11.3 Activating the account

Now that we have a correctly generated email as in Listing 11.25, we need to write the **edit** action in the Account Activations controller that actually activates the user. As usual, we'll write a test for this action, and once the code is tested we'll refactor it to move some functionality out of the Account Activations controller and into the User model.

11.3.1 Generalizing the authenticated? method

Recall from the discussion in Section 11.2.1 that the activation token and email are available as **params[:id]** and **params[:email]**, respectively. Following the model of passwords (Listing 8.7) and remember tokens (Listing 9.9), we plan to find and authenticate the user with code something like this:

```
user = User.find_by(email: params[:email])
if user && user.authenticated?(:activation, params[:id])
```

(As we'll see in a moment, there will be one extra boolean in the expression above. See if you can guess what it will be.)

The above code uses the **authenticated?** method to test if the account activation digest matches the given token, but at present this won't work because that method is specialized to the remember token (Listing 9.6):

```
# Returns true if the given token matches the digest.
def authenticated?(remember_token)
  return false if remember_digest.nil?
  BCrypt::Password.new(remember_digest).is_password?(remember_token)
end
```

Here **remember_digest** is an attribute on the User model, and inside the model we can rewrite it as follows:

```
self.remember_digest
```

Somehow, we want to be able to make this *variable*, so we can call

```
self.activation_digest
```

instead by passing in the appropriate parameter to the **authenticated?** method.

The solution involves our first example of *metaprogramming*, which is essentially a program that writes a program. (Metaprogramming is one of Ruby's strongest suits, and many of the "magic" features of Rails are due to its use of Ruby metaprogramming.) The key in this case is the powerful **send** method, which lets us call a method with a name of our choice by "sending a message" to a given object. For example, in this console session we use **send** on a native Ruby object to find the length of an array:

```
$ rails console
>> a = [1, 2, 3]
>> a.length
=> 3
>> a.send(:length)
=> 3
>> a.send("length")
=> 3
```

Here we see that passing the symbol **:length** or string **"length"** to **send** is equivalent to calling the **length** method on the given object. As a second example, we'll access the **activation_digest** attribute of the first user in the database:

```
>> user = User.first
>> user.activation_digest
=> "$2a$10$4e6TFzEJAVNyjLv8Q5u22ensMt28qEkx0roaZvtRcp6UZKRM6N9Ae"
>> user.send(:activation_digest)
=> "$2a$10$4e6TFzEJAVNyjLv8Q5u22ensMt28qEkx0roaZvtRcp6UZKRM6N9Ae"
>> user.send("activation_digest")
```

```
=> "$2a$10$4e6TFzEJAVNyjLv8Q5u22ensMt28qEkx0roaZvtRcp6UZKRM6N9Ae"
>> attribute = :activation
>> user.send("#{attribute}_digest")
=> "$2a$10$4e6TFzEJAVNyjLv8Q5u22ensMt28qEkx0roaZvtRcp6UZKRM6N9Ae"
```

Note in the last example that we've defined an **`attribute`** variable equal to the symbol **`:activation`** and used string interpolation to build up the proper argument to **`send`**. This would work also with the string **`'activation'`**, but using a symbol is more conventional, and in either case

```
"#{attribute}_digest"
```

becomes

```
"activation_digest"
```

once the string is interpolated. (We saw how symbols are interpolated as strings in Section 7.4.2.)

Based on this discussion of **`send`**, we can rewrite the current **`authenticated?`** method as follows:

```
def authenticated?(remember_token)
  digest = self.send("remember_digest")
  return false if digest.nil?
  BCrypt::Password.new(digest).is_password?(remember_token)
end
```

With this template in place, we can generalize the method by adding a function argument with the name of the digest, and then use string interpolation as above:

```
def authenticated?(attribute, token)
  digest = self.send("#{attribute}_digest")
  return false if digest.nil?
  BCrypt::Password.new(digest).is_password?(token)
end
```

(Here we have renamed the second argument **token** to emphasize that it's now generic.) Because we're inside the user model, we can also omit **self**, yielding the most idiomatically correct version:

```
def authenticated?(attribute, token)
  digest = send("#{attribute}_digest")
  return false if digest.nil?
  BCrypt::Password.new(digest).is_password?(token)
end
```

We can now reproduce the previous behavior of **authenticated?** by invoking it like this:

```
user.authenticated?(:remember, remember_token)
```

Applying this discussion to the User model yields the generalized **authenticated?** method shown in Listing 11.26.

