In this chapter, you'll see:

- Adding secure passwords to models
- Using more validations
- Adding authentication to a session
- Using rails console
- Using database transactions
- Writing an Active Record hook

CHAPTER 14

Task I: Logging In

We have a happy customer: in a short time, we've jointly put together a basic shopping cart that she can start showing to her users. She'd like to see just one more change. Right now, anyone can access the administrative functions. She'd like us to add a basic user administration system that would force you to log in to get into the administration parts of the site.

Chatting with our customer, it seems as if we don't need a particularly sophisticated security system for our application. We just need to recognize a number of people based on usernames and passwords. Once recognized, these folks can use all of the administration functions.

Iteration I1: Adding Users

Let's start by creating a model and database table to hold our administrators' usernames and passwords. Rather than store passwords in plain text, we'll store a digest hash value of the password. By doing so, we ensure that even if our database is compromised, the hash won't reveal the original password, so it can't be used to log in as this user using the forms:

```
depot> bin/rails generate scaffold User name:string password:digest
```

We declare the password as a digest type, which is another one of the nice extra touches that Rails provides. Now run the migration as usual:

```
depot> bin/rails db:migrate
```

Next, we have to flesh out the user model:

rails7/depot_r/app/models/user.rb

```
class User < ApplicationRecord
  has_secure_password
end
```

We check that the name is present and unique (that is, no two users can have the same name in the database).

Then there's the mysterious has_secure_password().

You know those forms that prompt you to enter a password and then make you reenter it in a separate field so they can validate that you typed what you thought you typed? That's exactly what has_secure_password() does for you: it tells Rails to validate that the two passwords match. This line was added for you because you specified password:digest when you generated your scaffold.

The next step is to uncomment the bcrypt-ruby gem in your Gemfile:

rails7/depot_r/Gemfile

```
# Use Active Model has_secure_password
# [https://guides.rubyonrails.org/active_model_basics.html#securepassword]
gem "bcrypt", "~> 3.1.7"
```

Next, you need to install the gem:

```
depot> bundle install
```

Finally, you need to restart your server.

With this code in place, we have the ability to present both a password and a password confirmation field in a form, as well as the ability to authenticate a user, given a name and a password.

Administering Our Users

In addition to the model and table we set up, we already have some scaffolding generated to administer the model. Let's go through it and make some tweaks as necessary.

We start with the controller. It defines the standard methods: index(), show(), new(), edit(), create(), update(), and delete(). By default, Rails omits the unintelligible password hash from the view. This means that in the case of users, there isn't much to show() except a name. So let's avoid the redirect to showing the user after a create operation. Instead, let's redirect to the user's index and add the username to the flash notice:

rails7/depot_r/app/controllers/users_controller.rb

```
  def create
    @user = User.new(user_params)

    respond_to do |format|
      if @user.save
        format.html { redirect_to users_url,
          notice: "User #{@user.name} was successfully created." }
        format.json { render :show, status: :created, location: @user }
```

```
      else
        format.html { render :new, status: :unprocessable_entity }
        format.json { render json: @user.errors,
          status: :unprocessable_entity }
      end
    end
  end
```

Let's do the same for an update operation:

```
  def update
    respond_to do |format|
      if @user.update(user_params)
➤       format.html { redirect_to users_url,
➤         notice: "User #{@user.name} was successfully updated." }
        format.json { render :show, status: :ok, location: @user }
      else
        format.html { render :edit, status: :unprocessable_entity }
        format.json { render json: @user.errors,
          status: :unprocessable_entity }
      end
    end
  end
```

While we're here, let's also order the users returned in the index by `name`:

```
  def index
➤   @users = User.order(:name)
  end
```

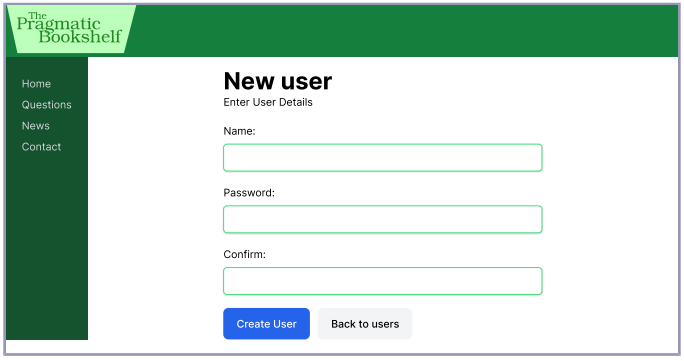
Now that the controller changes are done, let's attend to the view. We need to update the form used both to create a new user and to update an existing user. Note this form is already set up to show the password and password confirmation fields. We'll make a few aesthetic changes so the form looks nice and matches the look and feel of the site.

