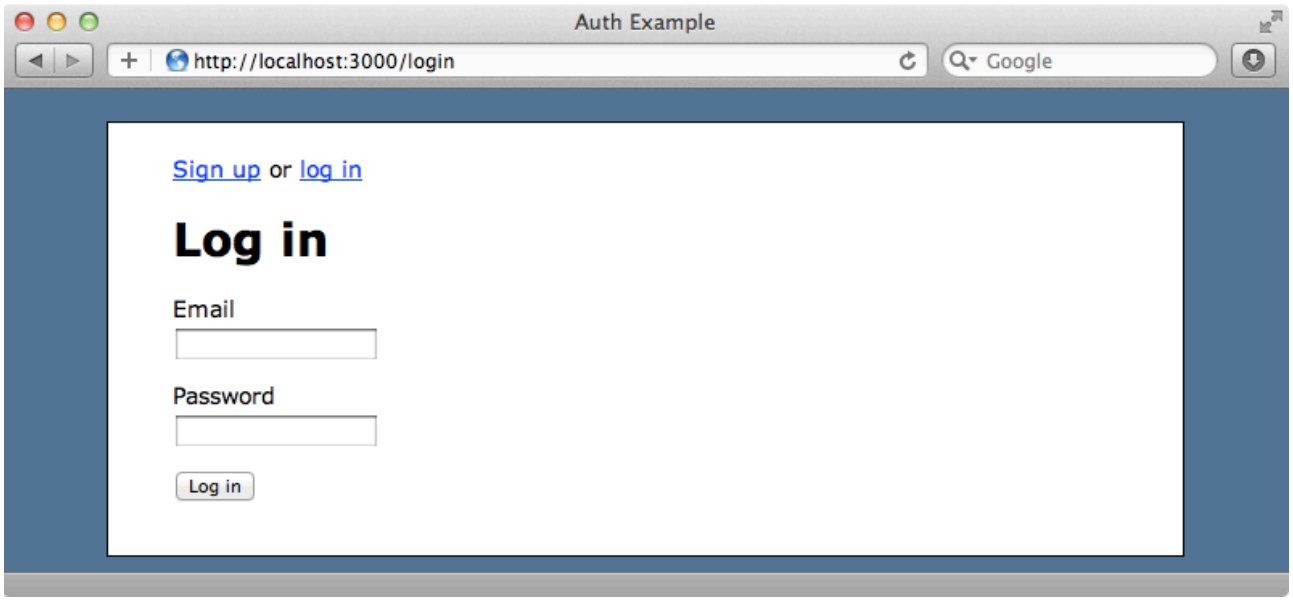
Episode 275

How I Test

From this episode onwards we're going to cover the subject of testing more frequently. This time we're going to show you how we would have written tests for the last episode [watch[1], read[2]], specifically the "forgotten password" link that we added to the login form.

When we started the last episode we had an application with a login form. The form had some basic authentication and a log in for but not the "remember me" checkbox or "forgotten password" link that we added as we went along. We'll add the link again this time but use Test Driven Development (TDD) to do it.



Last time we tested the application in the browser as we wrote it. This time we'll keep the browser closed and write code to test the functionality, only opening the browser when we need to focus on the user experience.

To help us write our tests we'll need to add some test-related gems to our application's `Gemfile`. We're using Rails 3.1, but everything we'll do here should work just as well in Rails 3.0. We'll add the gems at the bottom of the file in the `test` group.

---

[1] http://railscasts.com/episodes/274-remember-me-reset-password

[2] http://asciicasts.com/episodes/274-remember-me-reset-password

/Gemfile

```
source 'http://rubygems.org'

gem 'rails', '3.1.0.rc4'

gem 'sqlite3'

# Asset template engines
gem 'sass-rails', "~> 3.1.0.rc"
gem 'coffee-script'
gem 'uglifier'

gem 'jquery-rails'

gem "rspec-rails", :group => [:test, :development]
group :test do
  gem "factory_girl_rails"
  gem "capybara"
  gem "guard-rspec"
end
```

We're using RSpec[3] here, but any testing framework will do. Note that, unlike the other test-related gems, RSpec is in the `development` group as well so that the Rake tasks will run properly. We've also chosen Factory Girl[4] over fixtures, Capybara[5] for simulating user interaction with a web browser and Guard[6] for running tests automatically. Each of these gems has been covered in an earlier episode: Factory Girl was covered back in Episode 158[watch[7], read[8]], Capybara in Episode 257 [watch[9], read[10]] and Guard in episode 264 [watch[11], read[12]].

