Ruby on Rails 101 – Chapter 7

In the chapter, we are going to take a look on the JSON output and also introducing the Behavior-driven development, BDD, approach.

Introducing jbuilder

In the past, we output JSON in controller.

```
format.json { render json: @jobs }
```

But it has been like a hacking with lot's of nesting block while the JSON format grows.

But the controller JSON story is over. Now we can define how the data is formatted in JSON, as same as how the data is organized in HTML. This are done by a gem named "JBuilder" from the official rails team. And it is now pre-installed in rails version 4.

In the `Gemfile`, uncomment the following line and `bundle install` it.

```
gem 'jbuilder'
```

JBuilder gem uses file extension `.json.jbuilder`. This follows the extension convention like `.html.erb`.

Using jbuilder

In the `.json.jbuilder` file, we use `json` object to defile the output JSON node. the syntax in `json.any_name`, Then passing the value as argument.

For example, `jsor.id(@album.id)` will output `{id: 123}`

So, for the album show API, we can create a file named `show.json.jbuilder` inside `views/albums/` folder.

```
json.id @album.id
json.title @album.title
```

Or we can create a new node by using the `json.name do |json| end` block.

The following is an album information with the user basic information.

```
json.id @album.id
json.title @album.title
json.created_at @album.created_at
json.link album_url(@album)
json.user do |json|
  json.id @album.user.id
  json.email @album.user.email
end
```

How about listind all the photos in the album?

```
# photos
json.photos @album.photos do |json, photo|
  json.id photo.id
  json.title photo.title
  json.image_url request.protocol +
request.host_with_port + photo.file.url
  json.thumb_url request.protocol +
request.host_with_port + photo.file.url(:thumb)
end
```

The image_url is a little but tricky, the paperclip gem returns relative url only.

we can prepend the URL to the relative path, and the URL is composited by the `protocol` and `host_with_port`

If you feel uncomfortable with the long string concatenation here, helper is the place to abstract that long line.

Extracting view code to helpers

Helper is used to abstract view methods. It is designed for using in view.

`app/helpers/photos_helper.rb` file:

```
module PhotoHelper
  def absolute_photo_url(photo, style=nil)
    request.protocol + request.host_with_port +
photo.file.url(style)
  end
end
```

So now, the photo url can be like this:

```
json.thumb_url absolute_photo_url(photo, :thumb)
```

Partial json views

How about we need the photos JSON output in several API?

We can use partial too in the JSON JBuilder. Same as ERB partial that begins with an underscore `_partial.html.erb`.

Here is our `_photo.json.jbuilder` file. Please note that it only describe one photo object because the parent file handles the loop.

```
json.id photo.id
json.title photo.title
json.image_url absolute_photo_url(photo)
json.thumb_url absolute_photo_url(photo, :thumb)
```

And here is the full `show.json.jbuilder` file after using the partial.

```
json.data do |json|
  json.id @album.id
  json.title @album.title
  json.created_at @album.created_at
  json.link album_url(@album)

  # user
  json.user do |json|
    json.id @album.user.id
    json.email @album.user.email
  end

  # photos
  json.photos @album.photos do |json, photo|
    json.partial! photo
  end
end
```

Note: Since `json.any_name` will create `any_name` node, all json special method – such as `json.partial!` – uses the `!` mark. Normally the `!` mark is used to indicate dangerous method that changes itself.

Introducing behavior driven development

What is BDD?

BDD, Behavior-driven development, means we define how the users behave when using our software. How they interact with the app. And we only write the first single line of code when such interaction is defined into the feature spec.

Before getting into the BDD, we want to create a new empty project to demonstrate the test first approach. We don't want to write our code and then add tests to test our code. Instead, we write the features spec at the beginning and write code to meet the spec requirement.

We are going to re-creating the photo album.

