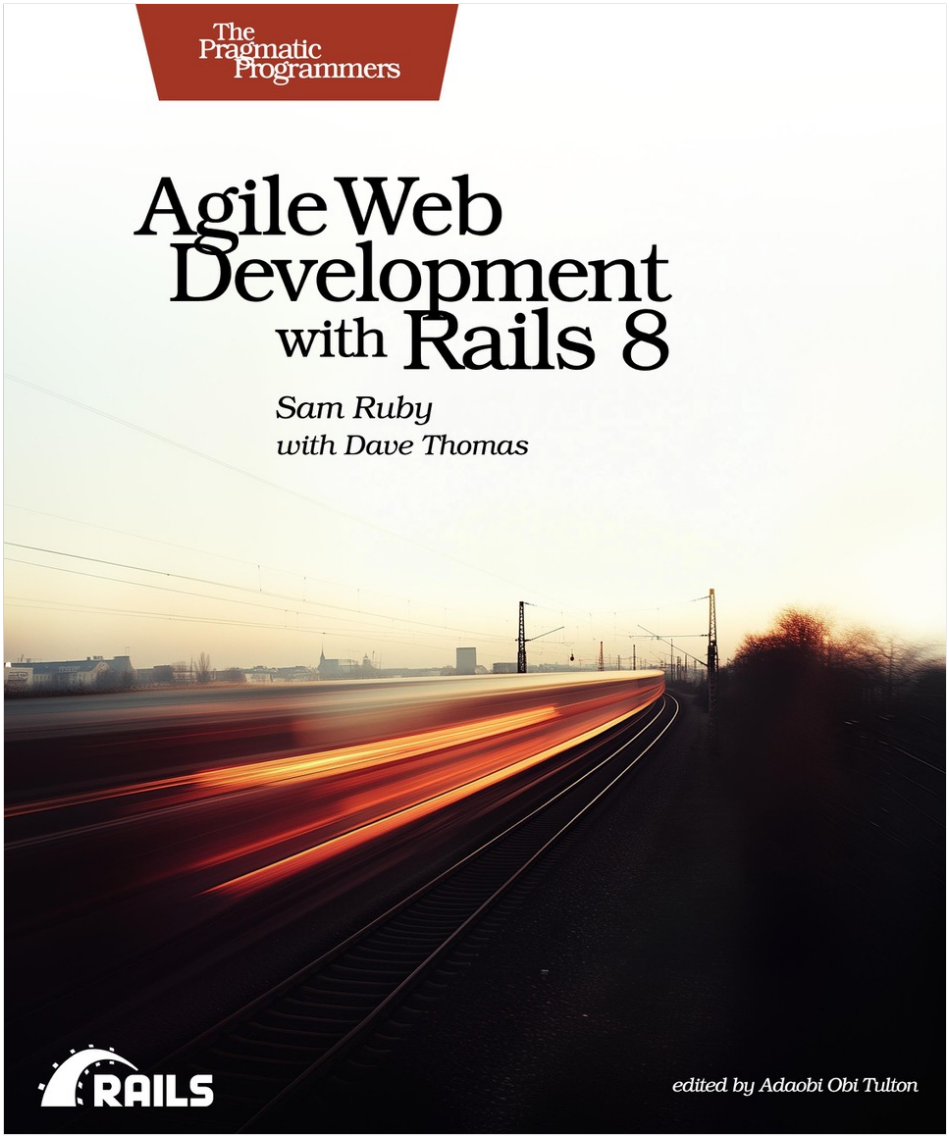


This extract shows the online version of this title, and may contain features (such as hyperlinks and colors) that are not available in the print version.

For more information, or to purchase a paperback or ebook copy, please visit https://www.pragprog.com.

Iteration I1: Authenticating Users

Building a user administration system is a common task in web applications. Rails provides a generator to help you get started. The code it generates is a good starting point, and takes care of important details like storing passwords securely. It builds upon sessions, mailers, and jobs which we've seen in previous chapters.