rails7/depot_r/app/views/users/_form.html.erb

```
<%= form_with(model: user, class: "contents") do |form| %>
  <% if user.errors.any? %>
    <div id="error_explanation"
      class="bg-red-50 text-red-500 px-3 py-2 font-medium rounded-lg mt-3">
      <h2><%= pluralize(user.errors.count, "error") %>
        prohibited this user from being saved:</h2>

      <ul>
        <% user.errors.each do |error| %>
          <li><%= error.full_message %></li>
        <% end %>
      </ul>
    </div>
  <% end %>
```

```
    <h2>Enter User Details</h2>

    <div class="my-5">
      <%= form.label :name, 'Name:' %>
      <%= form.text_field :name, class: "input-field" %>
    </div>

    <div class="my-5">
      <%= form.label :password, 'Password:' %>
      <%= form.password_field :password, class: "input-field" %>
  </div>

  <div class="my-5">
      <%= form.label :password_confirmation, 'Confirm:' %>

      <%= form.password_field :password_confirmation,
                              id: :user_password_confirmation,
                              class: "input-field" %>
    </div>

    <div class="inline">
      <%= form.submit class: "rounded-lg py-3 px-5 bg-blue-600 text-white
         inline-block font-medium cursor-pointer" %>
    </div>
  <% end %>
```

Let's try it. Navigate to http://localhost:3000/users/new. For a stunning example of page design, see the following screenshot.



After Create User is clicked, the index is redisplayed with a cheery flash notice. If we look in our database, you'll see that we've stored the user details:

```
depot> sqlite3 -line db/development.sqlite3 "select * from users"
            id = 1
          name = dave
password_digest = $2a$10$lki6/oAcOW4AWg4A0e0...
     created_at = 2022-01-10 23:52:15.599643
     updated_at = 2022-01-10 23:52:15.599643
```

As we've done before, we need to update our tests to reflect the validation and redirection changes we've made. First we update the test for the create() method:

rails7/depot_r/test/controllers/users_controller_test.rb

```
  test "should create user" do
    assert_difference("User.count") do
➤     post users_url, params: { user: { name: 'sam',
➤     password: "secret", password_confirmation: "secret" } }
    end

➤   assert_redirected_to users_url
  end
```

Because the redirect on the update() method changed too, the update test also needs to change:

```
  test "should update user" do
    patch user_url(@user), params: { user: { name: @user.name,
      password: "secret", password_confirmation: "secret" } }
➤   assert_redirected_to users_url
  end
```

We need to update the test fixtures to ensure there are no duplicate names:

rails7/depot_r/test/fixtures/users.yml

```
# Read about fixtures at
# https://api.rubyonrails.org/classes/ActiveRecord/FixtureSet.html

one:
➤   name: dave
    password_digest: <%= BCrypt::Password.create("secret") %>

two:
➤   name: adaobi
    password_digest: <%= BCrypt::Password.create("secret") %>
```

Note the use of dynamically computed values in the fixture, specifically for the value of password_digest. This code was also inserted by the scaffolding command and uses the same function that Rails uses to compute the password.[1]

At this point, we can administer our users; we need to first authenticate users and then restrict administrative functions so they'll be accessible only by administrators.

Iteration I2: Authenticating Users

What does it mean to add login support for administrators of our store?

1. https://github.com/rails/rails/blob/5-1-stable/activemodel/lib/active_model/secure_password.rb

- We need to provide a form that allows them to enter a username and password.
- Once they're logged in, we need to record that fact somehow for the rest of the session (or until they log out).
- We need to restrict access to the administrative parts of the application, allowing only people who are logged in to administer the store.

