles/:id(.:format)              roles#destroy
           welcome_index GET    /welcome/index(.:format)          welcome#index
             staff_entry GET    /welcome/staff_entry(.:format)    welcome#staff_entry
```

rake routesの読み方

```
                role GET    /roles/:id(.:format)          roles#show
                 PATCH  /roles/:id(.:format)          roles#update
                   PUT    /roles/:id(.:format)          roles#update
                DELETE /roles/:id(.:format)          roles#destroy
welcome_index GET    /welcome/index(.:format)      welcome#index
           user_root GET    /merchandises(.:format)       merchandises#index
       merchandises GET    /merchandises(.:format)       merchandises#index
                  POST   /merchandises(.:format)       merchandises#create
new_merchandise GET    /merchandises/new(.:format)   merchandises#new
edit_merchandise GET   /merchandises/:id/edit(.:format) merchandises#edit
        merchandise GET    /merchandises/:id(.:format)   merchandises#show
```

左から順に

- rubyのプログラム中での呼び名
- HTMLのメソッド
- URL
- 処理を実行するcontrollerとメソッド

（deviseのメソッドのソースは、修正できない。）

HTTPコマンドと、メソッドの対応

- merchandises_controller.rb の コメントとメソッド名を対照させてみてみる。

GET /merchandises	index
GET /merchandises/1	show
GET /merchandises/new	new
GET /merchandises/1/edit	edit
POST /merchandises	create
PATCH /merchandises/1	update
DELETE /merchandises/1	destroy

HTTP/1.1 Method

- HTTPで伝送されるメソッドは、GET/POST/PUT/DELETEなどになる。
- コマンド文字列がDeleteになり、それを受け取った際のアクションは、サーバ側で決められる。

HTTPメソッドの一覧

メソッド	HTTP/0.9	HTTP/1.0	HTTP/1.1
GET	○	○	○
POST		○	○
PUT		△	○
HEAD		○	○
DELETE		△	○
OPTION			○
TRACE			○
CONNECT			○

http://ja.wikipedia.org/wiki/Hypertext_Transfer_Protocol

Link_toの読み方(newの場合）

私のサンプルでは33行目

link_to　'New Merchandise, new_merchandise_path

- コマンドラインでの

rake routes

- の表示では

new_merchandise GET　/merchandises/new(.:format) merchandises#new

- 画面表示の’New Merchandise’を　new_merchandise_pathにリンクさせると、このリンクのクリックで、HTMLのGETメソッドで、/merchandises/newのURLを開き、merchandises_controller.rbのnewメソッドで処理される。
- その結果は、同名のviews/merchandises/new.html.erbに渡される。

new.html.erb と edit.html.erb

- 3行目
- render 'form’
- views/merchandises/_form.html.erbが呼び出される。
- newもeditも、どちらも同じ書式でformを呼び出す

Controllerでは、newの場合

@merchandise = Merchandise.new

で、@merchandiseに値が設定されて_formで編集用にデータが渡される。

editの場合は、指定がなくなった。

_form.html.erb

- 1行目で、フォームに表示する元のデータを記述している。
  - _form.html.erbの1行目で展開される作成データは、それぞれ、newメソッドやeditメソッドで制作されたハッシュなので、フォームの書き込みボタンが押された際に、newからならばPOST, editからばPUTという異なるHTTPメッセージが作成される。

```
POST   /merchandises(.:format)          merchandises#create
 PUT   /merchandises/:id(.:format)      merchandises#update
```

  - この結果、newの場合にはcontrollerのcreateメソッドで、editの場合はupdateメソッドで処理される。

Newによる書き込み

```
1  <%= form_for(@merchandise) do |f| %>
2    <% if @merchandise.errors.any? %>
3      <div id="error_explanation">
4        <h2><%= pluralize(@merchandise.errors.co
```

```
18   <div class="field">
19     <%= f.label :name %><br>
20     <%= f.text_field :name %>
21   </div>
```

fieldのclassでは、@merchandise のそれぞれの要素に値が設定され、merchandise_params]で取り出される。

Createメソッドで処理される。

