Chapter 10

Updating, showing, and deleting users

In this chapter, we will complete the REST actions for the Users resource (Table 7.1) by adding **`edit`**, **`update`**, **`index`**, and **`destroy`** actions. We'll start by giving users the ability to update their profiles, which will also provide a natural opportunity to enforce an authorization model (made possible by the authentication code in Chapter 8). Then we'll make a listing of all users (also requiring authentication), which will motivate the introduction of sample data and pagination. Finally, we'll add the ability to destroy users, wiping them clear from the database. Since we can't allow just any user to have such dangerous powers, we'll take care to create a privileged class of administrative users authorized to delete other users.

10.1 Updating users

The pattern for editing user information closely parallels that for creating new users (Chapter 7). Instead of a **`new`** action rendering a view for new users, we have an **`edit`** action rendering a view to edit users; instead of **`create`** responding to a `POST` request, we have an **`update`** action responding to a `PATCH` request (Box 3.2). The biggest difference is that, while anyone can sign up, only the current user should be able to update their information. The authentication machinery from Chapter 8 will allow us to use a *before filter* to ensure that this is the case.

To get started, let's start work on an **`updating-users`** topic branch:

```
$ git checkout -b updating-users
```

10.1.1 Edit form

We start with the edit form, whose mockup appears in Figure 10.1.[1] To turn the mockup in Figure 10.1 into a working page, we need to fill in both the Users controller **`edit`** action and the user edit view. We start with the **`edit`** action, which requires pulling the relevant user out of the database. Note from Table 7.1 that the proper URL for a user's edit page is /users/1/edit (assuming the user's id is `1`). Recall that the id of the user is available in the **`params[:id]`** variable, which means that we can find the user with the code in Listing 10.1.

Listing 10.1: The user **`edit`** action.
app/controllers/users_controller.rb

```
class UsersController < ApplicationController

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

  def new
    @user = User.new
  end

  def create
    @user = User.new(user_params)
    if @user.save
      log_in @user
      flash[:success] = "Welcome to the Sample App!"
      redirect_to @user
    else
      render 'new'
```

[1]Image retrieved from https://www.flickr.com/photos/sashawolff/4598355045/ on 2014-08-25. Copyright © 2010 by Sasha Wolff and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