We could put all of the logic into a single controller, but it makes more sense to split it into two—a session controller to support logging in and out and a controller to welcome administrators:

```
depot> bin/rails generate controller Sessions new create destroy
depot> bin/rails generate controller Admin index
```

The SessionsController#create action will need to record something in session to say that an administrator is logged in. Let's have it store the ID of that person's User object using the key :user_id. The login code looks like this:

rails7/depot_r/app/controllers/sessions_controller.rb

```
  def create
➤   user = User.find_by(name: params[:name])
➤   if user&.authenticate(params[:password])
➤     session[:user_id] = user.id
➤     redirect_to admin_url
➤   else
➤     redirect_to login_url, alert: "Invalid user/password combination"
➤   end
  end
```

This code makes use of the Ruby Safe Navigation Operator, which checks to see if a variable has a value of nil before trying to call the method.

We're also doing something else new here: using a form that isn't directly associated with a model object. To see how that works, let's look at the template for the sessions#new action:

rails7/depot_r/app/views/sessions/new.html.erb

```
<div class="mx-auto md:w-2/3 w-full">
  <% if notice.present? %>
    <p class="py-2 px-3 bg-green-50 mb-5 text-green-500 font-medium
              rounded-lg inline-block" id="notice">
      <%= notice %>
    </p>
  <% end %>

  <%= form_tag do %>
    <h2 class="font-bold text-3xl">Please Log In</h2>
```

```
    <div class="my-5">
      <%= label_tag :name, 'Name:' %>
      <%= text_field_tag :name, params[:name], class: "payment-field" %>
    </div>

    <div class="my-5">
      <%= label_tag :password, 'Password:' %>
      <%= password_field_tag :password, params[:password],
        class: "payment-field" %>
    </div>

    <div class="actions">
      <%= submit_tag "Login",  class: "rounded-lg py-3 px-5
        bg-green-600 text-black inline-block font-medium cursor-pointer" %>
    </div>
  <% end %>
</div>
```

This form is different from ones you saw earlier. Rather than using form_with, it uses form_tag, which simply builds a regular HTML <form>. Inside that form, it uses text_field_tag and password_field_tag, two helpers that create HTML <input> tags. Each helper takes two parameters. The first is the name to give to the field, and the second is the value with which to populate the field. This style of form allows us to associate values in the params structure directly with form fields—no model object is required. In our case, we choose to use the params object directly in the form. An alternative would be to have the controller set instance variables.

We also make use of the label_tag helpers to create HTML <label> tags. This helper also accepts two parameters. The first contains the name of the field, and the second contains the label to be displayed.

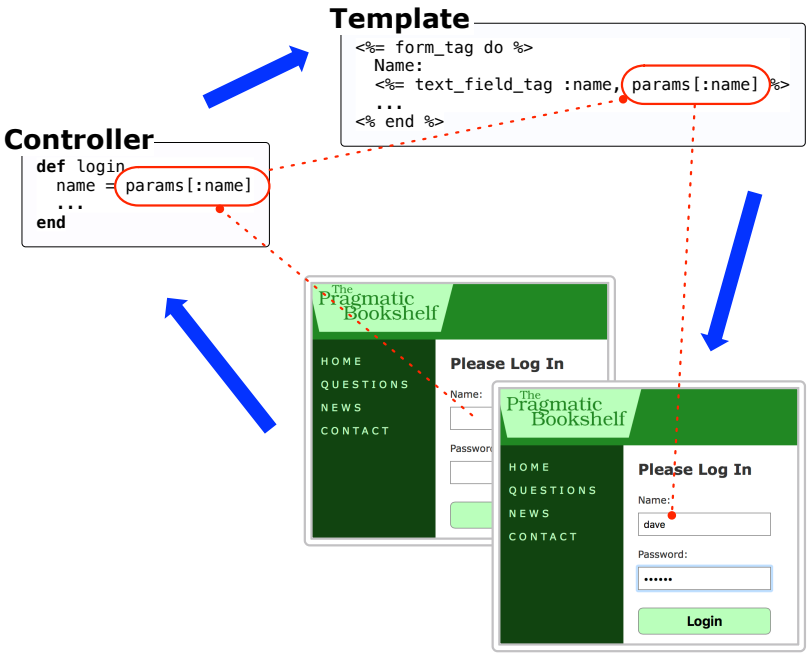
See the figure on page 214. Note how the value of the form field is communicated between the controller and the view via the params hash: the view gets the value to display in the field from params[:name], and when the user submits the form, the new field value is made available to the controller the same way.

If the user successfully logs in, we store the ID of the user record in the session data. We'll use the presence of that value in the session as a flag to indicate that an administrative user is logged in.