---

[3] http://relishapp.com/rspec

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

[5] https://github.com/jnicklas/capybara

[6] https://github.com/guard/guard

[7] http://railscasts.com/episodes/158-factories-not-fixtures

[8] http://asciicasts.com/episodes/158-factories-not-fixtures

[9] http://railscasts.com/episodes/257-request-specs-and-capybara

[10] http://asciicasts.com/episodes/257-request-specs-and-capybara

[11] http://railscasts.com/episodes/264-guard

[12] http://asciicasts.com/episodes/264-guard

We can install the gems by running `bundle`. Once they've installed we'll set up RSpec by running

```
$ rails g rspec:install
```

We'll create some directories under the `/spec` directory now: a `support` directory for support files, a `models` directory and a `routing` directory which is required by Guard.

```
$ mkdir spec/support spec/models spec/routing
```

Now is also a good time to run Guard's initalizer.

```
$ guard init rspec
```

As we're developing on OS X we'll also want to install the `rb-fsevent` gem[13] so that Guard can detect file changes. Once it's installed we'll start up Guard in a new Terminal tab and keep it running in the background.

```
$ guard
Please install growl gem for Mac OS X notification support and add
it to your Gemfile
Guard is now watching at '/Users/eifion/auth'
Guard::RSpec is running, with RSpec 2!
Running all specs
No examples found.
```

When we ran the RSpec generator it created a file at `/spec/spec_helper.rb`. We'll need to enable Capybara in this file and we can do so by adding `require 'capybara/rspec'` in there.

[13] https://rubygems.org/gems/rb-fsevent

/spec/spec_helper.rb

```
# This file is copied to spec/ when you run 'rails generate
rspec:install'
ENV["RAILS_ENV"] ||= 'test'
require File.expand_path("../../config/environment", __FILE__)
require 'rspec/rails'
require 'capybara/rspec'
# rest of file...
```

We'll also follow the advice in the file's comments and remove the fixture path line as we're not using fixtures any more.

/spec/spec_helper.rb

```
# Remove this line if you're not using ActiveRecord or ↵
  ActiveRecord fixtures
config.fixture_path = "#{::Rails.root}/spec/fixtures"
```

Our First Test

We're ready now to start testing and we'll start with an integration test that we'll call `password_reset`.

```
$ rails g integration_test password_reset
```

The RSpec generator will take over here and create what it calls a request spec. The default code in the spec looks like this:

/spec/requests/password_resets_spec.rb

```
require 'spec_helper'

describe "PasswordResets" do
  describe "GET /password_resets" do
    it "works! (now write some real specs)" do
      # Run the generator again with the --webrat flag if you want
to use webrat methods/matchers
      get password_resets_path
      response.status.should be(200)
    end
  end
end
```

We're going to remove the default spec and replace it with our own. We'll test that a user is sent an email when they request to have their password reset. To do this we'll need a `User` record to work with. We could go through the signup page and register a user that way but it's better to focus on exactly what we want to test and so we'll create a user from a factory. We'll create a `User` factory before we start on our test, putting it in a `factories.rb` file in the `/spec` directory. This name and location mean that any factories we define in here are automatically picked up by Factory Girl.

`/spec/factories.rb`

```
Factory.define :user do |f|
  f.sequence(:email) { |n| "foo#{n}@example.com" }
  f.password "secret"
end
```

This factory is simple and will generate a user with a unique email address and a password. We'll use it now in our test.

`/spec/requests/password_resets_spec.rb`

```
require 'spec_helper'

describe "PasswordResets" do
  it "emails user when requesting password reset"
    user = Factory(:user)
    visit login_path
    click_link "password"
    fill_in "Email", :with => user.email
    click_button "Reset Password"
  end
end
```

This test uses our factory to create a user then simulates the steps the user would take to reset their password with several Capybara commands. We visit the login page and then click a link containing the word “password”. By not defining the link’s text exactly our tests are less brittle so if the text is changed from, say, “Forgotten password” to “Forgot your password?” it will still pass. On the page that the link takes us to we find a text field with an associated label whose text contains “Email” and fill it with the user’s email address. Finally we’ll click the “Reset Password” button.

Our spec isn’t finished yet but when we save it Guard will run it and we’ll see our first failure.

```
  1) PasswordResets emails user when requesting password reset
     Failure/Error: click_link "password"
     Capybara::ElementNotFound:
       no link with title, id or text 'password' found
     # (eval):2:in `click_link'
     # ./spec/requests/password_resets_spec.rb:7:in `block (2
levels) in <top (required)>'
```

The spec failed as Capybara couldn’t find the “password” link. We’ll fix that before we continue. All we need to do is add the link on the login page.