Listing 11.26: A generalized **authenticated?** method. RED
app/models/user.rb

```
class User < ApplicationRecord
  .
  .
  .
  # Returns true if the given token matches the digest.
  def authenticated?(attribute, token)
    digest = send("#{attribute}_digest")
    return false if digest.nil?
    BCrypt::Password.new(digest).is_password?(token)
  end
  .
  .
  .
end
```

The caption to Listing 11.26 indicates a RED test suite:

Listing 11.27: RED

```
$ rails test
```

The reason for the failure is that the **`current_user`** method (Listing 9.9) and the test for **`nil`** digests (Listing 9.17) both use the old version of **`authenticated?`**, which expects one argument instead of two. Note that this is exactly the kind of error a test suite is supposed to catch.

To fix the issue, we simply update the two cases to use the generalized method, as shown in Listing 11.28 and Listing 11.29.

Listing 11.28: Using the generalized **`authenticated?`** method in **`current_user`**. RED
app/helpers/sessions_helper.rb

```
module SessionsHelper
  .
  .
  .
  # Returns the current logged-in user (if any).
  def current_user
    if (user_id = session[:user_id])
      @current_user ||= User.find_by(id: user_id)
    elsif (user_id = cookies.signed[:user_id])
      user = User.find_by(id: user_id)
      if user && user.authenticated?(:remember, cookies[:remember_token])
        log_in user
        @current_user = user
      end
    end
  end
  .
  .
  .
end
```

Listing 11.29: Using the generalized **`authenticated?`** method in the User test. GREEN
test/models/user_test.rb

```ruby
require 'test_helper'

class UserTest < ActiveSupport::TestCase

  def setup
    @user = User.new(name: "Example User", email: "user@example.com",
                     password: "foobar", password_confirmation: "foobar")
  end
  .
  .
  .
  test "authenticated? should return false for a user with nil digest" do
    assert_not @user.authenticated?(:remember, '')
  end
end
```

At this point, the tests should be GREEN:

Listing 11.30: GREEN

```
$ rails test
```

Refactoring the code as above is incredibly more error-prone without a solid test suite, which is why we went to such trouble to write good tests in Section 9.1.2 and Section 9.3.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Create and remember new user at the console. What are the user's remember and activation tokens? What are the corresponding digests?

2. Using the generalized `authenticated?` method from Listing 11.26, verify that the user is authenticated according to both the remember token and the activation token.

11.3.2 Activation **edit** action

With the **authenticated?** method as in Listing 11.26, we're now ready to write an **edit** action that authenticates the user corresponding to the email address in the **params** hash. Our test for validity will look like this:

```
if user && !user.activated? && user.authenticated?(:activation, params[:id])
```

Note the presence of **!user.activated?**, which is the extra boolean alluded to above. This prevents our code from activating users who have already been activated, which is important because we'll be logging in users upon confirmation, and we don't want to allow attackers who manage to obtain the activation link to log in as the user.

If the user is authenticated according to the booleans above, we need to activate the user and update the **activated_at** timestamp:[8]

```
user.update_attribute(:activated,    true)
user.update_attribute(:activated_at, Time.zone.now)
```

This leads to the **edit** action shown in Listing 11.31. Note also that Listing 11.31 handles the case of an invalid activation token; this should rarely happen, but it's easy enough to redirect in this case to the root URL.