As you might expect, the controller actions for logging out are much shorter:

rails7/depot_r/app/controllers/sessions_controller.rb

```
  def destroy
➤   session[:user_id] = nil
➤   redirect_to store_index_url, notice: "Logged out"
  end
```



Finally, it's about time to add the index page—the first screen that administrators see when they log in. Let's make it useful. We'll have it display the total number of orders in our store. Create the template in the index.html.erb file in the app/views/admin directory. (This template uses the pluralize() helper, which in this case generates the order or orders string, depending on the cardinality of its first parameter.)

rails7/depot_r/app/views/admin/index.html.erb

```
<div class="w-full">
  <h1 class="mx-auto text-lg font-bold text-4xl">Welcome</h1>

  <p>
    It's <%= Time.now %>.
    We have <%= pluralize(@total_orders, "order") %>.
  </p>
</div>
```

The index() action sets up the count:

rails7/depot_r/app/controllers/admin_controller.rb

```
class AdminController < ApplicationController
  def index
➤   @total_orders = Order.count
  end
end
```

We have one more task to do before we can use this. Whereas previously we relied on the scaffolding generator to create our model and routes for us, this time we simply generated a controller because there's no database-backed model for this controller. Unfortunately, without the scaffolding conventions to guide it, Rails has no way of knowing which actions are to respond to GET requests, which are to respond to POST requests, and so on, for this controller. We need to provide this information by editing our config/routes.rb file:

rails7/depot_r/config/routes.rb

```
Rails.application.routes.draw do
  get 'admin' => 'admin#index'
  controller :sessions do
    get  'login' => :new
    post 'login' => :create
    delete 'logout' => :destroy
  end

  get 'sessions/create'
  get 'sessions/destroy'
  resources :users
  resources :orders
  resources :line_items
  resources :carts
  root 'store#index', as: 'store_index'
  resources :products do
    get :who_bought, on: :member
  end

  # Define your application routes per the DSL in
  # https://guides.rubyonrails.org/routing.html

  # Defines the root path route ("/")
  # root "articles#index"
end
```

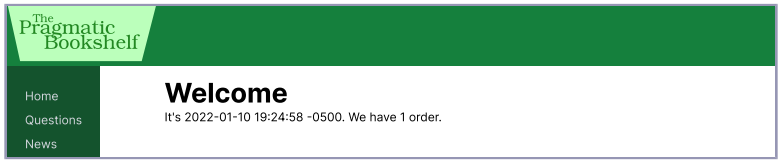
We've touched this before, when we added a root statement in Iteration C1: Creating the Catalog Listing, on page 101. What the generate command adds to this file are fairly generic get statements for each action specified. You can (and should) delete the routes provided for sessions/new, sessions/create, and sessions/destroy.

In the case of admin, we'll shorten the URL that the user has to enter (by removing the /index part) and map it to the full action. In the case of session actions, we'll completely change the URL (replacing things like session/create with simply login) as well as tailor the HTTP action that we'll match. Note that login is mapped to both the new and create actions, the difference being whether the request was an HTTP GET or HTTP POST.

We also make use of a shortcut: wrapping the session route declarations in a block and passing it to a controller() class method. This saves us a bit of typing

as well as makes the routes easier to read. We'll describe all you can do in this file in Dispatching Requests to Controllers, on page 338.

With these routes in place, we can experience the joy of logging in as an administrator. See the following screenshot.



We need to replace the functional tests in the session controller to match what was implemented. First, change the admin controller test to get the admin URL:

rails7/depot_r/test/controllers/admin_controller_test.rb

```
require "test_helper"

class AdminControllerTest < ActionDispatch::IntegrationTest
  test "should get index" do
    get admin_url
    assert_response :success
  end
end
```

Then we implement several tests for both successful and failed login attempts:

rails7/depot_r/test/controllers/sessions_controller_test.rb

```
require "test_helper"

class SessionsControllerTest < ActionDispatch::IntegrationTest
  test "should prompt for login" do
    get login_url
    assert_response :success
  end

  test "should login" do
    dave = users(:one)
    post login_url, params: { name: dave.name, password: 'secret' }
    assert_redirected_to admin_url
    assert_equal dave.id, session[:user_id]
  end

  test "should fail login" do
    dave = users(:one)
    post login_url, params: { name: dave.name, password: 'wrong' }
    assert_redirected_to login_url
  end

  test "should logout" do
    delete logout_url
    assert_redirected_to store_index_url
  end
end
```

We show our customer where we are, but she points out that we still haven't controlled access to the administrative pages (which was, after all, the point of this exercise).

