```
    user = users(:michael)
    user.reset_token = User.new_token
    mail = UserMailer.password_reset(user)
    assert_equal "Password reset", mail.subject
    assert_equal [user.email], mail.to
    assert_equal ["noreply@example.com"], mail.from
    assert_match user.reset_token,        mail.body.encoded
    assert_match CGI.escape(user.email),  mail.body.encoded
  end
end
```

At this point, the test suite should be GREEN:

Listing 12.13: 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. Run just the mailer tests. Are they GREEN?
2. Confirm that the test goes RED if you remove the second call to **`CGI.escape`** in Listing 12.12.

12.3 Resetting the password

Now that we have a correctly generated email as in Listing 12.11, we need to write the **`edit`** action in the Password Resets controller that actually resets the user's password. As in Section 11.3.3, we'll write a thorough integration test as well.

12.3.1 Reset `edit` action

Password reset emails such as that shown in Listing 12.11 contain links of the following form:

```
https://example.com/password_resets/3BdBrXeQZSWqFIDRN8cxHA/edit?email=fu%40bar.com
```

To get these links to work, we need a form for resetting passwords. The task is similar to updating users via the user edit view (Listing 10.2), but in this case with only password and confirmation fields.

There's an additional complication, though: we expect to find the user by email address, which means we need its value in both the **`edit`** and **`update`** actions. The email will automatically be available in the **`edit`** action because of its presence in the link above, but after we submit the form its value will be lost. The solution is to use a *hidden field* to place (but not display) the email on the page, and then submit it along with the rest of the form's information. The result appears in Listing 12.14.

Listing 12.14: The form to reset a password.
app/views/password_resets/edit.html.erb

```
<% provide(:title, 'Reset password') %>
<h1>Reset password</h1>

<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= form_with(model: @user, url: password_reset_path(params[:id]),
                  local: true) do |f| %>
      <%= render 'shared/error_messages' %>

      <%= hidden_field_tag :email, @user.email %>

      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>

      <%= f.label :password_confirmation, "Confirmation" %>
      <%= f.password_field :password_confirmation, class: 'form-control' %>

      <%= f.submit "Update password", class: "btn btn-primary" %>
    <% end %>
  </div>
</div>
```

Note that Listing 12.14 uses the form tag helper

```
hidden_field_tag :email, @user.email
```

instead of

```
f.hidden_field :email, @user.email
```

because the reset link puts the email in **params[:email]**, whereas the latter would put it in **params[:user][:email]**.

To get the form to render, we need to define an **@user** variable in the Password Resets controller's **edit** action. As with account activation (Listing 11.31), this involves finding the user corresponding to the email address in **params[:email]**. We then need to verify that the user is valid, i.e., that it exists, is activated, and is authenticated according to the reset token from **params[:id]** (using the generalized **authenticated?** method defined in Listing 11.26). Because the existence of a valid **@user** is needed in both the **edit** and **update** actions, we'll put the code to find and validate it in a couple of before filters, as shown in Listing 12.15.

Listing 12.15: The **edit** action for password reset.
app/controllers/password_resets_controller.rb

```
class PasswordResetsController < ApplicationController
  before_action :get_user,   only: [:edit, :update]
  before_action :valid_user, only: [:edit, :update]
  .
  .
  .
  def edit
  end

  private

    def get_user
      @user = User.find_by(email: params[:email])
    end
```

```
    # Confirms a valid user.
    def valid_user
      unless (@user && @user.activated? &&
              @user.authenticated?(:reset, params[:id]))
        redirect_to root_url
      end
    end
end
```

In Listing 12.15, compare the use of

```
authenticated?(:reset, params[:id])
```

to

```
authenticated?(:remember, cookies[:remember_token])
```

in Listing 11.28 and

```
authenticated?(:activation, params[:id])
```

in Listing 11.31. Together, these three uses complete the authentication methods shown in Table 11.1.

With the code as above, following the link from Listing 12.11 should render a password reset form. The result of pasting the link from the log (Listing 12.11) appears in Figure 12.11.

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. Follow the link in the email from the server log in Section 12.2.1. Does it properly render the form as shown in Figure 12.11?