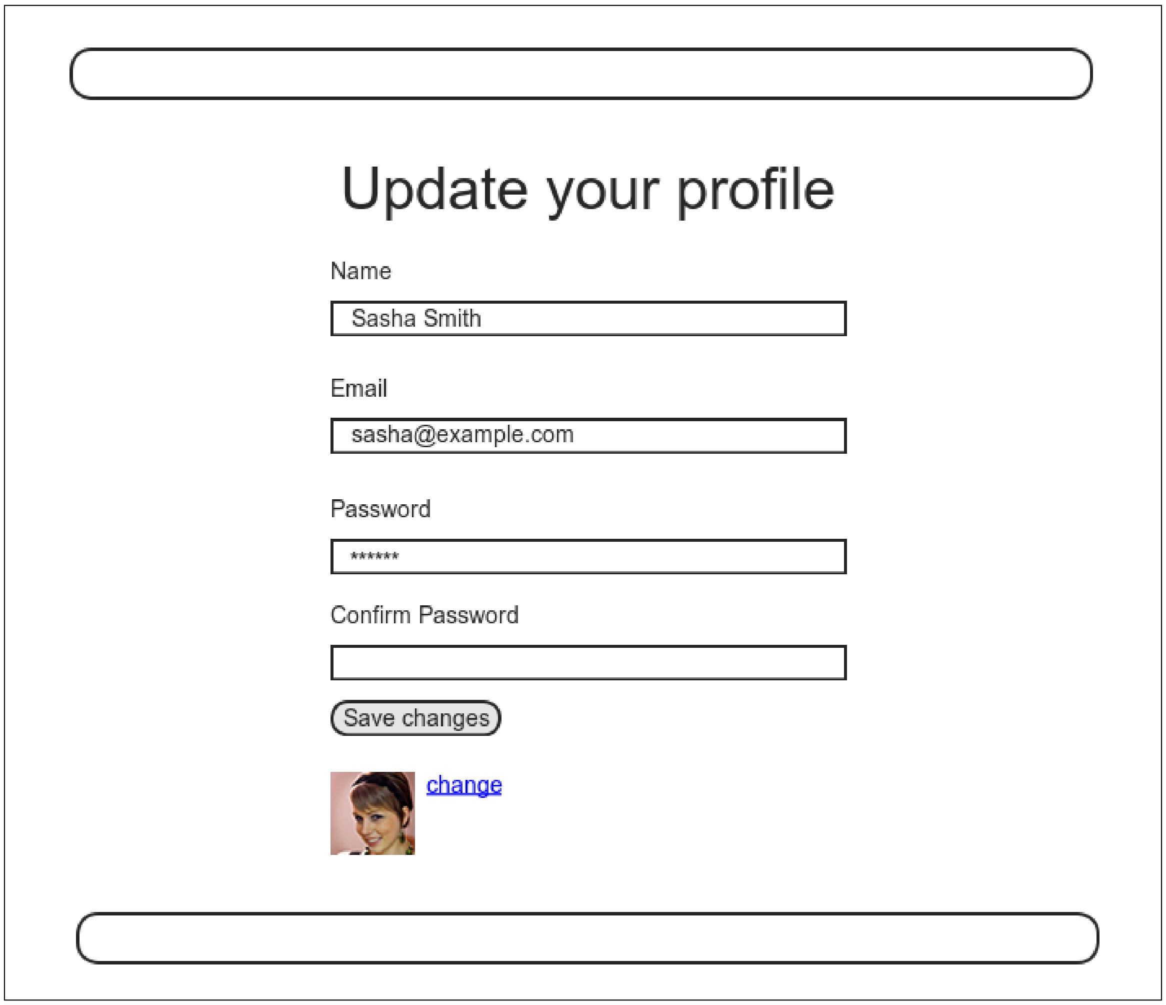


Figure 10.1: A mockup of the user edit page.

```
    end
  end

  def edit
    @user = User.find(params[:id])
  end

  private

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

The corresponding user edit view (which you will have to create by hand) is shown in Listing 10.2. Note how closely this resembles the new user view from Listing 7.15; the large overlap suggests factoring the repeated code into a partial, which is left as an exercise (Section 10.1.1).

Listing 10.2: The user edit view.
app/views/users/edit.html.erb