Iteration I3: Limiting Access

We want to prevent people without an administrative login from accessing our site's admin pages. It turns out that we can do it with very little code using the Rails *callback* facility.

Rails callbacks allow you to intercept calls to action methods, adding your own processing before they're invoked, after they return, or both. In our case, we'll use a *before action* callback to intercept all calls to the actions in our admin controller. The interceptor can check session[:user_id]. If it's set and if it corresponds to a user in the database, the application knows an administrator is logged in and the call can proceed. If it's not set, the interceptor can issue a redirect, in this case to our login page.

Where should we put this method? It could sit directly in the admin controller, but—for reasons that'll become apparent shortly—let's put it instead in ApplicationController, the parent class of all our controllers. This is in the application_controller.rb file in the app/controllers directory. Note, too, that we chose to restrict access to this method. This prevents it from ever being exposed to end users as an action:

rails7/depot_r/app/controllers/application_controller.rb

```
class ApplicationController < ActionController::Base
  before_action :authorize

  # ...

  protected

    def authorize
      unless User.find_by(id: session[:user_id])
        redirect_to login_url, notice: "Please log in"
      end
    end
end
```

The before_action() line causes the authorize() method to be invoked before every action in our application.

This is going too far. We've just limited access to the store itself to administrators. That's not good.

We could go back and change things so that we mark only those methods that specifically need authorization. Such an approach, called *denylisting*, is

prone to errors of omission. A much better approach is to *allowlist*—list methods or controllers for which authorization is *not* required. We do this by inserting a skip_before_action() call within the StoreController:

rails7/depot_r/app/controllers/store_controller.rb
```
class StoreController < ApplicationController
➤  skip_before_action :authorize
```

And we do it again for the SessionsController class:

rails7/depot_r/app/controllers/sessions_controller.rb
```
class SessionsController < ApplicationController
➤  skip_before_action :authorize
```

We're not done yet; we need to allow people to create, update, and delete carts:

rails7/depot_r/app/controllers/carts_controller.rb
```
class CartsController < ApplicationController
➤  skip_before_action :authorize, only: %i[ create update destroy ]
```

And we allow them to create line items:

rails7/depot_r/app/controllers/line_items_controller.rb
```
class LineItemsController < ApplicationController
➤  skip_before_action :authorize, only: %i[ create ]
```

We also allow them to create orders (which includes access to the new form):

rails7/depot_r/app/controllers/orders_controller.rb
```
class OrdersController < ApplicationController
➤  skip_before_action :authorize, only: %i[ new create ]
```

With the authorization logic in place, we can now navigate to http://localhost:3000/products. The callback method intercepts us on the way to the product listing and shows us the login screen instead.

Unfortunately, this change pretty much invalidates most of our functional and system tests because most operations will now redirect to the login screen instead of doing the function desired. Fortunately, we can address this globally by creating a setup() method in the test_helper. While we're there, we also define some helper methods to login_as() and logout() a user.

We'll put those into a module because we need all of these methods to be included in both ActionDispatch::IntegrationTest and ActionDispatch::SystemTestCase. We'll define a module AuthenticationHelpers and then include that in both classes, like so:

rails7/depot_r/test/test_helper.rb
```
class ActionDispatch::IntegrationTest
  def login_as(user)
    if respond_to? :visit
      visit login_url
```

```
      fill_in :name, with: user.name
      fill_in :password, with: 'secret'
      click_on 'Login'
    else
      post login_url, params: { name: user.name, password: 'secret' }
    end
  end

  def logout
    delete logout_url
  end

  def setup
    login_as users(:one)
  end
end
```

Note that the setup() method will call login_as() only if session is defined. This prevents the login from being executed in tests that don't involve a controller.

Also note that the scaffold-generated test in test/system/users_test.rb won't be passing. That was generated by Rails for us but doesn't really represent how we implemented login. We'll leave that as an exercise for you to fix at the end of this chapter.

We show our customer and are rewarded with a big smile and a request: could we add a sidebar and put links to the user and product administration stuff in it? And while we're there, could we add the ability to list and delete administrative users? You betcha!