2. What happens if you submit the form from the previous exercise?

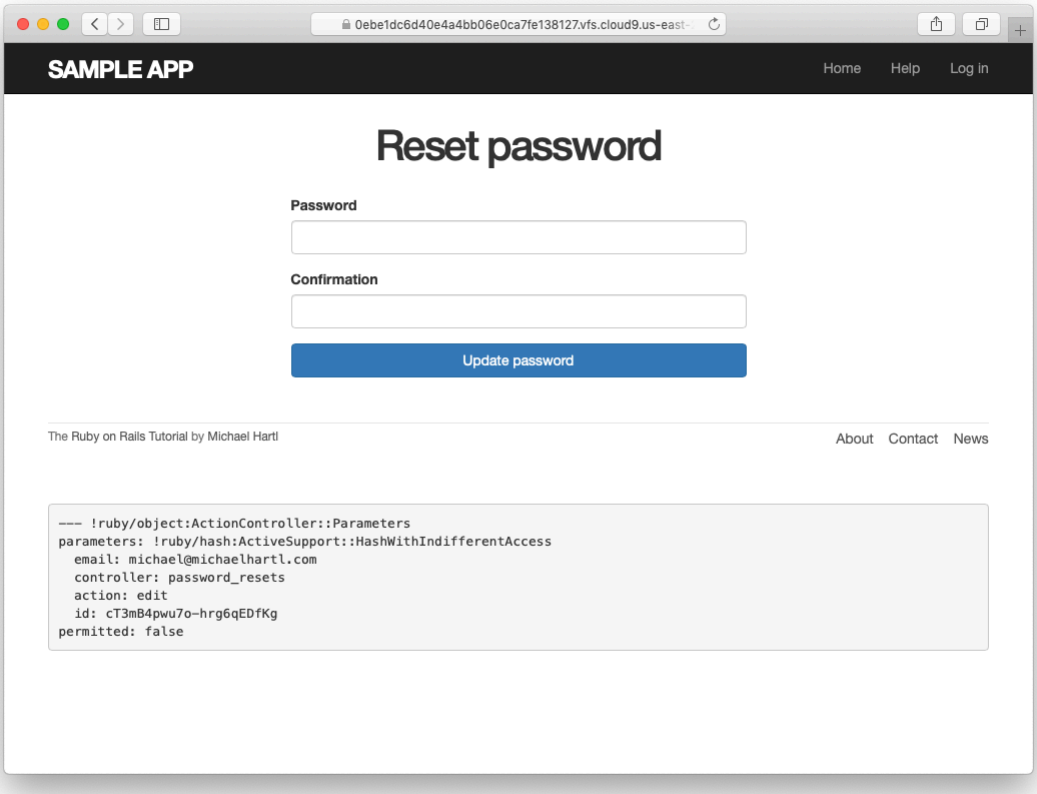

Figure 12.11: The password reset form.

12.3.2 Updating the reset

Unlike the Account Activations **edit** method, which simply toggles the user from "inactive" to "active", the **edit** method for Password Resets is a form, which must therefore submit to a corresponding **update** action. To define this **update** action, we need to consider four cases:

1. An expired password reset
2. A failed update due to an invalid password
3. A failed update (which initially looks "successful") due to an empty password and confirmation
4. A successful update

Cases (1), (2), and (4) are fairly straightforward, but Case (3) is non-obvious and is explained in more detail below.

Case (1) applies to both the **edit** and **update** actions, and so logically belongs in a before filter:

```
before_action :check_expiration, only: [:edit, :update]    # Case (1)
```

This requires defining a private **check_expiration** method:

```
# Checks expiration of reset token.
def check_expiration
  if @user.password_reset_expired?
    flash[:danger] = "Password reset has expired."
    redirect_to new_password_reset_url
  end
end
```

In the **check_expiration** method, we've deferred the expiration check to the instance method **password_reset_expired?**, which is a little tricky and will be defined in a moment.

Listing 12.16 shows the implementation of these filters, together with the **update** action that implements Cases (2)–(4). Case (2) gets handled by a failed

update, with the error messages from the shared partial in Listing 12.14 displaying automatically when the **edit** form is re-rendered. Case (4) corresponds to a successful change, and the result is similar to a successful login (Listing 8.29).