Let's create a new project named `gallery` and migrate the DB the first time.

```
$ rails _3.2.8_ new gallery
$ rake db:migrate
```

Installing cucumber for rails

Add the following test group to `Gemfile` which contains `cucumber-rails` gem and others related gems.

```
group :test do
  gem 'cucumber-rails', :require => false
  gem 'database_cleaner'
  gem 'rspec-rails'
end
```

Then install the bundle to the system.

```
$ bundle install
```

Cucumber requires an installation before using it. Run the following cucumber:install command.

```
$ rails generate cucumber:install
    create  config/cucumber.yml
    create  script/cucumber
     chmod  script/cucumber
    create  features/step_definitions
    create  features/support
    create  features/support/env.rb
     exist  lib/tasks
    create  lib/tasks/cucumber.rake
      gsub  config/database.yml
      gsub  config/database.yml
     force  config/database.yml
```

or alternatively, you can check the options before executing the installation script.

```
$ rails generate cucumber:install --help
```

It makes use of the Capybara API. So, if you want to master the cucumber, you need to check all the tools available in the Capybara.

Writing our first feature

Let's create the our first feature. Create a new file named `basic.feature` under the folder `features/` with the following content.

```
Feature: Basic
  Scenario: In the homepage
    Given I am on homepage
    Then I should see "Welcome"
```

Now we can run `rake cucumber` to test the scenario.

And here is the result.

```
Feature: Basic

  Scenario: In the homepage     #
features/basic.feature:2
    Given I am on homepage      #
features/basic.feature:3
      Undefined step: "I am on homepage"
(Cucumber::Undefined)
      features/basic.feature:3:in `Given I am on
homepage'
    Then I should see "Welcome" #
features/basic.feature:4
      Undefined step: "I should see "Welcome""
(Cucumber::Undefined)
      features/basic.feature:4:in `Then I should see
"Welcome"'

1 scenario (1 undefined)
2 steps (2 undefined)
0m0.398s

You can implement step definitions for undefined steps
with these snippets:

Given(/^I am on homepage$/) do
  pending # express the regexp above with the code you
wish you had
end
```

```
Then(/^I should see "(.*?)"$/) do |arg1|
  pending # express the regexp above with the code you
wish you had
end
```

The cucumber is cleaver enough to find all non-handle English sentense and tell us how to handle them. And it will replace double-quoted string into variable. And rerember, the steps are just regular expression. Feel free to change it to whatever that works for you.

So the next step is to create `step` files to describe how to deal with the feature sentences.

Create a new file named `basic_steps.rb` under `features/step_definitions` folder.

Then we copy the cucumber suggestions into the step file.

```
Given(/^I am on homepage$/) do
  pending # express the regexp above with the code you
wish you had
end

Then(/^I should see "(.*?)"$/) do |arg1|
  pending # express the regexp above with the code you
wish you had
end
```

When we run the `rake cucumber` command now, we see the step is reconized and in `pending` state.

```
Feature: Basic

  Scenario: In the homepage     #
features/basic.feature:2
    Given I am on homepage      #
features/step_definitions/basic_steps.rb:1
      TODO (Cucumber::Pending)
      ./features/step_definitions/basic_steps.rb:2:in
`/^I am on homepage$/'
      features/basic.feature:3:in `Given I am on
homepage'
    Then I should see "Welcome" #
features/step_definitions/basic_steps.rb:5

1 scenario (1 pending)
2 steps (1 skipped, 1 pending)
0m0.439s
```

Let's implement the first pending step with the following code.

```
Given(/^I am on homepage$/) do
  visit root_path
end
```

Now we get a fail when running `rake cucumber`.

```
Feature: Basic

  Scenario: In the homepage     #
features/basic.feature:2
    Given I am on homepage      #
features/step_definitions/basic_steps.rb:1
      undefined local variable or method `root_path'
for #<Cucumber::Rails::World:0x007f8b6170afc8>
(NameError)
      ./features/step_definitions/basic_steps.rb:2:in
`/^I am on homepage$/'
      features/basic.feature:3:in `Given I am on
homepage'
    Then I should see "Welcome" #
features/step_definitions/basic_steps.rb:5

Failing Scenarios:
cucumber features/basic.feature:2 # Scenario: In the
homepage

1 scenario (1 failed)
2 steps (1 failed, 1 skipped)
0m0.322s
```

It is because we haven't modify our root page yet.

