CH10 Notes “Updating, Showing, and Deleting Users”

This chapter is about finishing the REST actions for the Users resource, and adding administrator pages/rights to allow the creation, destruction, and editing of users.

10.1 Updating Users

First, we created a new git branch:

```
git checkout –b updating-users
```

First we added a edit action definition to the **users_controller.rb** and called the current user by saying

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

We created **/views/users/edit.html.erb** to show an edit form at **www.sample_app.com/user/1/edit/**

NOTE: Use hidden input field to send PATCH requests because web browsers cannot facilitate. Rails uses *.new_record?* to question whether the form should send POST or PATCH.

Handle unsuccessful edits like unsuccessful signups by using a new method in the users_controller.rb, add code to update the data of a successful user edit submission or render the edit page again with error messages:

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

Then we created a new integration test to test for unsuccessful edits.

Handling successful edits was next…Gravatar handles image edits, so that is done! Then added a flash success message to the ‘#Handle successful update” comment section of code listed above (adding a test for a successful edit and flash message first, to perform TDD).

10.2 Authorization

Although the user update/edit functions are complete, anyone is allowed to use them right now, on any other user – we need to add **authorization** to restrict certain functions on our site to only some users.

BEFORE_ACTIONs
A **before_action** added to a controller gets run before a listed defined method, for example in:

before_action :logged_in_user, only: [:edit, :update]

the method **logged_in_user** will get called before the methods **edit** or **update** are called. (If no *only:* parameter is listed, the default of a before_action is to be called before every method in the controller)

We then added a before_action and Users controller method called **correct_user** to redirect to the root url unless the current user is only trying to edit their own user information.

FRIENDLY FORWARDING
After a non-logged in user receives an error and successfully logs in, they are always redirected to their profile page. Friendly forwarding saves the location of where they were trying to go, and then forwards them on to that destination after a successful login.
To do this we add the following code to sessions_helper.rb:

```
# Redirects to stored location (or to the default).
  def redirect_back_or(default)
    redirect_to(session[:forwarding_url] || default)
    session.delete(:forwarding_url)
  end

  # Stores the URL trying to be accessed.
  def store_location
    session[:forwarding_url] = request.original_url if request.get?
  end
```

This **store_location** method stores the original destination in **forwarding_url**, while the **redirect_back_or** method sends the user to the original destination or to the action default url.

10.3 Showing All Users

First job was to add an **index** method to users_controller.rb and check to be sure users were logged in before accessing the index method.

Then we added *@users = User.all* to the users_controller **index** method, calling all users and storing in a variable. We made a /users/index.html.erb view to iterate through each user in @users and show their gravatar and name with a link to their 'show' page. We then fixed the stub 'Users' link in the navigation to go to the users/index view.

SAMPLE USERS
We added sample users by including **faker** in our Gemfile. Then add generation code to db/seeds.rb to generate 100 users. Then run *rails db:migrate:reset* and *rails db:seed* to add generated users to the database.

PAGINATION
To start pagination, add **will_paginate** and **bootstrap-will_paginate** to the Gemfile. Then add *<%= will_paginate %>* code to user views to show pagination links on the page. Add paginate method to list users in the controller like this:

```
def index
        @users = User.paginate(page: params[:page])
end
```

We then generated a users_indext_test to test the users index page. NOTE: you can also generate test users using ruby code in the fixtures/users.yml file.

10.4 Deleting Users

First we added an **admin** Boolean attribute to the users table:

rails generate migration add_admin_to_users admin:Boolean

Then update the [timestamp]add_admin_to_users.rb migration file to add a default:false clause, making the default value of the admin attribute false (0).

We updated our seed file to add one user with admin attribute set to true, and ran *rails db:migrate:reset* and *rails db:seed* again. Then created a test to make sure that a user could not issue a PATCH command to update their admin Boolean over the web.

DESTROY ACTION
We first updated the _users partial to show delete links on the users index view if the logged-in user is an admin. Then we defined a destroy method in the users controller like this:

```
def destroy
  User.find(params[:id]).destroy
  flash[:success] = "User deleted"
  redirect_to users_url
end
```

Making sure to put the method in the :logged_in_user **before_action**, so that a user must be logged in before they can perform the destroy method. Then for more security,

we added an :admin_user **before_action** so only admins can perform the destroy method:

before_action :admin_user, only: :destroy

Then we wrote tests to verify this important security functionality.

10.5 Conclusion

What we learned: PATCH actions, strong parameters' protection, before filters, friendly forwarding, rails db:seed, using embedded ruby inside fixtures to generate users. We finished the user edit, update, index, and destroy actions.

Git add, commit, checkout, merge, push!
NOTE: Make sure to run rails db:migrate and rails db:seed on heroku:

```
$ rails test
$ git push heroku
$ heroku pg:reset DATABASE
$ heroku run rails db:migrate
$ heroku run rails db:seed
$ heroku restart
```