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

```
<h1>Log in</h1>

<%= form_tag sessions_path do %>
  <div class="field">
    <%= label_tag :email %>
    <%= text_field_tag :email, params[:email] %>
  </div>
  <div class="field">
    <%= label_tag :password %>
    <%= password_field_tag :password %>
  </div>
  <p><%= link_to "forgotten password?", new_password_reset_path %>
  <div class="actions"><%= submit_tag "Log in" %></div>
<% end %>
```

The link goes to `new_password_reset_path`, but as we don't have that path defined yet Guard will give us another error when it runs the spec again.

```
  1) PasswordResets emails user when requesting password reset
     Failure/Error: visit login_path
     ActionView::Template::Error:
       undefined local variable or method
`new_password_reset_path' for #<#<Class:0x000001039349d8>:
0x000001039269f0>
```

This shows the advantage to this approach to testing. It always shows you the next error and it should only require a small code change to fix. To fix this issue we'll generate a `PasswordResets` controller with a `new` action. As we're using request specs to test the controller and view layers we don't need the controller and view spec files. We can tell the generator to not create them by passing in the `--no-test-framework` option.

```
$ rails g controller password_resets new --no-test-framework
```

We'll also need to modify the routes file to make `PasswordResets` a resource.

`/config/routes.rb`

```
Auth::Application.routes.draw do

  get "logout" => "sessions#destroy", :as => "logout"
  get "login" => "sessions#new", :as => "login"
  get "signup" => "users#new", :as => "signup"
  root :to => "home#index"
  resources :users
  resources :sessions
  resources :password_resets
end
```

When Guard runs now it tells us that it cannot find the email text field on the reset password page.

```
  1) PasswordResets emails user when requesting password reset
     Failure/Error: fill_in "Email", :with => user.email
     Capybara::ElementNotFound:
       cannot fill in, no text field, text area or password field
with id, name, or label 'Email' found
```

To fix this we'll replace the default code in the password reset view with a form with the appropriate text field and button.

/app/views/password_resets/new.html.erb

```
<%= form_tag password_resets_path, :method => :post do %>
  <div class="field">
    <%= label_tag :email %>
    <%= text_field_tag :email, params[:email] %>
  </div>
  <div class="actions"><%= submit_tag "Reset Password" %></div>
<% end %>
```

The next error we see is caused by there being no `create` action for the form to POST to.

```
  1) PasswordResets emails user when requesting password reset
     Failure/Error: click_button "Reset Password"
     AbstractController::ActionNotFound:
       The action 'create' could not be found for ↵
       PasswordResetsController
```

We'll create this action in the controller and have it redirect to the home page.

/app/controllers/password_resets_controller.rb

```
class PasswordResetsController < ApplicationController
  def new
  end

  def create
    redirect_to :root
  end

end
```

We've done enough now to get out spec passing.

```
Running: spec/controllers/password_resets_controller_spec.rb
.

Finished in 0.14507 seconds
1 example, 0 failures
```

Expanding Our Spec

Now that our spec passes we can expand it. We want to show a flash message after the "Reset Password" button is sent so we'll add that to the spec.

/spec/requests/password_resets_spec.rb

```
require 'spec_helper'

describe "PasswordResets" do
  it "emails user when requesting password reset" do
    user = Factory(:user)
    visit login_path
    click_link "password"
    fill_in "Email", :with => user.email
    click_button "Reset Password"
    page.should have_content("Email sent")
  end
end
```

Of course this fails as we haven't written the code to show the message. We'll modify the controller so that it does.

/app/controllers/password_resets.rb

```
class PasswordResetsController < ApplicationController
  def new
  end

  def create
    redirect_to :root, :notice => "Email sent with password reset
instructions."
  end

end
```

The spec now passes again but we aren't actually sending the email. We can test for this by using `ActionMailer::Base::deliveries` to get at a list of the delivered emails. We can then call `last` on that list to get the last delivered email. This is something we'll be doing quite a bit in our specs so we'll create a new file in the `/spec/support` directory and write some code in it to fetch the last email.

/spec/support/mailer_macros.rb

```
module MailerMacros
  def last_email
    ActionMailer::Base.deliveries.last
  end

  def reset_email
    ActionMailer::Base.deliveries = []
  end
end
```

We've also written a `reset_email` method that we'll call at the beginning of each spec. This will empty the list so that we can start each spec from a known state.

So that we can use these new methods in our specs we'll call `config.include` in our `spec_helper` file and include our new module.