We start by running the generator:

```
depot> bin/rails generate authentication
```

This creates three models: Session, User, and Current. It creates controllers for sessions and passwords, and a controller concern for authentication. Finally, it creates views for passwords and their associated mailer. The one task it leaves to you the task of defining the user. We could create this from scratch, but we'll use the scaffold generator to get us started, and tell it to *not* modify what was produced by the authentication generator. The existing User model defines the user's email address and password, we just need to add the user's name.

```
depot> bin/rails generate scaffold User \
  name:string email_address:string password:digest \
  --skip-collision-check --skip
```

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

Because we skipped the collision check, we need to manually update the migration:

rails80/depot_r/db/migrate/20241021000011_create_users.rb

```
class CreateUsers < ActiveRecord::Migration[8.0]
  def change
    create_table :users do |t|
➤     t.string :name, null: false
      t.string :email_address, null: false
      t.string :password_digest, null: false

      t.timestamps
    end
    add_index :users, :email_address, unique: true
  end
end
```

And then run the migration:

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

Next, we have to flesh out the user model:

rails80/depot_r/app/models/user.rb

```
class User < ApplicationRecord
  validates :name, presence: true, uniqueness: true
  validates :email_address, presence: true, uniqueness: true
  has_secure_password
  has_many :sessions, dependent: :destroy

  normalizes :email_address, with: ->(e) { e.strip.downcase }
end
```

We check that the name and email addresses are present and unique (that is, no two users can have the same name or email address 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.

A user is defined to have many sessions, and those sessions are to be destroyed when the user is destroyed. Finally, email addresses are normalized to lowercase before being stored in the database.

Finally, you need to restart your server as a new gem was installed by the authentication generator.

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. Not bad for two commands and three lines of code.

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

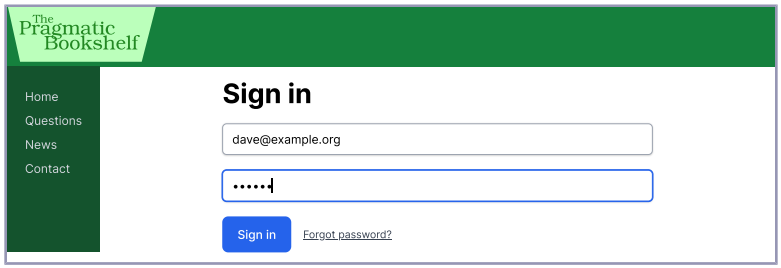
Fortunately, we can quickly add a user to the database from the command line. If you invoke the rails console command, Rails invokes Ruby's irb utility, but it does so in the context of your Rails application. That means you can interact with your application's code by typing Ruby statements and looking 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 (Rails 8.0.0.rc1)
work(dev)* User.create(name: "dave",
```

```
work(dev)>   email_address: "dave@example.org", password: "secret")
work(dev)> exit
```

With this in place, we can now log in as the user dave with the password secret:



If the blue button offends you, the file to change is app/views/sessions/new.html.erb.

You can also use this interface to send a password reset email. If you haven't set up email you can configure Rails in development mode to save the emails into files by editing the config/environments/development.rb file.

rails80/depot_r/config/environments/development.rb

```
  # Save emails as files in tmp/mails
  config.action_mailer.delivery_method = :file
```

We also have a small problem in that all of our controller tests are now failing. We can fix this by defining a method in the test helper to log in as a user:

rails80/depot_r/test/test_helper.rb

```
ENV["RAILS_ENV"] ||= "test"
require_relative "../config/environment"
require "rails/test_help"

module ActiveSupport
  class TestCase
    # Run tests in parallel with specified workers
    parallelize(workers: :number_of_processors)