```
# POST /merchandises
 # POST /merchandises.json
 def create
   @merchandise = Merchandise.new(merchan

   respond_to do |format|
     if @merchandise.save
       format.html { redirect_to @merchandise, notice: 'Merchandise was
    successfully created.' }
       format.json { render :show,      us: :created, location: @merchandise }
     else
       format.html { render :n
       format.json { render j           chandise.errors,
    status: :unprocessabl
     end
   end
 end
```

merchandise GET /merchandises/:id(.:format) merchandises#show

Saveに成功すると、merchandises#showにredirectされる。

プロトコル表示

実際のデータの流れは、端末に表示される。

```
Started POST "/merchandises" for 127.0.0.1 at 2014-11-05 02:01:45 +0900
Processing by MerchandisesController#create as HTML
  Parameters: {"utf8"=>"✓", "authenticity_token"=>"CFN0BwYLenbCTPK1h0989IlVyL7DSx88T+XS85wOqJE=", "merchandi
se"=>{"category"=>"野菜", "name"=>"観賞用かぼちゃ", "price"=>"1200", "maxOrder"=>"8"}, "commit"=>"登録する"}
   (0.1ms)  begin transaction
  SQL (0.9ms)  INSERT INTO "merchandises" ("category", "created_at", "maxOrder", "name", "price", "updated_a
t") VALUES (?, ?, ?, ?, ?, ?)  [["category", "野菜"], ["created_at", "2014-11-04 17:01:45.130438"], ["maxOrd
er", 8], ["name", "観賞用かぼちゃ"], ["price", 1200], ["updated_at", "2014-11-04 17:01:45.130438"]]
   (1.5ms)  commit transaction
Redirected to http://127.0.0.1:3000/merchandises/2
Completed 302 Found in 12ms (ActiveRecord: 2.5ms)


Started GET "/merchandises/2" for 127.0.0.1 at 2014-11-05 02:01:45 +0900
Processing by MerchandisesController#show as HTML
  Parameters: {"id"=>"2"}
  Merchandise Load (0.3ms)  SELECT  "merchandises".* FROM "merchandises"  WHERE "mercha
T 1  [["id", 2]]
  Rendered merchandises/show.html.erb within layouts/application (1.9ms)
  Rendered shared/_menu_bar.html.erb (0.8ms)
  Rendered shared/_right_bar.html.erb (0.0ms)
  Rendered shared/_footer.html.erb (0.1ms)
Completed 200 OK in 68ms (Views: 63.2ms | ActiveRecord: 0.3ms)
```

HTMLからのパラメータ

params[]でパラメータを取り出すことができる。

HTML側のformで渡されるものでは、
@merchandiseの他に