Now turn to the code ane create a `pages` controller with an `index` action.

```
$ rails generate controller pages index
```

And we add the `root_path` in the `routes.rb` file.

```
Gallery::Application.routes.draw do
  root to:'pages#index'
end
```

Finally we remove the `public/index.html` file.

Running `rake cucumber` now will pass the first step "Given I am on

homepage", and leave the second one "Then I should see "Welcome""

So the next step is to implement the remaining pending steps.

```
Then(/^I should see "(.*?)"$/) do |wording|
  page.should have_content wording
end
```

And we get the error "expected to find text "Welcome"" after running the `rake cucumber` again.

```
Feature: Basic

  Scenario: In the homepage     #
features/basic.feature:2
    Given I am on homepage      #
features/step_definitions/basic_steps.rb:1
    Then I should see "Welcome" #
features/step_definitions/basic_steps.rb:5
      expected to find text "Welcome" in "Pages#index
Find me in app/views/pages/index.html.erb"
(RSpec::Expectations::ExpectationNotMetError)
      ./features/step_definitions/basic_steps.rb:6:in
`/^I should see "(.*?)"$/'
      features/basic.feature:4:in `Then I should see
"Welcome"'

Failing Scenarios:
cucumber features/basic.feature:2 # Scenario: In the
homepage

1 scenario (1 failed)
2 steps (1 failed, 1 passed)
0m0.564s
```

So let's add the "Welcome" to the root page. Then running the `rake cucumber` again will give us all passes.

```
Feature: Basic

  Scenario: In the homepage     #
features/basic.feature:2
    Given I am on homepage      #
features/step_definitions/basic_steps.rb:1
    Then I should see "Welcome" #
features/step_definitions/basic_steps.rb:5

1 scenario (1 passed)
2 steps (2 passed)
```

```
0m0.520s
```

The idea of test-driven is to write the test first by describing how the work should work. Then obviously it is going to fail. And we observe what make the test fails and write code to make it just work. At last, we refractor the code to make it work in more generic way. Then we go to the next pending/failing case.

Writing the user authentication component

It is just a beginning. Assumning now we have the following feature specs defined in an `authenticate.feature` file.

```
Feature: Authenticate
  Scenario: Before login
    Given I am not logged in
    When I go to homepage
    Then I should see "Sign in"
    And I should see "Sign up"

  Scenario: Login Page
    Given I am not logged in
    And I am on homepage
    When I click "Sign in" link
    Then I should be in the sign in page

  Scenario: Login action
    Given I have an account
    And I am not logged in
    And I am on sign in page
    When I fill in correct account information
    And press "Sign in" button
    Then I should see "Signed in successfully"

  Scenario: Logged in
    Given I have logged in
    When I go to homepage
    Then I should see "Sign out"

  Scenario: Logout action
    Given I have logged in
    And I am on homepage
    When I click "Sign out" link
    Then I should see "Sign in"
    And I should see "Signed out successfully"
```

This feature can be written by you, the developer, or by the project planner who doesn't know programming at all.

Now when we run the `rake cucumber`, we get a large block of response, with lots of pending.

We can implement them one by one.

First comes with the basic steps about page nevigation and mouse clicking.

```
When(/^I go to homepage$/) do
```

```
  visit root_path
end

When(/^I click "(.*?)" link$/) do |link|
  click_link link
end

When(/^press "(.*?)" button$/) do |button|
  click_button button
end
```

In the authentication,

```
Given(/^I am not logged in$/) do
  visit '/users/sign_out'
end
```

Note that we use the URL instead of the devise path `destroy_user_session_path`. It is because during the feature and step writing, we are designing the interface interaction, including the URL. It is not nessesary the authentication must happen with devise gem as long as it works.