```
<% provide(:title, "Edit user") %>
<h1>Update your profile</h1>

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

      <%= f.label :name %>
      <%= f.text_field :name, class: 'form-control' %>

      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>

      <%= 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 "Save changes", class: "btn btn-primary" %>
    <% end %>
    <div class="gravatar_edit">
      <%= gravatar_for @user %>
      <a href="https://gravatar.com/emails" target="_blank">change</a>
    </div>
  </div>
</div>
```

Here we have reused the shared **error_messages** partial introduced in Section 7.3.3. By the way, the use of **target="_blank"** in the Gravatar link is a neat trick to get the browser to open the page in a new window or tab, which is sometimes convenient behavior when linking to third-party sites. (There's a minor security issue associated with **target="_blank"**; dealing with this detail is left as an exercise (Section 10.1.1).)

With the **@user** instance variable from Listing 10.1, the edit page should render properly, as shown in Figure 10.2. The "Name" and "Email" fields in Figure 10.2 also shows how Rails automatically pre-fills the Name and Email fields using the attributes of the existing **@user** variable.

Looking at the HTML source for Figure 10.2, we see a form tag as expected, as in Listing 10.3 (slight details may differ).

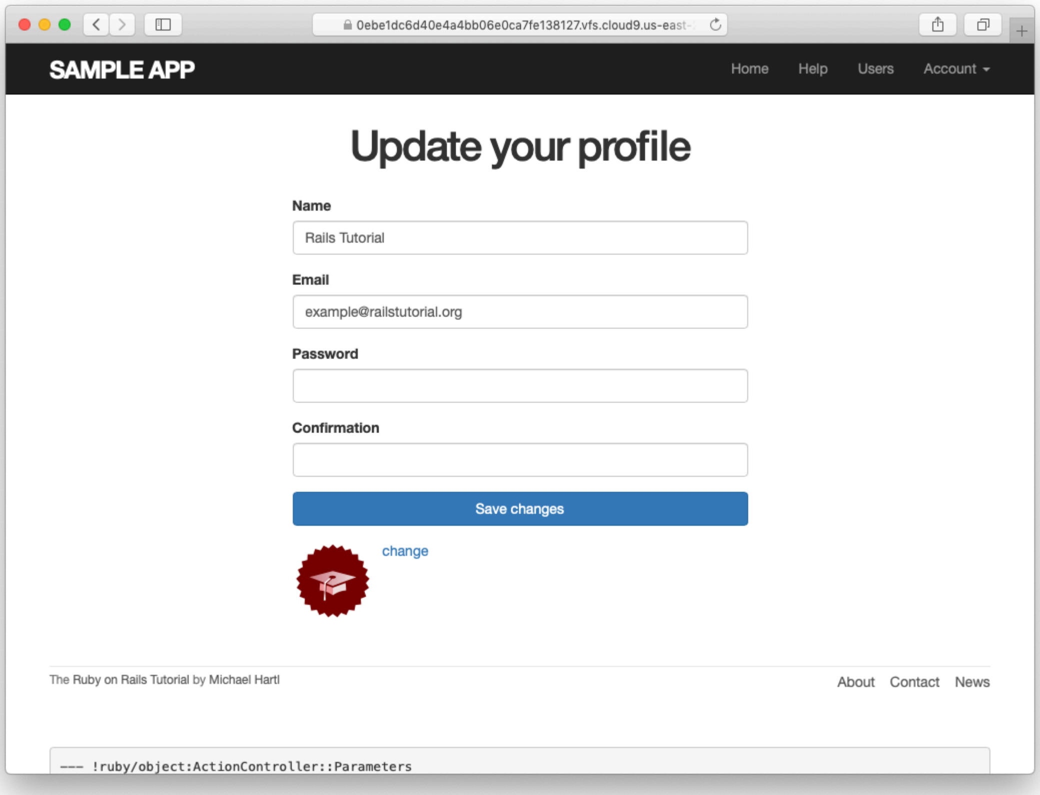
Listing 10.3: HTML for the edit form defined in Listing 10.2 and shown in Figure 10.2.

```
<form accept-charset="UTF-8" action="/users/1" class="edit_user"
      id="edit_user_1" method="post">
  <input name="_method" type="hidden" value="patch" />
  .
  .
  .
</form>
```

Note here the hidden input field:

```
<input name="_method" type="hidden" value="patch" />
```

Since web browsers can't natively send `PATCH` requests (as required by the REST conventions from Table 7.1), Rails fakes it with a `POST` request and a



```
--- !ruby/object:ActionController::Parameters
```

Figure 10.2: The initial user edit page with pre-filled name and email.

hidden **`input`** field.[2]

There's another subtlety to address here: the code **`form_with(@user)`** in Listing 10.2 is *exactly* the same as the code in Listing 7.15—so how does Rails know to use a `POST` request for new users and a `PATCH` for editing users? The answer is that it is possible to tell whether a user is new or already exists in the database via Active Record's **`new_record?`** boolean method:

```
$ rails console
>> User.new.new_record?
=> true
>> User.first.new_record?
=> false
```

When constructing a form using **`form_with(@user)`**, Rails uses `POST` if **`@user.new_record?`** is **`true`** and `PATCH` if it is **`false`**.

As a final touch, we'll fill in the URL of the settings link in the site navigation. This is easy using the named route **`edit_user_path`** from Table 7.1, together with the handy **`current_user`** helper method defined in Listing 9.9:

```
<%= link_to "Settings", edit_user_path(current_user) %>
```

The full application code appears in Listing 10.4).

Listing 10.4: Adding a URL to the "Settings" link in the site layout.
`app/views/layouts/_header.html.erb`

```
<header class="navbar navbar-fixed-top navbar-inverse">
  <div class="container">
    <%= link_to "sample app", root_path, id: "logo" %>
    <nav>
      <ul class="nav navbar-nav navbar-right">
        <li><%= link_to "Home", root_path %></li>
        <li><%= link_to "Help", help_path %></li>
        <% if logged_in? %>
          <li><%= link_to "Users", '#' %></li>
```

[2]Don't worry about how this works; the details are of interest to developers of the Rails framework itself, and by design are not important for Rails application developers.