/spec/spec_helper.rb

```
# This file is copied to spec/ when you run 'rails generate
rspec:install'
ENV["RAILS_ENV"] ||= 'test'
require File.expand_path("../../config/environment", __FILE__)
require 'rspec/rails'
require 'capybara/rspec'

# Requires supporting ruby files with custom matchers and macros,
etc,
# in spec/support/ and its subdirectories.
Dir[Rails.root.join("spec/support/**/*.rb")].each {|f| require f}

RSpec.configure do |config|
  config.mock_with :rspec
  config.use_transactional_fixtures = true
  config.include(MailerMacros)
  config.before(:each) { reset_email }
end
```

After we've included our macro we'll use `config.before(:each)` to call `reset_email` so that the delivered emails list is emptied before each spec is run.

We can use our new `last_email` method now in our spec, and check that the last email sent was sent to the user we created there.

/spec/requests/password_resets_spec.rb

```
require 'spec_helper'

describe "PasswordResets" do
  it "emails user when requesting password reset" do
    user = Factory(:user)
    visit login_path
    click_link "password"
    fill_in "Email", :with => user.email
    click_button "Reset Password"
    page.should have_content("Email sent")
    last_email.to.should include(user.email)
  end
end
```

This will of course fail. The `last_email` will be `nil` as we haven't written any code yet to send it. We'll generate a mailer so that we can do so.

```
$ rails g mailer user_mailer password_reset
```

The generator creates its own spec file and we'll take a look a this a little later. For now we'll comment it out so that we can concentrate on our current spec. We'll modify the mailer so that it sends email to the user's email address and set an appropriate subject.

/app/mailers/user_mailer.rb

```
class UserMailer < ActionMailer::Base
  default from: "from@example.com"

  def password_reset(user)
    @user = user
    mail :to => user.email, :subject => "Password Reset"
  end
end
```

In our `PasswordResetsController` we'll alter the `create` action so that it finds a `User` by the email address that was entered on the form and sends the email.

/app/controllers/password_resets_controller.rb

```
def create
  user = User.find_by_email(params[:email])
  UserMailer.password_reset(user).deliver
  redirect_to :root, :notice => "Email sent with password ↵
   reset instructions."
end
```

Our spec now passes again.

Handling the Password Reset Token

Even though the spec is passing its functionality is far from complete. We should be generating a password reset token and including that in the email. These details don't need to be inside the request spec; it's better to keep it simple and have it define the general flow of the request, in this case checking that the user receives a password reset email when they request one. We can use lower-level tests to test the details.

Now that we have a passing spec it's a good time to take a look at our code to see what can be moved from the controller into the model. A good example here is the line of code in the `PasswordResetsController` that sends the email. We can move this into a new `send_password_reset` method in the `User` model.

/app/controllers/password_resets_controller.rb

```
def create
  user = User.find_by_email(params[:email])
  user.send_password_reset
  redirect_to :root, :notice => "Email sent with password ↵
   reset instructions."
end
```

/app/models/user.rb

```
class User < ActiveRecord::Base
  attr_accessible :email, :password, :password_confirmation
  has_secure_password
  validates_presence_of :password, :on => :create

  def send_password_reset
    UserMailer.password_reset(self).deliver
  end
end
```

At this point we'll check that the specs still pass. They do so we can continue. Next we'll add some more specs to flesh out the `User` model. We'll create a spec file at `/spec/models/user.rb` and add some specs to it.

/spec/models/user_spec.rb

```
require 'spec_helper'

describe User do
  describe "#send_password_reset" do
    let(:user) { Factory(:user) }

    it "generates a unique password_reset_token each time" do
      user.send_password_reset
      last_token = user.password_reset_token
      user.send_password_reset
      user.password_reset_token.should_not eq(last_token)
    end

    it "saves the time the password reset was sent" do
      user.send_password_reset
      user.reload.password_reset_sent_at.should be_present
    end

    it "delivers email to user" do
      user.send_password_reset
      last_email.to.should include (user.email)
    end
  end
end
```

We want the `send_password_reset` method to do three things when it's called. It should create a unique password reset token, save the time that the token was sent and deliver an email to the user. It already does the last of these; we'll modify the method now so that it does the other two. Note that before the specs we call `let(:user)`. This assigns `user` to a new user from the factory before each spec is run.