Listing 11.31: An **edit** action to activate accounts.
app/controllers/account_activations_controller.rb

```
class AccountActivationsController < ApplicationController

  def edit
    user = User.find_by(email: params[:email])
    if user && !user.activated? && user.authenticated?(:activation, params[:id])
      user.update_attribute(:activated,    true)
      user.update_attribute(:activated_at, Time.zone.now)
```

[8]Here we use two calls to **update_attribute** rather than a single call to **update_attributes** because (per Section 6.1.5) the latter would run the validations. Lacking in this case the user password, these validations would fail.

```
      log_in user
      flash[:success] = "Account activated!"
      redirect_to user
    else
      flash[:danger] = "Invalid activation link"
      redirect_to root_url
    end
  end
end
```

With the code in Listing 11.31, you should now be able to paste in the URL from Listing 11.25 to activate the relevant user. For example, on my system I visited the URL

```
https://0ebe1dc6d40e4a4bb06e0ca7fe138127.vfs.cloud9.us-east-2.
amazonaws.com/account_activations/zdqs6sF7BMiDfXBaC7-6vA/
edit?email=michael%40michaelhartl.com
```

and got the result shown in Figure 11.6.

Of course, currently user activation doesn't actually *do* anything, because we haven't changed how users log in. In order to have account activation mean something, we need to allow users to log in only if they are activated. As shown in Listing 11.32, the way to do this is to log the user in as usual if `user.activated?` is true; otherwise, we redirect to the root URL with a `warning` message (Figure 11.7).

Listing 11.32: Preventing unactivated users from logging in.
app/controllers/sessions_controller.rb

```
class SessionsController < ApplicationController

  def new
  end

  def create
    user = User.find_by(email: params[:session][:email].downcase)
    if user && user.authenticate(params[:session][:password])
      if user.activated?
        log_in user
        params[:session][:remember_me] == '1' ? remember(user) : forget(user)
        redirect_back_or user
```

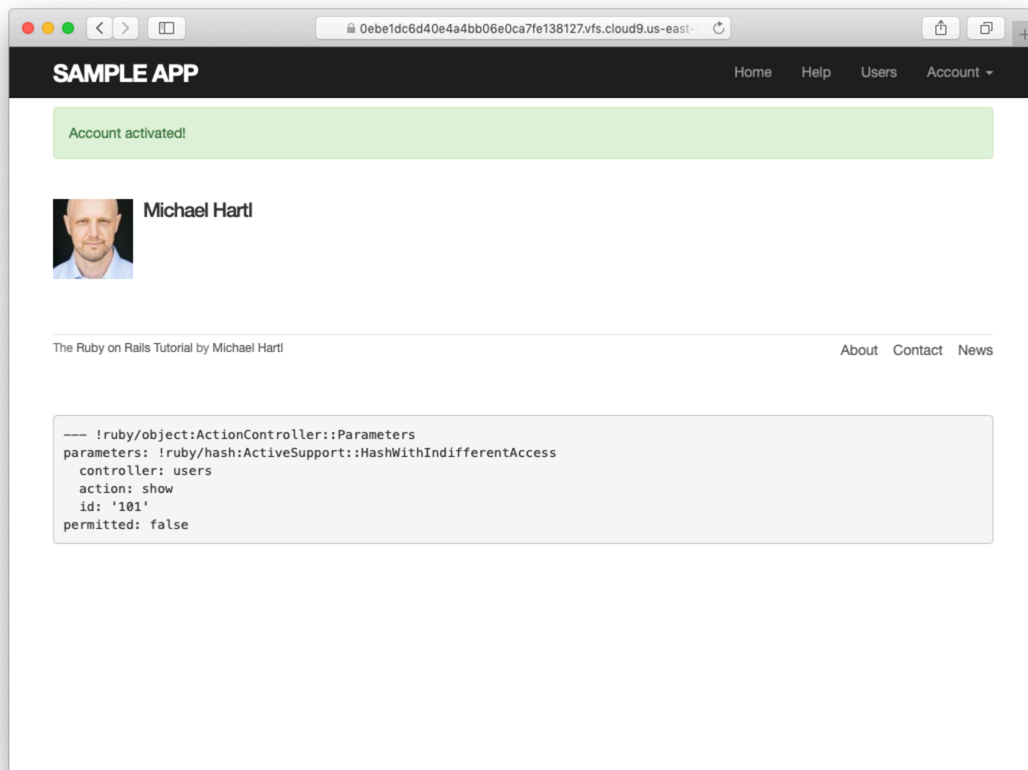

Figure 11.6: The profile page after a successful activation.

```
      else
        message  = "Account not activated. "
        message += "Check your email for the activation link."
        flash[:warning] = message
        redirect_to root_url
      end
    else
      flash.now[:danger] = 'Invalid email/password combination'
      render 'new'
    end
  end

  def destroy
    log_out if logged_in?
    redirect_to root_url
  end
end
```