```
<%= tag :input, { :type => 'hidden', :name =>
  'accessory', :value => 'flower arrangement' } %>
```

などのように、input tagの形式で、params[]に名前と値を手渡すことができる。

この例はparams[:accessory]に"flower arrangement"というデータが渡される。Hidden属性なので、入力用のBoxは表示されない。

Modelを持たない画面と制御

Welcome画面と同様に、generatorでviewとcontrollerを生成させる。

特定のModelのClassを保存するのではなく、処理内容に応じて、Controllerでデータを記録する。

作りたいシステムのイメージ

私の場合には、「野菜の販売サイト」です。

（皆さんは各自のイメージで、読み替えて作って下さい。）

出来上がり形を、画面から考えてみる。

※　こんなことを画面に出したい、というものを考えて、機能をイメージする。

テーブル設計の要点

第３正規形になるように設計する。

　リレーションを適切に張って、データ間の関係を整理する。

リレーションとなる情報は、最初からクラス名＋IDで名前をつける。

例：memosに、categoryへのリレーションを持たせる場合、フィールド名は

　category_id　:integer

となる。

Camel型とSnake型

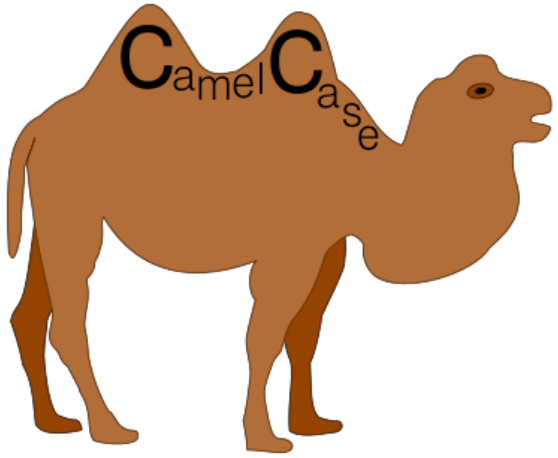
テーブル設計では、クラス名はCamel型、ファイル名や、メソッド名はsnake型となる点に注意して下さい。

Camel型（Camel Cace）の例

ActiveSupport

Snake Caseの例

confirmed_orders



実体型と関連型（前期8回P4再掲）

Relation ：「関連」/Table（表のこと）

呼び名は「関連」だが

実体を表すもの

関連を表すもの

の、2種類がある。

実体： Entity

対象実世界（物理的な存在でないものもある）

関連：Relation

実体間の関連

実体型とは（前期8回P5再掲）

Entity Type:

実世界に物理的に存在するもの

学生、教員、教室、ピザ（メニュー）、

実世界で、他と「区別して識別」できるもの

大学、授業科目、予約のフライト便、サークル、

「識別される」実体

大学の「建物」は物理的に存在するが、「大学」という組織は必ずしも「建物」を必要としない。

関連型とは（前期8回P6再掲）

Relation Type

二つの実体の関係を表すもの

「履修」：学生が科目を履修する。

「予約」：顧客がフライトを予約する。

「注文」：客がピザを注文する。

「レンタル」：客がDVDをレンタルする。

一人の客が複数のDVDをレンタルできる。

一人の客は複数のピザを注文できる。

学生は、複数科目を履修できる。

「メーカー」の用語について

メーカーは和製英語です。（一応通じるようですが・・）

一般に工業製品の場合に「メーカー」は

Manufacturer

になります。できるだけ正しい英語を使うように、「用例」などが豊富な辞書を用意してみて下さい。

私の普段使いは、Mac純正のWisdomで、お気に入り（おススメという訳ではありませんが）は alc です。

または、辞書ソフトをインストールしてみて下さい。

販売日などのフィールド

日付については、String型ではなく、timestamp型を使うのが一般的だと思います。

理由は、必要に応じて日付表示のフォーマット指定であるとか、日数計算のメソッドなど、日付処理に必要なメソッドが使える、ということがあります。

但し、「日数計算」などの対象にならない場合（例えば、CD/DVDのリリース日など）は、String型でもいいと思いますが、「発売から1年以上経ったCD/DVDを抽出」などという処理が将来的にありそうならば、やはりtimestamp型がいいと思います。

どんな感じの画面にしたいか！

Amount: [ ] Add to Cart

Cucumber (x5) 120　Organic

Amount: [ ] Add to Cart

Your Shopping Carts

carrot

@ 120 x 5 units. Subtotal: 600 Yen

Egg Plant(x3)

@ 120 x 3 units. Subtotal: 360 Yen

Total: 960 Yen.

機能と手順を考える。

前期の復習で、まずは「商品登録」／「商品一覧」の画面を先に作ってしまった。

ログインしない人でも、ショッピングカートに色々と買いたいものを入れてもらいたい。