Iteration I4: Adding a Sidebar, More Administration

Let's start with adding links to various administration functions to the sidebar in the layout and have them show up only if a :user_id is in the session:

rails7/depot_r/app/views/layouts/application.html.erb

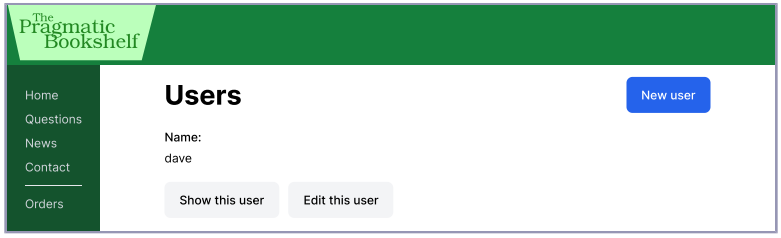
```
<!DOCTYPE html>
<html>
  <head>
    <title>Pragprog Books Online Store</title>
    <meta name="viewport" content="width=device-width,initial-scale=1">
    <%= csrf_meta_tags %>
    <%= csp_meta_tag %>
    <%= stylesheet_link_tag "inter-font", "data-turbo-track": "reload" %>
    <%= stylesheet_link_tag "tailwind", "data-turbo-track": "reload" %>

    <%= stylesheet_link_tag "application", "data-turbo-track": "reload" %>
    <%= javascript_importmap_tags %>
  </head>

  <body>
```

```
    <header class="bg-green-700">
      <%= image_tag 'logo.svg', alt: 'The Pragmatic Bookshelf' %>
      <h1><%= @page_title %></h1>
    </header>

    <section class="flex">
      <nav class="bg-green-900 p-6">
        <%= render partial: 'layouts/cart', locals: {cart: @cart } %>

        <ul class="text-gray-300 leading-8">
          <li><a href="/">Home</a></li>
          <li><a href="/questions">Questions</a></li>
          <li><a href="/news">News</a></li>
          <li><a href="/contact">Contact</a></li>
        </ul>

        <% if session[:user_id] %>
          <hr class="my-2">

          <ul class="text-gray-300 leading-8">
            <li><%= link_to 'Orders',   orders_path   %></li>
            <li><%= link_to 'Products', products_path %></li>
            <li><%= link_to 'Users',    users_path    %></li>
            <li><%= button_to 'Logout', logout_path, method: :delete %></li>
          </ul>
        <% end %>
      </nav>

      <main class="container mx-auto mt-4 px-5 flex">
        <%= yield %>
      </main>
    </section>
  </body>
</html>
```

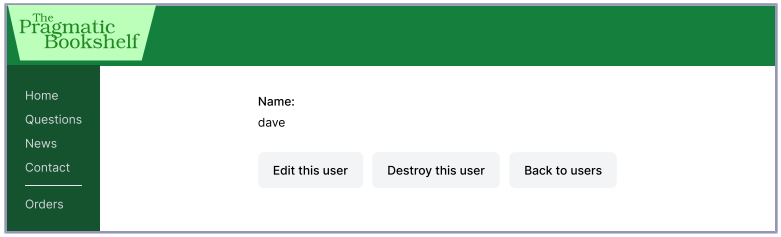
Now it's all starting to come together. We can log in, and by clicking a link in the sidebar, we can see a list of users. Let's see if we can break something.

Would the Last Admin to Leave…

We bring up the user list screen that looks something like the following screenshot:



If we click the Show this user link, we see the following:



Now click the Destroy this user link to delete that user. Sure enough, our user is removed. But to our surprise, we're then presented with the login screen instead. We just deleted the only administrative user from the system. When the next request came in, the authentication failed, so the application refused to let us in. We have to log in again before using any administrative functions.

But now we have an embarrassing problem: there are no administrative users in the database, so we can't log in.

Fortunately, we can quickly add a user to the database from the command line. If you invoke the `rails console` command, Rails invokes Ruby's `irb` utility, but it does so in the context of your Rails application. That means you can interact with your application's code by typing Ruby statements and looking at the values they return.

We can use this to invoke our user model directly, having it add a user into the database for us:

```
depot> bin/rails console
Loading development environment.
>> User.create(name: 'dave', password: 'secret', password_confirmation: 'secret')
=> #<User:0x2933060 @attributes={...} ... >
>> User.count
=> 1
```

The `>>` sequences are prompts. After the first, we call the `User` class to create a new user, and after the second, we call it again to show that we do indeed have a single user in our database. After each command we enter, `rails console` displays the value returned by the code (in the first case, it's the model object, and in the second case, it's the count).