The only failure case not handled by Case (2) is when the password is empty, which is currently allowed by our User model (Listing 10.13) and so needs to be caught and handled explicitly.[3] This is Case (3) above. Our method in this case is to add an error directly to the **@user** object's error messages using **errors.add**:

```
@user.errors.add(:password, :blank)
```

This arranges to use the default message for blank content when the password is empty.[4]

The result of putting Cases (1)–(4) together is the **update** action shown in Listing 12.16.

Listing 12.16: The **update** action for password reset.
app/controllers/password_resets_controller.rb

```
class PasswordResetsController < ApplicationController
  before_action :get_user,         only: [:edit, :update]
  before_action :valid_user,       only: [:edit, :update]
  before_action :check_expiration, only: [:edit, :update]    # Case (1)

  def new
  end

  def create
    @user = User.find_by(email: params[:password_reset][:email].downcase)
    if @user
      @user.create_reset_digest
      @user.send_password_reset_email
      flash[:info] = "Email sent with password reset instructions"
      redirect_to root_url
```

[3] We need only handle the case where the password is empty because if the confirmation is empty, the confirmation validation (which is skipped if the password is empty) will catch the problem and supply a relevant error message.

[4] Alert reader Khaled Teilab has noted that one advantage of using **errors.add(:password, :blank)** is that the resulting message is automatically rendered in the correct language when using the `rails-i18n` gem.

```
    else
      flash.now[:danger] = "Email address not found"
      render 'new'
    end
  end

  def edit
  end

  def update
    if params[:user][:password].empty?                  # Case (3)
      @user.errors.add(:password, "can't be empty")
      render 'edit'
    elsif @user.update(user_params)                     # Case (4)
      log_in @user
      flash[:success] = "Password has been reset."
      redirect_to @user
    else
      render 'edit'                                     # Case (2)
    end
  end

  private

    def user_params
      params.require(:user).permit(:password, :password_confirmation)
    end

    # Before filters

    def get_user
      @user = User.find_by(email: params[:email])
    end

    # Confirms a valid user.
    def valid_user
      unless (@user && @user.activated? &&
              @user.authenticated?(:reset, params[:id]))
        redirect_to root_url
      end
    end

    # Checks expiration of reset token.
    def check_expiration
      if @user.password_reset_expired?
        flash[:danger] = "Password reset has expired."
        redirect_to new_password_reset_url
      end
    end
end
```

Note that we've added a **user_params** method permitting both the password and password confirmation attributes (Section 7.3.2).

As noted above, the implementation in Listing 12.16 delegates the boolean test for password reset expiration to the User model via the code

```
@user.password_reset_expired?
```

To get this to work, we need to define the **password_reset_expired?** method. As indicated in the email templates from Section 12.2.1, we'll consider a password reset to be expired if it was sent more than two hours ago, which we can express in Ruby as follows:

```
reset_sent_at < 2.hours.ago
```

This can be confusing if you read **<** as "less than", because then it sounds like "Password reset sent less than two hours ago", which is the opposite of what we want. In this context, it's better to read **<** as "earlier than", which gives something like "Password reset sent earlier than two hours ago." That *is* what we want, and it leads to the **password_reset_expired?** method in Listing 12.17. (For a formal demonstration that the comparison is correct, see the proof in Section 12.6.)

Listing 12.17: Adding password reset methods to the User model.
app/models/user.rb

```
class User < ApplicationRecord
  .
  .
  .
  # Returns true if a password reset has expired.
  def password_reset_expired?
    reset_sent_at < 2.hours.ago
  end

  private
    .
    .
    .
end
```