And the full `authenticate_steps.rb` file.

```
email = 'test@example.com'
password = 'thisisasecret'

Given(/^I am not logged in$/) do
  visit '/users/sign_out'
end

Then(/^I should be in the sign in page$/) do
  current_path.should == '/users/sign_in'
end

Given(/^I have an account$/) do
  User.new(email:email, password:password).save!
end

Given(/^I am on sign in page$/) do
  visit '/users/sign_in'
end

When(/^I fill in correct account information$/) do
  fill_in 'Email', with: email
  fill_in 'Password', with: password
end

Given(/^I have logged in$/) do
  User.new(email:email, password:password).save!
  visit '/users/sign_in'
  fill_in 'Email', with: email
  fill_in 'Password', with: password
  click_button 'Sign in'
end
```

Absolutely we will get failed scenarios because we have not written any code on authenticatation. It is time to plug the devise gem in.

In Gemfile

```
gem 'devise'
```

And `bundle install` it.

```
$ rails generate devise:install
```

Follow the post-installation instruction to add the notice and alert hook to the application.html.erb file.

```
<p class="notice"><%= notice %></p>
<p class="alert"><%= alert %></p>
```

Then we create the User model with the devise authentication. Don't forget to `rake db:migrate` before proceeding.

```
$ rails generate devise User
```

The setup is ready and we can now re-running the `rake cucumber` to tackle the failed cases.

The first error we encounter is `No route matches [GET] "/users/sign_out" (ActionController::RoutingError)`. It is because the devise's sign out route uses DELETE method. We can change it at the `devise.rb` config file.

```
config.sign_out_via = :delete
config.sign_out_via = [:get, :delete] if Rails.env.test?
```

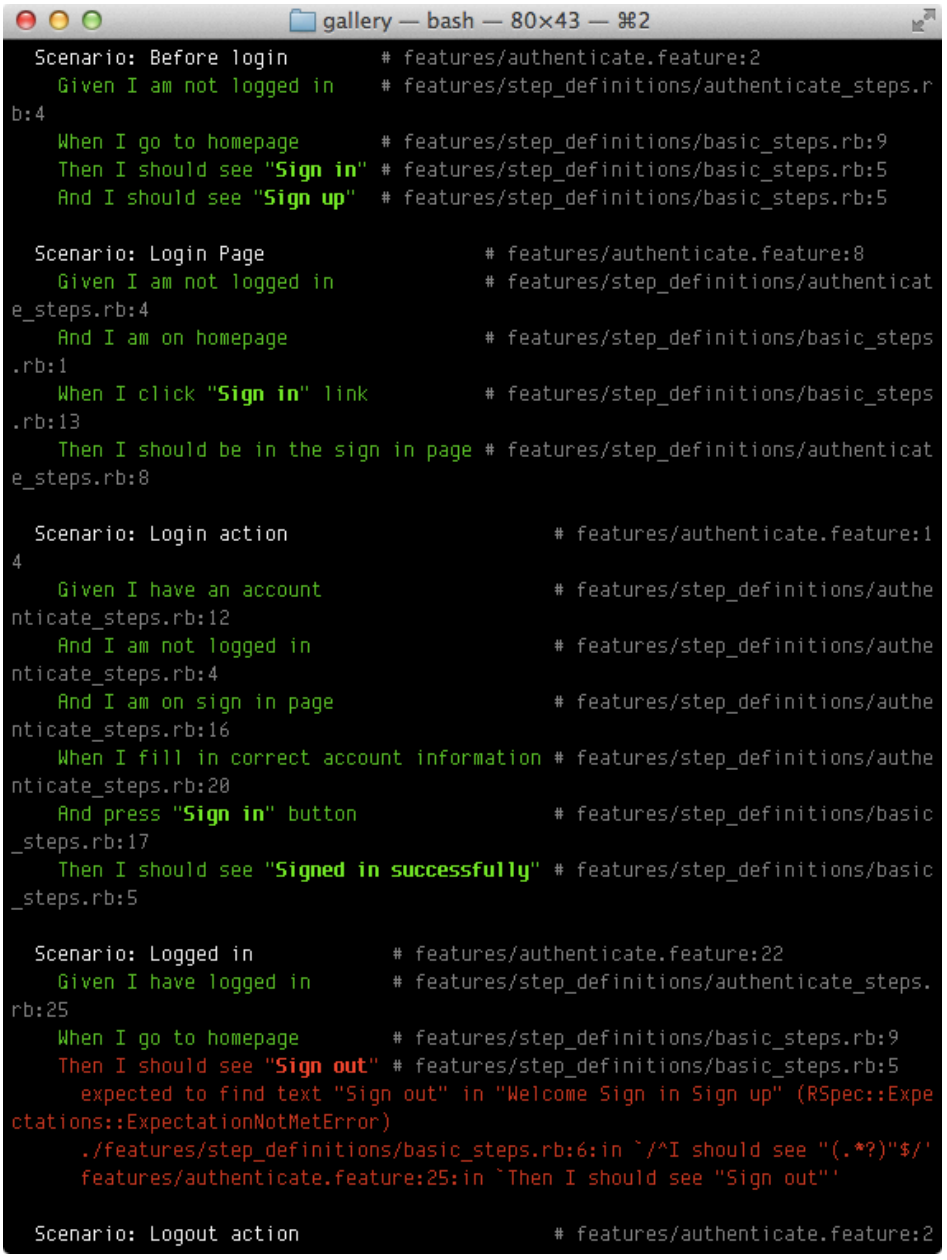
Next, the error is `expected to find text "Sign in"`. We can fix that by adding the "Sign in" link to layout file `application.html.erb`.

```
<%= link_to 'Sign in', new_user_session_path %>
```