```
          <li class="dropdown">
            <a href="#" class="dropdown-toggle" data-toggle="dropdown">
              Account <b class="caret"></b>
            </a>
            <ul class="dropdown-menu">
              <li><%= link_to "Profile", current_user %></li>
              <li><%= link_to "Settings", edit_user_path(current_user) %></li>
              <li class="divider"></li>
              <li>
                <%= link_to "Log out", logout_path, method: :delete %>
              </li>
            </ul>
          </li>
        <% else %>
          <li><%= link_to "Log in", login_path %></li>
        <% end %>
      </ul>
    </nav>
  </div>
</header>
```

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. As noted above, there's a minor security issue associated with using **`target="_blank"`** to open URLs, which is that the target site gains control of what's known as the "**`window`** object" associated with the HTML document. The result is that the target site could potentially introduce malicious content, such as a phishing page. This is extremely unlikely to happen when linking to a reputable site like Gravatar, but it turns out that we can eliminate the risk entirely by setting the **`rel`** attribute ("relationship") to **`"noopener"`** in the origin link. Add this attribute to the Gravatar edit link in Listing 10.2.

2. Remove the duplicated form code by refactoring the **`new.html.erb`** and **`edit.html.erb`** views to use the partial in Listing 10.5, as shown in Listing 10.6 and Listing 10.7. Note the use of the **`provide`** method,

which we used before in Section 3.4.3 to eliminate duplication in the layout.[3]

Listing 10.5: A partial for the **`new`** and **`edit`** form.
`app/views/users/_form.html.erb`

```
<%= form_with(model: @user, local: true) do |f| %>
  <%= render 'shared/error_messages', object: @user %>

  <%= f.label :name %>
  <%= f.text_field :name, class: 'form-control' %>

  <%= f.label :email %>
  <%= f.email_field :email, class: 'form-control' %>

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

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

  <%= f.submit yield(:button_text), class: "btn btn-primary" %>
<% end %>
```

Listing 10.6: The signup view with partial.
`app/views/users/new.html.erb`

```
<% provide(:title, 'Sign up') %>
<% provide(:button_text, 'Create my account') %>
<h1>Sign up</h1>
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= render 'form' %>
  </div>
</div>
```

Listing 10.7: The edit view with partial.
`app/views/users/edit.html.erb`

[3]Thanks to Jose Carlos Montero Gómez for a suggestion that further reduced duplication in the **`new`** and **`edit`** partials.

```
<% provide(:title, 'Edit user') %>
<% provide(:button_text, 'Save changes') %>
<h1>Update your profile</h1>
<div class="row">
  <div class="col-md-6 col-md-offset-3">
    <%= render 'form' %>
    <div class="gravatar_edit">
      <%= gravatar_for @user %>
      <a href="https://gravatar.com/emails" target="_blank">Change</a>
    </div>
  </div>
</div>
```

10.1.2 Unsuccessful edits

In this section we'll handle unsuccessful edits, following similar ideas to unsuccessful signups (Section 7.3). We start by creating an **`update`** action, which uses **`update`** (Section 6.1.5) to update the user based on the submitted **`params`** hash, as shown in Listing 10.8. With invalid information, the update attempt returns **`false`**, so the **`else`** branch renders the edit page. We've seen this pattern before; the structure closely parallels the first version of the **`create`** action (Listing 7.18).

Listing 10.8: The initial user **`update`** action.
`app/controllers/users_controller.rb`

```
class UsersController < ApplicationController

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

  def new
    @user = User.new
  end

  def create
    @user = User.new(user_params)
    if @user.save
      log_in @user
      flash[:success] = "Welcome to the Sample App!"
      redirect_to @user
    else
```

```
      render 'new'
    end
  end

  def edit
    @user = User.find(params[:id])
  end

  def update
    @user = User.find(params[:id])
    if @user.update(user_params)
      # Handle a successful update.
    else
      render 'edit'
    end
  end

  private

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

Note the use of **`user_params`** in the call to **`update`**, which uses strong parameters to prevent mass assignment vulnerability (as described in Section 7.3.2).