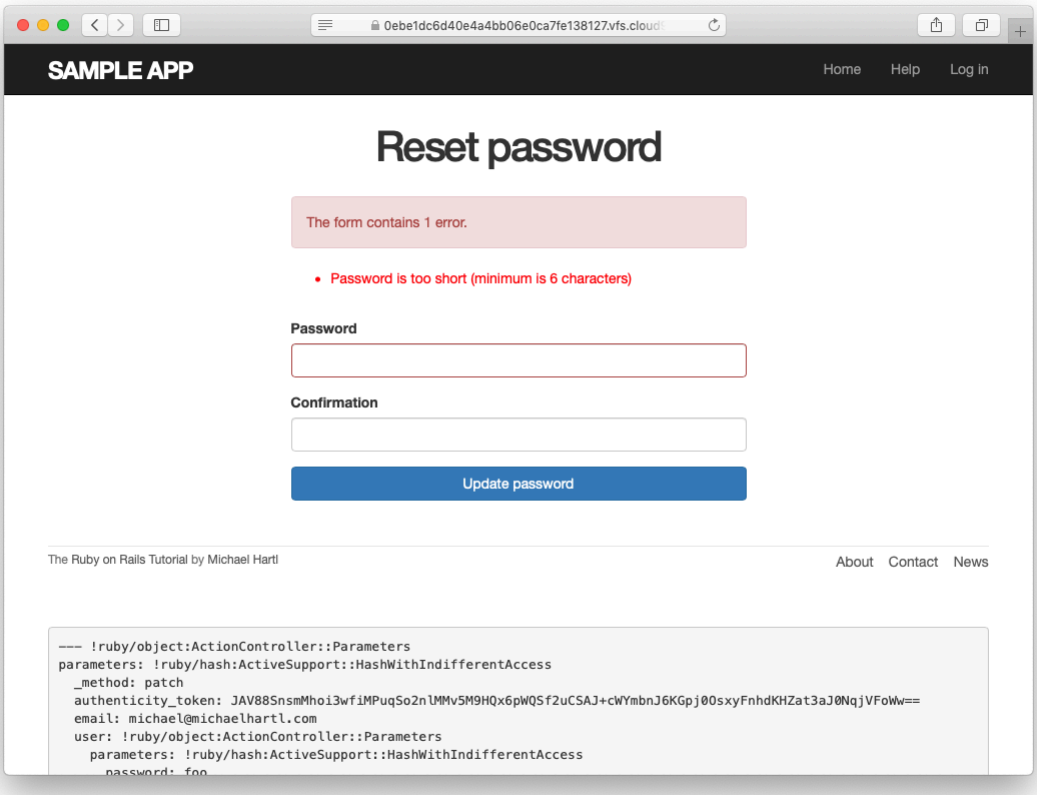

Figure 12.12: A failed password reset.

With the code in Listing 12.17, the **`update`** action in Listing 12.16 should be working. The results for invalid and valid submissions are shown in Figure 12.12 and Figure 12.13, respectively. (Lacking the patience to wait two hours, we'll cover the third branch in a test, which is left as an exercise (Section 12.3.3).)

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.