And alon the sign up page.

```
<%= link_to 'Sign up', new_user_registration_path %>
```

The result of running `rake cucumber` again:



Thanks to the devise gem have done so much things, we passed several scenarios now. The next error is `expected to find text "Sign out"`.

We can tackle this error by adjusting the authentication links into:

```
<%- if user_signed_in? -%>
  <%= link_to 'Sign out', destroy_user_session_path, method: :delete %>
<%- else -%>
  <%= link_to 'Sign in', new_user_session_path %>
  <%= link_to 'Sign up', new_user_registration_path %>
<%- end -%>
```

Now we get all passes.

```
6 scenarios (6 passed)
24 steps (24 passed)
0m1.072s
```

Sometimes the scenarios failed at a confusing points that you don't expect. How about you can duml the HTML output of that step to deal with failing scenarios? `capybara-screenshot` gem is the one to use.

Add the capybara-screenshot gem to the :test group

```
gem 'capybara-screenshot'
```

And in the `features/support/env.rb` file, add the following line:

```
require 'capybara-screenshot/cucumber'
```

Now when fail scenario occurs, a dump of the target HTML will be saved:

```
Saved file /Users/makzan/Dropbox/share_to_air_mak/CPTTM/CM436-
RoR/repo/lesson7_examples/gallery/tmp/capybara/screenshot_2013-10-17-
18-13-24.911.html
```

The idea here is that you can freely change how the view look and the test cases act as a guard to protect the most basic functionality – user can login and logout.

Writing the photo upload component

How about uploading files?

We can create a feature to test it.

```
Feature: Photo Upload
  Scenario: Uploading link
    Given I am on homepage
    When I click "Upload photo" link
    Then I should be on the photo upload page

  Scenario: Uploading photo
    Given I am on photo upload page
    When I upload a valid photo with title "Test"
    Then I should see "Success"
    And I should see "Test"

  Scenario: Uploading invalid photo
    Given I am on photo upload page
    When I upload an invalid photo with title "Test"
    Then I should see "Error"
```

Again, lot's of errorn/pendings and it is absolutely normal.

