Rails-Devise

Last updated 28 Jul 2014

GitHub Repository (https://github.com/RailsApps/rails-devise) · Issues (https://github.com/RailsApps/rails-devise/issues)

Introduction
A Starter App with Devise
How Rails Composer Sets Up an Application With Devise
Home Page
Navigation for Devise
Devise Architecture and API
Sign In
Sign Up
Edit Account
User Pages
Adding Attributes
Localization
Devise Testing with RSpec
Customizing the Application
Resources for Devise
Comments

Devise Quickstart Guide

Introduction

This guide shows how to use Devise in a Rails application. Versions:

- Devise 3.2
- Rails 4.1

Devise (https://github.com/plataformatec/devise) is the most popular gem for authentication and user management in Rails. Authentication securely identifies users, requiring a password to sign in to use the application. Devise also provides user management, letting a user sign up to create an account.

What's Here

You'll find what you need:

- how to build a starter application with Devise
- customizing an application with Devise
- adding tests for Devise

Is It for You?

This guide shows how to set up Devise and add important features. If you've created simple Rails applications, such as the one in the book Learn Ruby on Rails (https://tutorials.railsapps.org/learn-ruby-on-rails), you'll have enough background to follow this guide. You can find much information about Devise on the web; this guide brings together the most-needed information in one place.

A Starter App with Devise

For this tutorial, we'll build the rails-devise (https://github.com/RailsApps/rails-devise) example application using the Rails Composer (http://railsapps.github.io/rails-composer/) tool. The Rails Composer tool creates a complete starter app including:

- Devise
- Bootstrap or Foundation front-end framework
- RSpec testing framework

Elsewhere on the web you can learn how to add Devise to a default Rails starter app. I've written this tutorial because you'll need more. You'll want sign-up and sign-in pages nicely styled for Bootstrap or Foundation, plus a full test suite for your authentication and user management features. Using Rails Composer, your starter app will include all this without extra work.

In this tutorial, we'll examine the code generated by Rails Composer. You'll see how to customize the application for your needs.

Building the Application

See the article Installing Rails (http://railsapps.github.io/installing-rails.html) for instructions about setting up Rails and your development environment. Rails 4.1 must be installed in your development environment.

To build the starter app, run the command:

```
$ rails new rails-devise -m https://raw.github.com/RailsApps/rails-composer/master/composer.rb
```

You'll see a prompt:

```
question  Build a starter application?
      1)  Build a RailsApps example application
      2)  Contributed applications
      3)  Custom application
```

Enter "1" to select Build a RailsApps example application. You'll see a prompt:

```
question  Starter apps for Rails 4.1. More to come.
      1)  learn-rails
      2)  rails-bootstrap
      3)  rails-foundation
      4)  rails-mailinglist-signup
      5)  rails-omniauth
      6)  rails-devise
      7)  rails-devise-pundit
      8)  rails-signup-download
```

Choose rails-devise.

The example applications also include the rails-devise-pundit (https://github.com/RailsApps/rails-devise-pundit) example application. Pundit (https://github.com/elabs/pundit) adds authorization to control access by roles. It is often used in combination with Devise. You can explore Pundit later with the Pundit Quickstart Guide (https://tutorials.railsapps.org/tutorials/pundit-quickstart). For now, we'll build the rails-devise (https://github.com/RailsApps/rails-devise) starter app.

Choose these options to create the starter application for this tutorial:

- Web server for development? WEBrick (default)
- Web server for production? Same as development
- Database used in development? SQLite
- Template engine? ERB
- Test framework? RSpec with Capybara
- Continuous testing? None
- Front-end framework? Bootstrap 3.0
- Add support for sending email? Gmail
- Devise modules? Devise with default modules
- Use a form builder gem? None
- Install page-view analytics? None
- Set a robots.txt file to ban spiders? no
- Create a GitHub repository? no
- Use or create a project-specific rvm gemset? yes

If you have problems creating the starter app, ask for help on Stack Overflow (http://stackoverflow.com/questions/tagged/railsapps). Use the tag "railsapps" on Stack Overflow for extra attention. If you find a problem with the rails-devise (https://github.com/RailsApps/rails-devise) example application, open an issue on GitHub (https://github.com/RailsApps/rails-devise/issues).

Running the Application

After you create the application, switch to its folder to work directly in the application:

```
$ cd rails-devise
```

Launch the application by entering the command:

```
$ rails server
```

To see your application in action, open a web browser window and navigate to http://localhost:3000/ (http://localhost:3000).

Exploring the Application

Let’s explore the interface so you understand the features provided by Devise. You’ve got a starter application with a home page, a navigation bar, and the functionality needed to sign in and see a list of users.

The script that set up the starter app created the database and seeded it with an example user. You’ll see a link on the home page for “Users: 1 registered.” If you click the link, you’ll see a flash message, “You need to sign in or sign up before continuing.” The starter app is set up so you must be *authenticated* before you can access anything other than the home page.

Sign Up As a New User

Let’s create a new user and sign in. Click “Sign up” in the navigation bar, fill in the form, and click submit. Notice that the password must be at least eight characters (you can change this requirement in the config/initializers/devise.rb file). You’ll see a flash message, “Welcome! You have signed up successfully.” This message can be changed in the file config/locales/devise.en.yml.

View the List of Users

If you were attempting to view the “Users” page before you signed up, you’ll now see the “Users” page. Otherwise you’ll see the home page. Notice the navigation bar now has a link for the “Users” page. Take a look at the “Users” page. It lists the first example user plus the account you created for yourself. Click your email address in the list and you’ll see a User Profile page that lists your name and email address. Go back to the “Users” page and click the “user@example.com” link. You’ll see a message, “Access denied.” This illustrates that the signed-in user can only see his or her own profile.