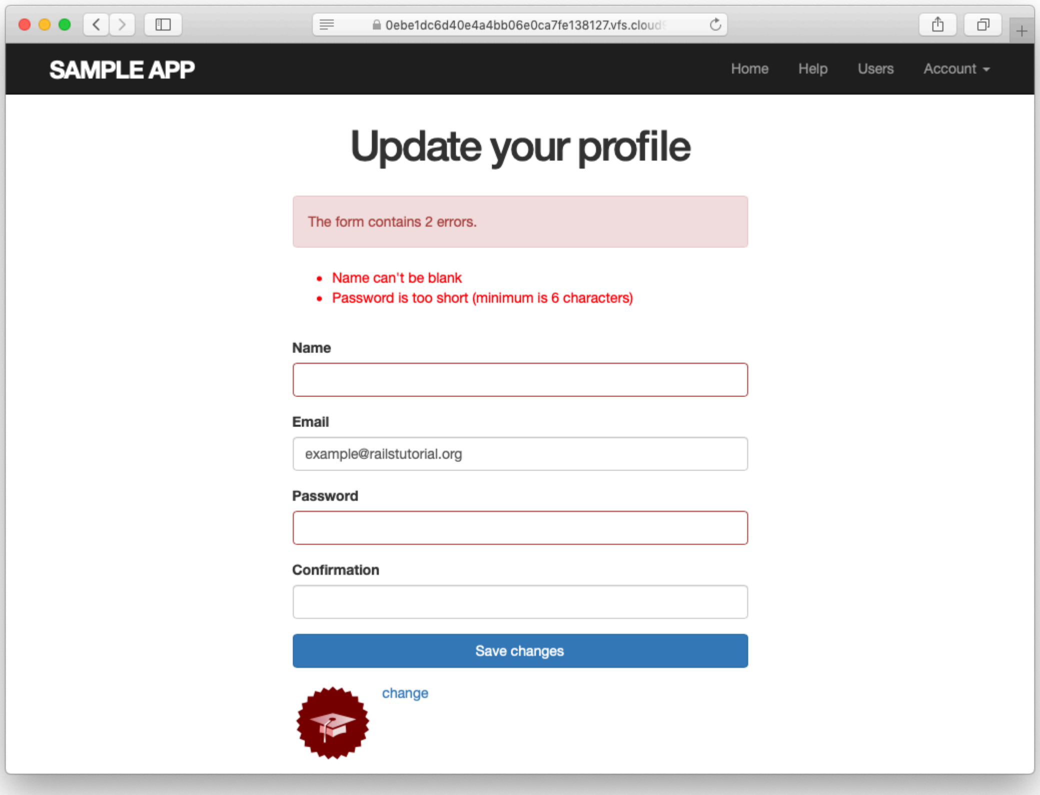
Because of the existing User model validations and the error-messages partial in Listing 10.2, submission of invalid information results in helpful error messages (Figure 10.3).

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. Confirm by submitting various invalid combinations of username, email, and password that the edit form won't accept invalid submissions.



Update your profile

The form contains 2 errors.

- Name can't be blank
- Password is too short (minimum is 6 characters)

Name

Email
example@railstutorial.org

Password

Confirmation

Figure 10.3: Error messages from submitting the update form.

10.1.3 Testing unsuccessful edits

We left Section 10.1.2 with a working edit form. Following the testing guidelines from Box 3.3, we'll now write an integration test to catch any regressions. Our first step is to generate an integration test as usual:

```
$ rails generate integration_test users_edit
      invoke  test_unit
      create    test/integration/users_edit_test.rb
```

Then we'll write a simple test of an unsuccessful edit, as shown in Listing 10.9. The test in Listing 10.9 checks for the correct behavior by verifying that the edit template is rendered after getting the edit page and re-rendered upon submission of invalid information. Note the use of the **`patch`** method to issue a `PATCH` request, which follows the same pattern as **`get`**, **`post`**, and **`delete`**.

Listing 10.9: A test for an unsuccessful edit. GREEN
`test/integration/users_edit_test.rb`

```ruby
require 'test_helper'

class UsersEditTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end

  test "unsuccessful edit" do
    get edit_user_path(@user)
    assert_template 'users/edit'
    patch user_path(@user), params: { user: { name:  "",
                                              email: "foo@invalid",
                                              password:              "foo",
                                              password_confirmation: "bar" } }

    assert_template 'users/edit'
  end
end
```

At this point, the test suite should still be GREEN:

Listing 10.10: 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.