The first error is the missing "Upload photo" link. Easy one, Let's create a link.

```
<%= link_to 'Upload photo', '#' %>
```

Yes, link to `'#'` now because we just want minimal code to make the test works.

Then we add the pending step one by one. The next one is the `I should be on the photo upload page`.

```
Then(/^I should be on the photo upload page$/) do
  current_path.should == '/photos/new'
end
```

Run the `rake cucumber` again and we get `expected: "/photos/new" got: "/"` error. That's good, it indicates that we can finally create the photo model and controllers. Don't write unnessesary code until the last minute.

In order to make the case passes, we need the photo resource and a photo upload page.

First, it is about the routes.

```
Gallery::Application.routes.draw do
devise_for :users

resources :photos

root to:'pages#index'
```

Next, the model.

```
$ rails generate model photo title:string
$ bundle install
$ rails generate paperclip photo image
$ rake db:migrate
```

Then make the photo.rb file match the following.

```
class Photo < ActiveRecord::Base
  attr_accessible :title, :image

  has_attached_file :image, :styles => { :medium => "300x300>",
:thumb => "100x100>" }, :default_url => "/images/:style/missing.png"

end
```

And the controller with `new` action.

```
$ rails generate controller photos new
```

The controller file.

```
class PhotosController < ApplicationController
  def new
    @photo = Photo.new
  end
end
```

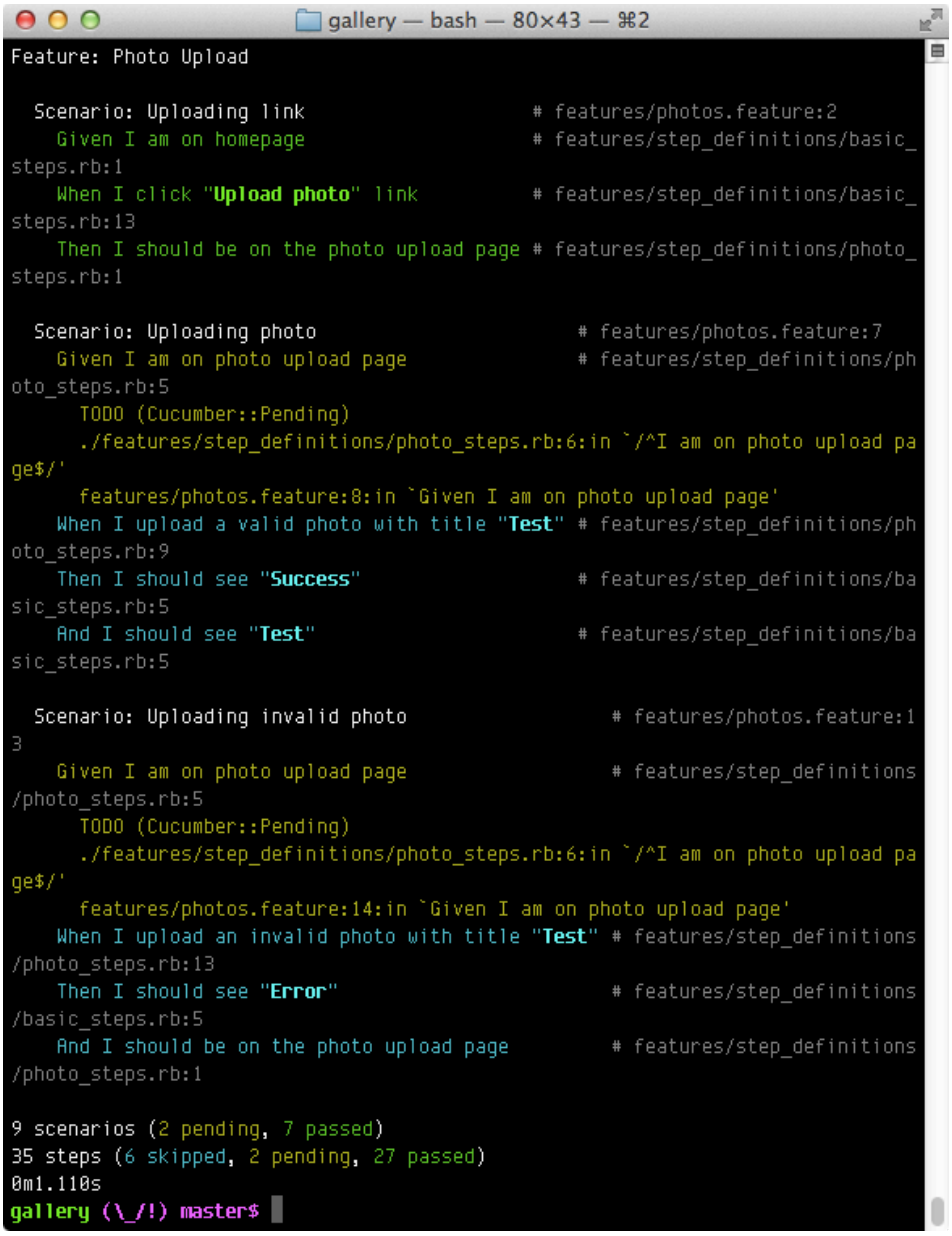
And the related view.

```
<%= form_for @photo, html: { multipart: true } do |f| %>
  <p>
    <%= f.label :title %>
    <%= f.text_field :title %>
  </p>

  <p>
    <%= f.label :image %>
    <%= f.file_field :image %>
  </p>

  <p><%= f.submit 'Upload Photo' %></p>
<% end %>
```