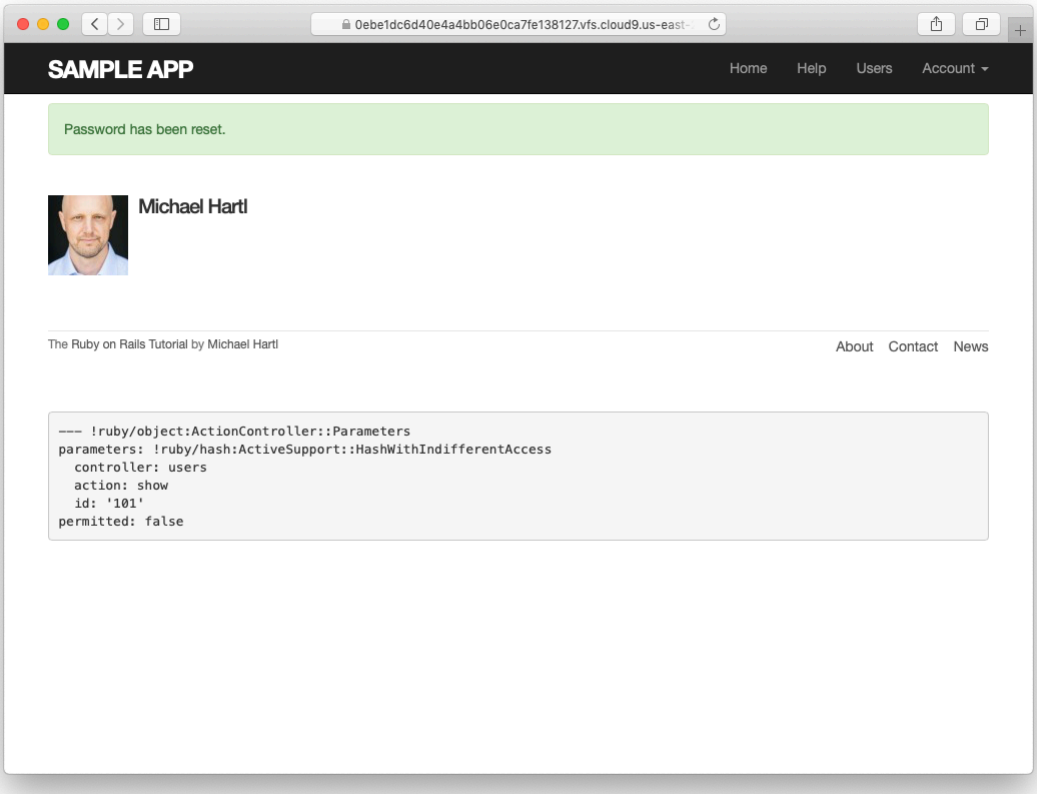

Figure 12.13: A successful password reset.

1. Follow the email link from Section 12.2.1 again and submit mismatched passwords to the form. What is the error message?

2. In the console, find the user belonging to the email link, and retrieve the value of the **`password_digest`** attribute. Now submit valid matching passwords to the form shown in Figure 12.12. Did the submission appear to work? How did it affect the value of **`password_digest`**? *Hint*: Use **`user.reload`** to retrieve the new value.

12.3.3 Password reset test

In this section, we'll write an integration test covering two of the three branches in Listing 12.16, invalid and valid submission. (As noted above, testing the third branch is left as an exercise (Section 12.3.3).) We'll get started by generating a test file for password resets:

```
$ rails generate integration_test password_resets
      invoke  test_unit
      create    test/integration/password_resets_test.rb
```

The steps to test password resets broadly parallel the test for account activation from Listing 11.33, though there is a difference at the outset: we first visit the "forgot password" form and submit invalid and then valid email addresses, the latter of which creates a password reset token and sends the reset email. We then visit the link from the email and again submit invalid and valid information, verifying the correct behavior in each case. The resulting test, shown in Listing 12.18, is an excellent exercise in reading code.

Listing 12.18: An integration test for password resets.
test/integration/password_resets_test.rb

```
require 'test_helper'

class PasswordResetsTest < ActionDispatch::IntegrationTest

  def setup
```

```
    ActionMailer::Base.deliveries.clear
    @user = users(:michael)
  end

  test "password resets" do
    get new_password_reset_path
    assert_template 'password_resets/new'
    assert_select 'input[name=?]', 'password_reset[email]'
    # Invalid email
    post password_resets_path, params: { password_reset: { email: "" } }
    assert_not flash.empty?
    assert_template 'password_resets/new'
    # Valid email
    post password_resets_path,
         params: { password_reset: { email: @user.email } }
    assert_not_equal @user.reset_digest, @user.reload.reset_digest
    assert_equal 1, ActionMailer::Base.deliveries.size
    assert_not flash.empty?
    assert_redirected_to root_url
    # Password reset form
    user = assigns(:user)
    # Wrong email
    get edit_password_reset_path(user.reset_token, email: "")
    assert_redirected_to root_url
    # Inactive user
    user.toggle!(:activated)
    get edit_password_reset_path(user.reset_token, email: user.email)
    assert_redirected_to root_url
    user.toggle!(:activated)
    # Right email, wrong token
    get edit_password_reset_path('wrong token', email: user.email)
    assert_redirected_to root_url
    # Right email, right token
    get edit_password_reset_path(user.reset_token, email: user.email)
    assert_template 'password_resets/edit'
    assert_select "input[name=email][type=hidden][value=?]", user.email
    # Invalid password & confirmation
    patch password_reset_path(user.reset_token),
          params: { email: user.email,
                    user: { password:              "foobaz",
                            password_confirmation: "barquux" } }
    assert_select 'div#error_explanation'
    # Empty password
    patch password_reset_path(user.reset_token),
          params: { email: user.email,
                    user: { password:              "",
                            password_confirmation: "" } }
    assert_select 'div#error_explanation'
    # Valid password & confirmation
    patch password_reset_path(user.reset_token),
          params: { email: user.email,
```

```
                    user: { password:              "foobaz",
                            password_confirmation: "foobaz" } }
    assert is_logged_in?
    assert_not flash.empty?
    assert_redirected_to user
  end
end
```