→　次は、ショッピングカートを設計しようか。

→　商品登録は「店員」だけになるようか。

→　商品に「写真」を入れてみようか。

Trial and Errorでとにかく作りながら考えよう！

作るべき画面（主だったもの）

一般ユーザにとっては・・・

- 商品一覧
- ショッピングカート（取り消し/合計額確認など）
- 支払い確認
- 発送状態の確認（今、クロネコの〇×支店など）
- About Us（誰がこんな店出してるの？）

こんな画面にしたい！

Amount: [ ] Add to Cart

Cucumber (x5) 120　Organic

Amount: [ ] Add to Cart

Your Shopping Carts

carrot

@ 120 x 5 units. Subtotal: 600 Yen

Egg Plant(x3)

@ 120 x 3 units. Subtotal: 360 Yen

Total: 960 Yen.

作るべき画面（主だったもの）

ショップの店員（オーナー）にとっては

- 販売する商品の登録
- 登録した顧客の一覧
- 支払い状況の確認
  - （出荷しなければならない商品の確認）
- 出荷するための処理
- 売り上げ状況の確認
- 在庫の確認

作るべき画面（主だったもの）

システム管理者にとっては

- マスターメンテナンス
- 「店員」の登録
  - 誰もが勝手に「店員」になっては困る。

作るべきテーブル

Cart(ショッピングカート)

CustomerDetail(顧客情報詳細（送り先住所など）)

PurchaseStatus(購入（出荷）状況一覧)

Sales(売り上げ状況)

他にもあるかも知れない、あるいは、設計変更があるかも知れない。

でも、とにかく作ってみて、試してみて、考えてみよう。

ダメなら設計変更すればいい。

機能を考える。

顧客は、商品を見て、ショッピングカートに入れる。
ショッピングカートの中身をみて、再考する。
ショッピングカートの中身を確認後、レジに進む。
支払い手続きを行う。
店員は、売り上げ状況を確認し、出荷処理を行う。
出荷したら、追跡番号を入力する。
売り上げ集計を行って、会計処理ができるようにする。
在庫状況を確認し、必要ならば仕入れ手配できるようにする。

開発手順(1)

前期の復習

- 画面を分割して、ヘッダ、フッタ、リンクバーを用意
- 商品に画像を添付できるようにする。

ショッピングカートの画面を作る。

- ここから、テスト手順を意識する。

過去に遡って、Welcome画面などに[rspec]による
テストを記述する。

支払いについて

- クレジット決済
- コンビニ決済
- 代金引換

など

「決済代行」で検索

細かい処理や手続きがあるが、ここでは「決済代行」にリンクしたものとして、「支払い」ボタンを押したら、そのまま「支払済」に移行するものとして設計する。

開発手順(2)

顧客は、「本気で買うぞ」と思ったら、「レジに進む」ボタンを押して、自分の住所などを入力し、「支払い」ボタンを押す。

店員は、「支払済」の商品について、直ちに「出荷」処理を行う。(これは、オフライン;自動倉庫での出荷システムがあったとしても、ここでは別設計)

出荷が済んだら、「追跡番号」を入力し、顧客が「商品が今どうなっているか」わかるようにする。

この順番で、実装して行く。

何かを作るときは・・・

卒研もそうですが・・・

そのプロジェクトごとに、開発のための「ノート」を一冊用意すると良い。(電子的でも良いと思いますが・・・)

Rspecが「ノート」の代わりになるかも知れませんが未確認なので・・・。

「こんな機能があったらいい」とか、「ここの動作がおかしい」など、気付いた点をノートに書き留めておく。

機能の追加は「思いつき」ではなく「影響範囲」を熟考して行う。(ノートに整理して行く)

PCをノート代わりにしている人は、プロジェクト用のメモファイルを作る。(IDEで代用)

Specを「設計文書」として使う

これまで見てきた「こんな動作をさせる」という内容を、最初にSpecに書いていく。

そのテストを書いてから、プログラムを動作させて、動作を確認する。

権限の制約のテストを書く

Devise::TestHelpersの組み込みが必須で、

http://konyu.hatenablog.com/entry/2014/11/12/230433

現在、サンプル作成中です。

上記参考サイトを見て、tryしてみて下さい。

テストの中身は別にして、まず、どんな動作をさせたいか、specを一つでも多く書いてみて下さい。

欠席課題

CapybaraによるviewのテストのSpecファイルを作成し、その実行結果を添付して下さい。