Nice, now we should now passes all existing cases with some pendings steps.



The next pending is `I am on photo upload page` which should be just a visit method.

```
Given(/^I am on photo upload page$/) do
  visit '/photos/new'
end
```

Then it is the `I upload a valid photo` step, which requires us to prepare a dummy PNG file and place it in the `features/upload_files` folder. (You can use any folder indeed)

But how to write the step for file uploading? Here it is.

```
When(/^I upload a valid photo with title "(.*?)"$/) do |title|
  fill_in 'Title', with: title
  attach_file :image, File.join(Rails.root, 'features',
'upload_files', 'ok.png')
  click_button "Upload Photo"
end
```

Now we got the `The action 'create' could not be found for PhotosController`

error. Well, we haven't handle the form POST yet.

And here we define the `create` method in photos_controller file.

```
def create
  @photo = Photo.new params[:photo]
  if @photo.save
    redirect_to @photo, notice: "Success"
  else
    flash[:alert] = "Error uploading photo."
    render :new
  end
end
```

This time cucumber generates another great failing: `The action 'show' could not be found for PhotosController`. It is like a virtual mentor telling us what to do next – the show method.

In the photos controller.

```
def show
  @photo = Photo.find params[:id]
end
```

The view `views/photos/show.html.erb`.

```
<%= @photo.title %>

<%= @photo.image.url %>
```

Great! The photo upload works and passes. Next one we will try to upload a non-image file.

```
When(/^I upload an invalid photo with title "(.*?)"$/) do |title|
  fill_in 'Title', with: title
  attach_file 'Image', File.join(Rails.root, 'features',
'upload_files', 'bad.txt')
  click_button "Upload Photo"
end
```

The cucumber shows an error that the txt file is successfully uploaded. That's because we haven't added any file format validation to the model yet. We can do that by adding the following code to the Photo model class, `photo.rb` file.

```
validates_attachment :image, presence: true, content_type:
["image/jpg", "image/png"]
```

Wonderful, all tests passed now.

```
9 scenarios (9 passed)
34 steps (34 passed)
0m1.883s
```

What's Next

You can't write the test until you know what you can do with it. So it is worth checking the Capybara API (https://github.com/jnicklas/capybara).

And if you search cucumber tutorials on internet, you may encounter something called "training wheel" which was deprecated and removed in cucumber now. For detail on why it is deprecated and how we can write better feature, check this post from Thoughtbot about writing better cucumber scenarios.