Panic over. We can now log back in to the application. But how can we stop this from happening again? We have several ways. For example, we could write code that prevents you from deleting your own user. That doesn't quite work: in theory, A could delete B at just the same time that B deletes A. Let's try a different approach. We'll delete the user inside a database transaction.

Transactions provide an all-or-nothing proposition, stating that each work unit performed in a database must either complete in its entirety or none of them will have any effect whatsoever. If no users are left after we've deleted the user, we'll roll the transaction back, restoring the user we just deleted.

To do this, we'll use an Active Record hook method. We've already seen one of these: the validate hook is called by Active Record to validate an object's state. It turns out that Active Record defines sixteen or so hook methods, each called at a particular point in an object's life cycle. We'll use the after_destroy() hook, which is called after the SQL delete is executed. If a method by this name is publicly visible, it'll conveniently be called in the same transaction as the delete—so if it raises an exception, the transaction will be rolled back. The hook method looks like this:

rails7/depot_t/app/models/user.rb

```
after_destroy :ensure_an_admin_remains

class Error < StandardError
end

private
  def ensure_an_admin_remains
    if User.count.zero?
      raise Error.new "Can't delete last user"
    end
  end
```

The key concept is the use of an exception to indicate an error when the user is deleted. This exception serves two purposes. First, because it's raised inside a transaction, it causes an automatic rollback. By raising the exception if the users table is empty after the deletion, we undo the delete and restore that last user.

Second, the exception signals the error back to the controller, where we use a rescue_from block to handle it and report the error to the user in the notice. If you want only to abort the transaction but not otherwise signal an exception, raise an ActiveRecord::Rollback exception instead, because this is the only exception that won't be passed on by ActiveRecord::Base.transaction:

rails7/depot_t/app/controllers/users_controller.rb

```
def destroy
  @user.destroy

  respond_to do |format|
    format.html { redirect_to users_url,
      notice: "User #{@user.name} deleted" }
    format.json { head :no_content }
  end
end
```

```
rescue_from 'User::Error' do |exception|
  redirect_to users_url, notice: exception.message
end
```

This code still has a potential timing issue: it's still possible for two administrators each to delete the last two users if their timing is right. Fixing this would require more database wizardry than we have space for here.

In fact, the login system described in this chapter is rudimentary. Most applications these days use a plugin to do this.

A number of plugins are available that provide ready-made solutions that not only are more comprehensive than the authentication logic shown here but generally require less code and effort on your part to use. Devise[2] is a common and popular gem that does this.

What We Just Did

By the end of this iteration, we've done the following:

- We used has_secure_password to store an encrypted version of the password into the database.
- We controlled access to the administration functions using before action callbacks to invoke an authorize() method.
- We used rails console to interact directly with a model (and dig us out of a hole after we deleted the last user).
- We used a transaction to help prevent deletion of the last user.

Playtime

Here's some stuff to try on your own:

- Modify the user update function to require and validate the current password before allowing a user's password to be changed.
- The system test in test/system/users_test.rb was generated by the scaffolding generator we used at the start of the chapter. Those tests don't pass. See if you can get them to pass without breaking the other system tests. You'll recall we created the module AuthenticationHelpers and included it in all of the system tests by default, so you might need to change the code to *not* do that so that you can properly test the login functionality.

2. https://github.com/plataformatec/devise

When the system is freshly installed on a new machine, no administrators are defined in the database, and hence no administrator can log on. But if no administrator can log on, then no one can create an administrative user.

Change the code so that if no administrator is defined in the database, any username works to log on (allowing you to quickly create a real administrator).

- Experiment with rails console. Try creating products, orders, and line items. Watch for the return value when you save a model object—when validation fails, you'll see false returned. Find out why by examining the errors:

```
>> prd = Product.new
=> #<Product id: nil, title: nil, description: nil, image_url:
nil, created_at: nil, updated_at: nil, price:
#<BigDecimal:246aa1c,'0.0',4(8)>>
>> prd.save
=> false
>> prd.errors.full_messages
=> ["Image url must be a URL for a GIF, JPG, or PNG image",
    "Image url can't be blank", "Price should be at least 0.01",
    "Title can't be blank", "Description can't be blank"]
```

- We've gotten our tests working by performing a login, but we haven't yet written tests that verify that access to sensitive data requires login. Write at least one test that verifies this by calling logout() and then attempting to fetch or update some data that requires authentication.