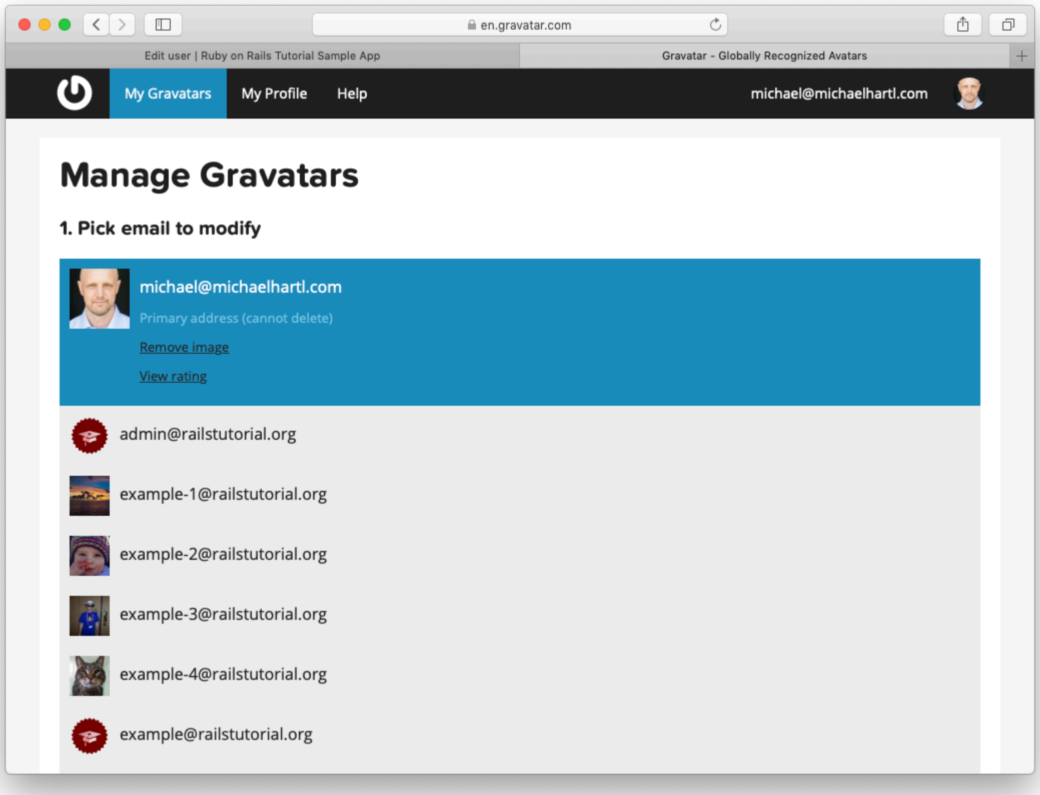
1. Add a line in Listing 10.9 to test for the correct *number* of error messages. *Hint*: Use an **`assert_select`** (Table 5.2) that tests for a **`div`** with class **`alert`** containing the text "The form contains 4 errors."

10.1.4 Successful edits (with TDD)

Now it's time to get the edit form to work. Editing the profile images is already functional since we've outsourced image upload to the Gravatar website: we can edit a Gravatar by clicking on the "change" link in Figure 10.2, which opens the Gravatar site in a new tab (due to **`target="_blank"`** in Listing 10.2), as shown in Figure 10.4. Let's get the rest of the user edit functionality working as well.

As you get more comfortable with testing, you might find that it's useful to write integration tests before writing the application code instead of after. In this context, such tests are sometimes known as *acceptance tests*, since they determine when a particular feature should be accepted as complete. To see how this works, we'll complete the user edit feature using test-driven development.