With that, apart from one refinement, the basic functionality of user activation is done. (That refinement is preventing unactivated users from being displayed, which is left as an exercise (Section 11.3.3).) In Section 11.3.3, we'll complete the process by adding some tests and then doing a little refactoring.

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. Paste in the URL from the email generated in Section 11.2.4. What is the activation token?

2. Verify at the console that the User is authenticated according to the activation token in the URL from the previous exercise. Is the user now activated?

11.3.3 Activation test and refactoring

In this section, we'll add an integration test for account activation. Because we already have a test for signing up with valid information, we'll add the steps to

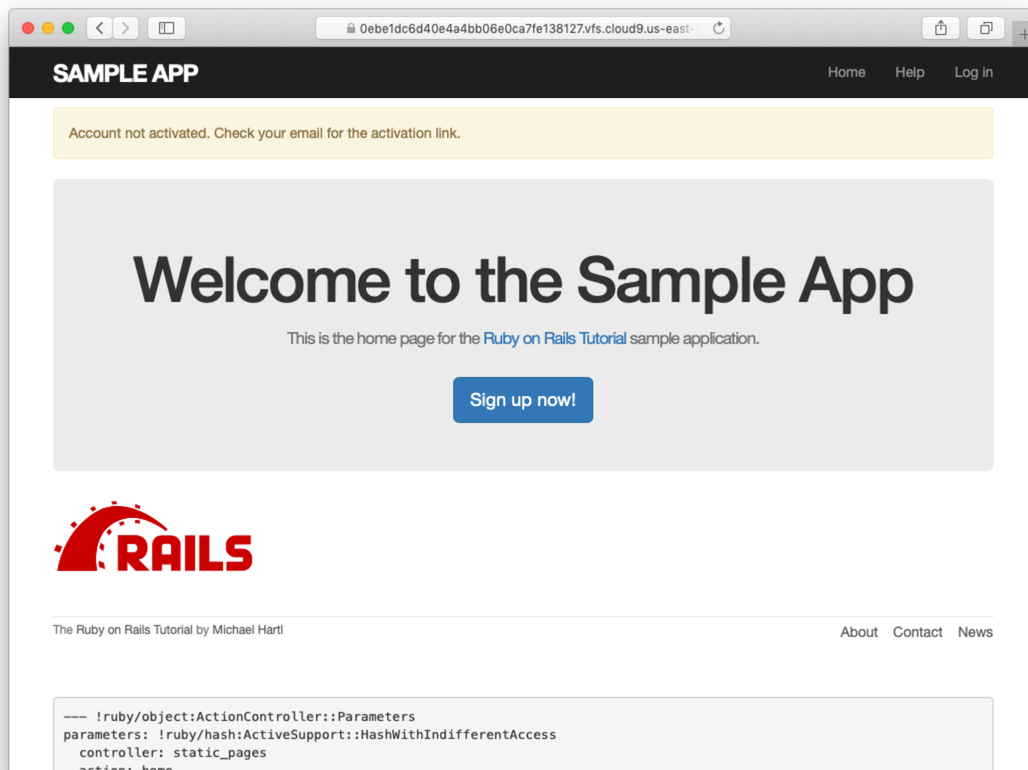

Figure 11.7: The warning message for a not-yet-activated user.

the test developed in Section 7.4.4 (Listing 7.31). There are quite a few steps, but they are mostly straightforward; see if you can follow along in Listing 11.33. (The highlights in Listing 11.33 indicate lines that are especially important or easy to miss, but there are other new lines as well, so take care to add them all.)