Two of the specs are currently failing and this is because we don't have either the `password_reset_token` or `password_reset_sent_at` fields in our `users` table yet. We can fix this by running the following migration and the migrating the database.

```
$ rails g migration add_password_reset_to_users ↵
password_reset_token:string password_reset_sent_at:datetime
```

With the new database fields in place the specs still fail but for different reasons.

```
Failures:

  1) User#send_password_reset generates a unique
password_reset_token each time
     Failure/Error: user.password_reset_token.should_not ↵
       eq(last_token)

       expected nil not to equal nil

       (compared using ==)
     # ./spec/models/user_spec.rb:11:in `block (3 levels) in ↵
       <top (required)>'

  2) User#send_password_reset saves the time the password ↵
     reset was sent
     Failure/Error: user.reload.password_reset_sent_at.should ↵
       be_present
       expected present? to return true, got false
     # ./spec/models/user_spec.rb:16:in `block (3 levels) in ↵
       <top (required)>'
```

The specs now fail because the `password_reset_token` and `password_reset_sent_at` fields aren't being set in the `sent_password_reset`

method. This can be fixed by writing a `generate_token` method that creates a unique token. We can then modify `sent_password_reset` so that it calls `generate_token` , sets the `password_reset_sent_at` time and saves the user.

/app/models/user.rb

```
class User < ActiveRecord::Base
  attr_accessible :email, :password, :password_confirmation
  has_secure_password
  validates_presence_of :password, :on => :create

  def send_password_reset
    generate_token(:password_reset_token)
    self.password_reset_sent_at = Time.zone.now
    save!
    UserMailer.password_reset(self).deliver
  end

  def generate_token(column)
    begin
      self[column] = SecureRandom.urlsafe_base64
    end while User.exists?(column => self[column])
  end
end
```

All our specs now pass again.

Testing The Mailer

Now that our specs pass we'll go back to the mailer spec that was created when we generated the mailer and which we commented out. We'll need to modify the default code so that we can test that our mailer works correctly. In the spec we'll create a new user from the factory but this time we'll set a `password_reset_token` for the user. We'll then alter the line that creates the mail so that the user is passed in to the call to `UserMailer.password_reset`.

The spec will check that the email is sent to the correct email address and that the body contains the correct link to the user's password reset token.

/spec/mailers/user_mailer_spec.rb

```
require "spec_helper"

describe UserMailer do
  describe "password_reset" do
    let(:user) { Factory(:user, :password_reset_token =>
"anything") }
    let(:mail) { UserMailer.password_reset(user) }

    it "sends user password reset url" do
      mail.subject.should eq("Password Reset")
      mail.to.should eq([user.email])
      mail.from.should eq(["from@example.com"])
    end

    it "renders the body" do
      mail.body.encoded.should
match(edit_password_reset_path(user.password_reset_token))
    end
  end
end
```

Our spec fails as the email's body doesn't contain the correct link. Let's add that in.

/app/views/user_mailer/password_reset.text.erb

```
To reset your password, click the URL below.

<%= edit_password_reset_url(@user.password_reset_token) %>

If you did not request your password to be reset just ignore this
email and your password will continue to stay the same.
```

The specs still fail as there's a missing `:host` option for sending the email. We can set this in our test environment's config file by adding the following line.

/config/environments/test.rb

```
config.action_mailer.default_url_options = { :host => ↵
  "www.example.com" }
```

We'll need to set this value in our development and production environments too but we won't do that now.

All of our specs now pass again. By the way, if we ever need to tell Guard to rerun the specs manually we can do so with CTRL+\.

Testing Other Scenarios

One of the hardest parts of test driven development is getting started and establishing a workflow. Once you've got going it's easy to copy and past tests to add variations and test functionality. For example let's test the case where a user enters an invalid email address and requests a password reset. We can easily copy the existing spec in password_resets_spec.rb and create a new one to test this.

/spec/requests/password_resets_spec.rb

```
it "does not email invalid user when requesting password reset" do
  visit login_path
  click_link "password"
  fill_in "Email", :with => "madeupuser@example.com"
  click_button "Reset Password"
  page.should have_content("Email sent")
  last_email.should be_nil
end
```

The spec fails as the controller's code fails if a matching user isn't found. We'll fix this now.

/app/controllers/password_resets_controller.rb

```
def create
  user = User.find_by_email(params[:email])
  user.send_password_reset if user?
  redirect_to :root, :notice => "Email sent with password ↵
    reset instructions."
end
```

This satisfies the test case and now all of our specs pass again.

With this testing pattern established it's easy to go into this request spec and add additional functionality for resetting a password, for example to test that the reset password token hasn't expired or test cases when the token passed in is invalid and

so on. More test cases are available in the final source code for this episode on Ryan Bates' Github site[14].

That's it for this episode on testing the "forgotten password" link. Testing can be a controversial subject and people have different views on the best way to write tests for Rails applications. What's most important is that you do test your applications whichever method you choose.

---

[14] https://github.com/ryanb/railscasts-episodes