    # Setup all fixtures in test/fixtures/*.yml for all tests in
  # alphabetical order.
    fixtures :all

    # Add more helper methods to be used by all tests here...
    def login_as(user)
      get users_path
      post session_path, params: {
        email_address: user.email_address,
        password: "password"
      }
    end
  end
end
```

And then each controller test needs to be updated to call this method:

rails80/depot_r/test/controllers/carts_controller_test.rb

```
  setup do
    @cart = carts(:one)
➤   login_as users(:one)
  end
```

rails80/depot_r/test/controllers/line_items_controller_test.rb

```
  setup do
    @line_item = line_items(:one)
➤   login_as users(:one)
  end
```

rails80/depot_r/test/controllers/orders_controller_test.rb

```
  setup do
    @order = orders(:one)
➤   login_as users(:one)
  end
```

rails80/depot_r/test/controllers/products_controller_test.rb

```
  setup do
    @product = products(:one)
    @title = "The Great Book #{rand(1000)}"
➤   login_as users(:one)
  end
```

rails80/depot_r/test/controllers/store_controller_test.rb

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

rails80/depot_r/test/controllers/users_controller_test.rb

```
  setup do
    @user = users(:one)
➤   login_as @user
  end
```

Once the tests are passing again, we can move on to the next step: adding the ability to administer users.

Administering Our Users

Now we turn our attention to the scaffolding we created for our users. 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 destroy(). 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 and an email. 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:

rails80/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
```

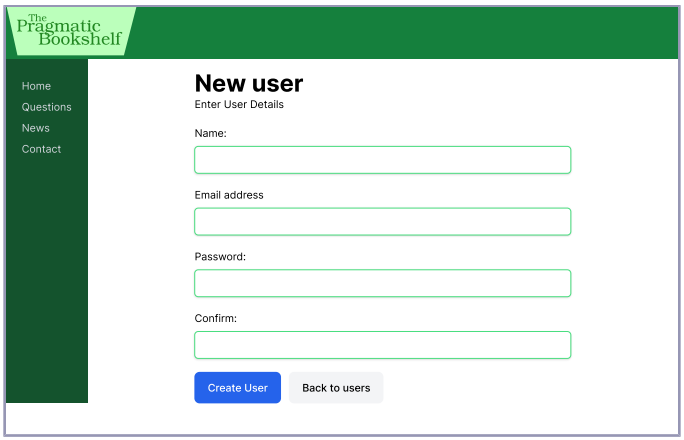
Now that the controller changes are done, let's attend to the view. We need to update the form used both to create a new user and to update an existing user. Note this form is already set up to show the password and password confirmation fields. We'll make a few aesthetic changes so the form looks nice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

rails80/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 :email_address %>
➤     <%= form.text_field :email_address, 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 storage/development.sqlite3 "select * from users"
           id = 1
         name = dave
  email_address = dave@example.org
password_digest = $2a$12$p1HwU98TtNu.j/UBv74e4.ljjpvWdPk4tN6kTkWxp1QVV7UyR73em
    created_at = 2024-10-24 14:43:27.934633
    updated_at = 2024-10-24 14:43:27.934633

           id = 2
         name = sam
  email_address = sam@example.org
password_digest = $2a$12$UQrQxRNRatkzGpwhnUQ3X.QjUQr57bcCui01wXYMjlosZO0rIzLLK
    created_at = 2024-10-24 14:43:35.232745
    updated_at = 2024-10-24 14:43:35.232745
```

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:

rails80/depot_r/test/controllers/users_controller_test.rb

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

We need to update the test fixtures to add names to the users.

rails80/depot_r/test/fixtures/users.yml

```
<% password_digest = BCrypt::Password.create("password") %>

one:
➤   name: one
    email_address: one@example.com
    password_digest: <%= password_digest %>

two:
➤   name: two
    email_address: two@example.com
    password_digest: <%= password_digest %>
```

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; and only authenticated users can access our site. Now we need to open things up so that customers can access the store.

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