Listing 11.33: Adding account activation to the user signup test. GREEN
test/integration/users_signup_test.rb

```
require 'test_helper'

class UsersSignupTest < ActionDispatch::IntegrationTest

  def setup
    ActionMailer::Base.deliveries.clear
  end

  test "invalid signup information" do
    get signup_path
    assert_no_difference 'User.count' do
      post users_path, params: { user: { name:  "",
                                         email: "user@invalid",
                                         password:              "foo",
                                         password_confirmation: "bar" } }
    end
    assert_template 'users/new'
    assert_select 'div#error_explanation'
    assert_select 'div.field_with_errors'
  end

  test "valid signup information with account activation" do
    get signup_path
    assert_difference 'User.count', 1 do
      post users_path, params: { user: { name:  "Example User",
                                         email: "user@example.com",
                                         password:              "password",
                                         password_confirmation: "password" } }
    end
    assert_equal 1, ActionMailer::Base.deliveries.size
    user = assigns(:user)
    assert_not user.activated?
    # Try to log in before activation.
    log_in_as(user)
    assert_not is_logged_in?
    # Invalid activation token
    get edit_account_activation_path("invalid token", email: user.email)
    assert_not is_logged_in?
    # Valid token, wrong email
    get edit_account_activation_path(user.activation_token, email: 'wrong')
```

```
    assert_not is_logged_in?
    # Valid activation token
    get edit_account_activation_path(user.activation_token, email: user.email)
    assert user.reload.activated?
    follow_redirect!
    assert_template 'users/show'
    assert is_logged_in?
  end
end
```

There's a lot of code in Listing 11.33, but the only completely novel code is in the line

```
assert_equal 1, ActionMailer::Base.deliveries.size
```

This code verifies that exactly 1 message was delivered. Because the **deliveries** array is global, we have to reset it in the **setup** method to prevent our code from breaking if any other tests deliver email (as will be the case in Chapter 12).

Listing 11.33 also uses the **assigns** method for the first time in the main tutorial; as explained in a Chapter 9 exercise (Section 9.3.1), **assigns** lets us access instance variables in the corresponding action. For example, the Users controller's **create** action defines an **@user** variable (Listing 11.23), so we can access it in the test using **assigns(:user)**. The **assigns** method is deprecated in default Rails tests as of Rails 5, but I still find it useful in many contexts, and it's available via the `rails-controller-testing` gem we included in Listing 3.2.

Finally, note that Listing 11.33 restores the lines we commented out in Listing 11.24.

At this point, the test suite should be GREEN:

Listing 11.34: GREEN

```
$ rails test
```

With the test in Listing 11.33, we're ready to refactor a little by moving some of the user manipulation out of the controller and into the model. In particular,

we'll make an **activate** method to update the user's activation attributes and a **send_activation_email** to send the activation email. The extra methods appear in Listing 11.35, and the refactored application code appears in Listing 11.36 and Listing 11.37.

Listing 11.35: Adding user activation methods to the User model.
app/models/user.rb

```
class User < ApplicationRecord
  .
  .
  .
  # Activates an account.
  def activate
    update_attribute(:activated,    true)
    update_attribute(:activated_at, Time.zone.now)
  end

  # Sends activation email.
  def send_activation_email
    UserMailer.account_activation(self).deliver_now
  end

  private
    .
    .
    .
end
```

Listing 11.36: Sending email via the user model object.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def create
    @user = User.new(user_params)
    if @user.save
      @user.send_activation_email
      flash[:info] = "Please check your email to activate your account."
      redirect_to root_url
    else
      render 'new'
    end
```

```
  end
  .
  .
  .
end
```

Listing 11.37: Account activation via the user model object.
app/controllers/account_activations_controller.rb

```ruby
class AccountActivationsController < ApplicationController

  def edit
    user = User.find_by(email: params[:email])
    if user && !user.activated? && user.authenticated?(:activation, params[:id])
      user.activate
      log_in user
      flash[:success] = "Account activated!"
      redirect_to user
    else
      flash[:danger] = "Invalid activation link"
      redirect_to root_url
    end
  end
end
```

Note that Listing 11.35 eliminates the use of **`user.`**, which would break inside the User model because there is no such variable:

```
-user.update_attribute(:activated,    true)
-user.update_attribute(:activated_at, Time.zone.now)
+update_attribute(:activated,    true)
+update_attribute(:activated_at, Time.zone.now)
```

(We could have switched from **`user`** to **`self`**, but recall from Section 6.2.5 that **`self`** is optional inside the model.) It also changes **`@user`** to **`self`** in the call to the User mailer:

```
-UserMailer.account_activation(@user).deliver_now
+UserMailer.account_activation(self).deliver_now
```

These are *exactly* the kinds of details that are easy to miss during even a simple refactoring but will be caught by a good test suite. Speaking of which, the test suite should still be GREEN:

Listing 11.38: GREEN

```
$ rails test
```

Exercises

Solutions to the exercises are available to all Rails Tutorial purchasers here.

To see other people's answers and to record your own, subscribe to the Rails Tutorial course or to the Learn Enough All Access Bundle.

1. In Listing 11.35, the **`activate`** method makes two calls to the **`update_attribute`**, each of which requires a separate database transaction. By filling in the template shown in Listing 11.39, replace the two **`update_attribute`** calls with a single call to **`update_columns`**, which hits the database only once. (Note that, like **`update_attribute`**, **`update_columns`** doesn't run the model callbacks or validations.) After making the changes, verify that the test suite is still GREEN.

2. Right now *all* users are displayed on the user index page at /users and are visible via the URL /users/:id, but it makes sense to show users only if they are activated. Arrange for this behavior by filling in the template shown in Listing 11.40.[9] (This uses the Active Record **`where`** method, which we'll learn more about in Section 13.3.3.)

3. To test the code in the previous exercise, write integration tests for both /users and /users/:id.

[9]Note that Listing 11.40 uses **`and`** in place of **`&&`**. The two are nearly identical, but the latter operator has a higher *precedence*, which binds too tightly to **`root_url`** in this case. We could fix the problem by putting **`root_url`** in parentheses, but the idiomatically correct way to do it is to use **`and`** instead.

Listing 11.39: A template for using **`update_columns`**.
app/models/user.rb

```
class User < ApplicationRecord
  attr_accessor :remember_token, :activation_token
  before_save   :downcase_email
  before_create :create_activation_digest
  .
  .
  .
  # Activates an account.
  def activate
    update_columns(activated: FILL_IN, activated_at: FILL_IN)
  end

  # Sends activation email.
  def send_activation_email
    UserMailer.account_activation(self).deliver_now
  end

  private

    # Converts email to all lower-case.
    def downcase_email
      self.email = email.downcase
    end

    # Creates and assigns the activation token and digest.
    def create_activation_digest
      self.activation_token  = User.new_token
      self.activation_digest = User.digest(activation_token)
    end
end
```

Listing 11.40: A template for code to show only active users.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def index
    @users = User.where(activated: FILL_IN).paginate(page: params[:page])
  end

  def show
    @user = User.find(params[:id])
```

```
    redirect_to root_url and return unless FILL_IN
  end
  .
  .
  .
end
```

11.4 Email in production

Now that we've got account activations working in development, in this section we'll configure our application so that it can actually send email in production. We'll first get set up with a free service to send email, and then configure and deploy our application.

To send email in production, we'll use SendGrid, which is available as an add-on at Heroku for verified accounts. (This requires adding credit card information to your Heroku account, but there is no charge when verifying an account.) For our purposes, the "starter" tier (which as of this writing is limited to 400 emails a day but costs nothing) is the best fit. We can add it to our app as follows:

```
$ heroku addons:create sendgrid:starter
```

(This might fail on systems with an older version of Heroku's command-line interface. In this case, either upgrade to the latest Heroku toolbelt or try the older syntax **heroku addons:add sendgrid:starter**.)

To configure our application to use SendGrid, we need to fill out the SMTP settings for our production environment. As shown in Listing 11.41, you will also have to define a **host** variable with the address of your production website.

Listing 11.41: Configuring Rails to use SendGrid in production.
config/environments/production.rb

```
Rails.application.configure do
  .
```