Most of the ideas in Listing 12.18 have appeared previously in this tutorial; the only really novel element is the test of the **input** tag:

```
assert_select "input[name=email][type=hidden][value=?]", user.email
```

This makes sure that there is an **input** tag with the right name, (hidden) type, and email address:

```
<input id="email" name="email" type="hidden" value="michael@example.com" />
```

With the code as in Listing 12.18, our test suite should be GREEN:

Listing 12.19: 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 12.6, the **create_reset_digest** method makes two calls to **update_attribute**, each of which requires a separate database operation. By filling in the template shown in Listing 12.20, replace the two **update_attribute** calls with a single call to **update_columns**,

which hits the database only once. After making the changes, verify that the test suite is still GREEN. (For convenience, Listing 12.20 includes the results of solving the exercise in Listing 11.39.)

2. Write an integration test for the expired password reset branch in Listing 12.16 by filling in the template shown in Listing 12.21. (This code introduces **`response.body`**, which returns the full HTML body of the page.) There are many ways to test for the result of an expiration, but the method suggested by Listing 12.21 is to (case-insensitively) check that the response body includes the word "expired".

3. Expiring password resets after a couple of hours is a nice security precaution, but there is an even more secure solution for cases where a public computer is used. The reason is that the password reset link remains active for 2 hours and can be used even if logged out. If a user reset their password from a public machine, anyone could press the back button and change the password (and get logged in to the site). To fix this, add the code shown in Listing 12.22 to clear the reset digest on successful password update.[5]

4. Add a line to Listing 12.18 to test for the clearing of the reset digest in the previous exercise. *Hint*: Combine **`assert_nil`** (first seen in Listing 9.25) with **`user.reload`** (Listing 11.33) to test the **`reset_digest`** attribute directly.

Listing 12.20: A template for using **`update_columns`**.
app/models/user.rb

```
class User < ApplicationRecord
  attr_accessor :remember_token, :activation_token, :reset_token
  before_save   :downcase_email
  before_create :create_activation_digest
  .
  .
  .
```

[5]Thanks to reader Tristan Ludowyk for suggesting this feature and for providing both a detailed description and a suggested implementation.

```
  # Activates an account.
  def activate
    update_columns(activated: true, activated_at: Time.zone.now)
  end

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

  # Sets the password reset attributes.
  def create_reset_digest
    self.reset_token = User.new_token
    update_columns(reset_digest:  FILL_IN, reset_sent_at: FILL_IN)
  end

  # Sends password reset email.
  def send_password_reset_email
    UserMailer.password_reset(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 12.21: A test for an expired password reset. GREEN
test/integration/password_resets_test.rb

```
require 'test_helper'

class PasswordResetsTest < ActionDispatch::IntegrationTest

  def setup
    ActionMailer::Base.deliveries.clear
    @user = users(:michael)
  end
  .
  .
  .
```

```
  test "expired token" do
    get new_password_reset_path
    post password_resets_path,
         params: { password_reset: { email: @user.email } }

    @user = assigns(:user)
    @user.update_attribute(:reset_sent_at, 3.hours.ago)
    patch password_reset_path(@user.reset_token),
          params: { email: @user.email,
                    user: { password:              "foobar",
                            password_confirmation: "foobar" } }
    assert_response :redirect
    follow_redirect!
    assert_match /FILL_IN/i, response.body
  end
end
```

Listing 12.22: Clearing the reset digest on successful password reset.
app/controllers/password_resets_controller.rb

```
class PasswordResetsController < ApplicationController
  .
  .
  .
  def update
    if params[:user][:password].empty?
      @user.errors.add(:password, "can't be empty")
      render 'edit'
    elsif @user.update(user_params)
      log_in @user
      @user.update_attribute(:reset_digest, nil)
      flash[:success] = "Password has been reset."
      redirect_to @user
    else
      render 'edit'
    end
  end
  .
  .
  .
end
```