We'll test for the correct behavior of updating users by writing a test similar to the one shown in Listing 10.9, only this time we'll submit valid information. Then we'll check for a nonempty flash message and a successful redirect to the profile page, while also verifying that the user's information correctly changed in the database. The result appears in Listing 10.11. Note that the password and confirmation in Listing 10.11 are blank, which is convenient for


en.gravatar.com
Edit user | Ruby on Rails Tutorial Sample App
Gravatar - Globally Recognized Avatars
My Gravatars
My Profile
Help
michael@michaelhartl.com

Manage Gravatars

1. Pick email to modify

michael@michaelhartl.com
Primary address (cannot delete)
Remove image
View rating

admin@railstutorial.org
example-1@railstutorial.org
example-2@railstutorial.org
example-3@railstutorial.org
example-4@railstutorial.org
example@railstutorial.org

Figure 10.4: The Gravatar image-editing interface.

users who don't want to update their passwords every time they update their names or email addresses. Note also the use of **`@user.reload`** (first seen in Section 6.1.5) to reload the user's values from the database and confirm that they were successfully updated. (This is the kind of detail you could easily forget initially, which is why acceptance testing (and TDD generally) require a certain level of experience to be effective.)

Listing 10.11: A test of a successful edit. RED
test/integration/users_edit_test.rb

```
require 'test_helper'

class UsersEditTest < ActionDispatch::IntegrationTest

  def setup
    @user = users(:michael)
  end
  .
  .
  .
  test "successful edit" do
    get edit_user_path(@user)
    assert_template 'users/edit'
    name  = "Foo Bar"
    email = "foo@bar.com"
    patch user_path(@user), params: { user: { name:  name,
                                              email: email,
                                              password:              "",
                                              password_confirmation: "" } }
    assert_not flash.empty?
    assert_redirected_to @user
    @user.reload
    assert_equal name,  @user.name
    assert_equal email, @user.email
  end
end
```

The **`update`** action needed to get the tests in Listing 10.11 to pass is similar to the final form of the **`create`** action (Listing 8.29), as seen in Listing 10.12.

Listing 10.12: The user **`update`** action. RED
app/controllers/users_controller.rb

```
class UsersController < ApplicationController
  .
  .
  .
  def update
    @user = User.find(params[:id])
    if @user.update(user_params)
      flash[:success] = "Profile updated"
      redirect_to @user
    else
      render 'edit'
    end
  end
  .
  .
  .
end
```

As indicated in the caption to Listing 10.12, the test suite is still RED, which is the result of the password length validation (Listing 6.43) failing due to the empty password and confirmation in Listing 10.11. To get the tests to GREEN, we need to make an exception to the password validation if the password is empty. We can do this by passing the **`allow_nil: true`** option to **`validates`**, as seen in Listing 10.13.

Listing 10.13: Allowing empty passwords on update. GREEN
`app/models/user.rb`

```
class User < ApplicationRecord
  attr_accessor :remember_token
  before_save { self.email = email.downcase }
  validates :name, presence: true, length: { maximum: 50 }
  VALID_EMAIL_REGEX = /\A[\w+\-.]+@[a-z\d\-.]+\.[a-z]+\z/i
  validates :email, presence: true, length: { maximum: 255 },
                    format: { with: VALID_EMAIL_REGEX },
                    uniqueness: true
  has_secure_password
  validates :password, presence: true, length: { minimum: 6 }, allow_nil: true
  .
  .
  .
end
```

In case you're worried that Listing 10.13 might allow new users to sign up with empty passwords, recall from Section 6.3.3 that **`has_secure_password`** includes a separate presence validation that specifically catches **`nil`** passwords. (Because **`nil`** passwords now bypass the main presence validation but are still caught by **`has_secure_password`**, this also fixes the duplicate error message mentioned in Section 7.3.3.)

With the code in this section, the user edit page should be working (Figure 10.5), as you can double-check by re-running the test suite, which should now be GREEN:

Listing 10.14: 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. Double-check that you can now make edits by making a few changes on the development version of the application.

2. What happens when you change the email address to one without an associated Gravatar?

10.2 Authorization

In the context of web applications, *authentication* allows us to identify users of our site, while *authorization* lets us control what they can do. One nice effect of building the authentication machinery in Chapter 8 is that we are now in a position to implement authorization as well.

Although the edit and update actions from Section 10.1 are functionally complete, they suffer from a ridiculous security flaw: they allow anyone (even non-logged-in users) to access either action and update the information for any