Edit the User Account

Click the navigation link “Edit account.” You can update the information stored in the database or cancel your account (deleting your user record).

Sign out and Sign In

Sign out and you’ll see the home page. Click the navigation link “Sign in.” Notice two special features of the sign-in form:

- Forgot password?
- Remember me

Forgot Password

Click the “Forgot password?” link and you’ll see a form that prompts you for your email address. The starter app is configured to send email using your Gmail account. If you have Unix environment variables for your Gmail username and password, or hardcoded your credentials in the config/secrets.yml file, you can enter your email address and click “Reset Password” to receive password reset instructions by email.

Remember Me

If you select “Remember me” at the time you sign in, you will be automatically signed in whenever you visit the site. By default, you’ll be remembered for two weeks before you need to sign in again. You change the default in the file config/initializers/devise.rb.

Restricting Access to the List of Users

The “List of users” feature is accessible to all users. In most real applications, you will not want a list of all registered users to be visible to everyone. The starter app includes this page for demonstration purposes only. Later, we’ll see how to restrict it to the user who was created first. You can remove this page in your own application, if you wish. Alternatively, you can build the rails-devise-pundit (https://github.com/RailsApps/rails-devise-pundit) starter application which only allows an administrator to access the “Users” page.

Summary of the Features

That’s a tour of the features provided by Devise. You’ve seen a demonstration of basic features of authentication and user management:

- Sign up (create account)
- Sign in
- “Forgot password?” (send password reset instructions)
- “Remember me” (stay signed in in for two weeks)
- View account (also called “user profile”)
- Edit account (change name, email, or password)
- Delete account
- List of users

The “current user” (the signed-in user) can only view, edit, or delete his or her own profile.

Adding Features

The authentication and user management features of the basic application are sufficient for most websites, so the starter application may be all you need. However, Devise gives you options for additional features if needed.

Adding a User Name

Devise’s default sign-up form only asks for a user’s email address and password. Typically, you’ll want to ask for a name, and possibly other information. The rails-devise (https://github.com/RailsApps/rails-devise) example application adds a user name to the sign-up form. In this tutorial, we’ll examine the code to learn how to add other attributes.

Requiring Email Confirmation

Optionally, Devise will make sure users register with a real email address by sending an email and requiring the recipient to click a link in the email message before completing a registration. Adding the Confirmable module to the User model adds this feature. You can add this after building the application.

Accounts by Invitation

Sergio Cambra's devise_invitable (https://github.com/scambra/devise_invitable) gem enables your application (or optionally, any user) to create a temporary account using only an email address and send an email invitation containing a link to complete the account registration. The email recipient is prompted to create a password when they click the invitation link. You can add this after building the application.

Sign Up With Facebook or Twitter or …

Instead of asking for an email address, some websites allow a visitor to sign up using Facebook, Twitter, GitHub, LinkedIn, or other sites that support the OAuth standard. You can add the Omniauthable module to the User model for this feature.

Before we look at these additional features, let's see how Rails Composer adds Devise to the example application.

How Rails Composer Sets Up an Application With Devise

Devise is a Rails engine—a modular, plugin application—that is installed as a gem. Like much of Rails, the complexity is hidden, making it easy to use but difficult to examine.

We've already built our rails-devise (https://github.com/RailsApps/rails-devise) example application using Rails Composer, but before we examine the application, I'll describe how Rails Composer modifies a default Rails starter app to set up Devise. If you don't use Rails Composer, you can follow these steps to duplicate the example application. For now, read through the steps (while appreciating the work is done for you).

New App With Testing Framework

The first step, if you were building from scratch, would be to generate a default Rails starter app and switch to its directory:

```
$ rails new myapp
$ cd myapp
```

When you run Rails Composer, it uses the RailsApps Testing (https://github.com/RailsApps/rails_apps_testing) gem to set up a testing framework.

```
$ rails generate testing:configure rspec
```

Add Devise to the Gemfile

Rails Composer adds Devise to the basic Gemfile and then runs `bundle install`:

```
source 'https://rubygems.org'
.
.
.
gem 'devise'
.
.
.
```

Devise Generator

Rails Composer runs the Devise generator to create configuration and localization files:

```
$ rails generate devise:install
```

The generator creates two files:

- config/initializers/devise.rb – configuration settings
- config/locales/devise.en.yml – English language messages (localization)

You can examine these files to see how to change messages or configuration settings. We'll look closely at the localization file later in the tutorial.

You may wish to open the config/initializers/devise.rb file and glance over the configuration settings. Settings include items such as the length of time before a session will time out and require a user to sign in again (the default is 30 minutes). In most situations, you won't need to change any defaults.

Prevent Logging of Passwords

We don't want passwords written to our log file.

Rails Composer modifies the file config/initializers/filter_parameter_logging.rb to include:

```
# Configure sensitive parameters which will be filtered from the log file.
Rails.application.config.filter_parameters += [:password, :password_confirmation]
```

The parameter `:password` is present by default; Rails Composer adds `:password_confirmation` to prevent Devise from leaking the password to the log file.

Devise Initializer

Rails Composer makes one minor change to the Devise initializer file config/initializers/devise.rb:

```
# ==> Mailer Configuration
# Configure the e-mail address which will be shown in Devise::Mailer,
# note that it will be overwritten if you use your own mailer class
# with default "from" parameter.
config.mailer_sender = 'no-reply@' + Rails.application.secrets.domain_name
```

When Devise sends a password reset email message to a user, the `config.mailer_sender` setting determines the `reply-to` address. Your users only see this if they try to reply to the password reset email message. When you use Gmail to send mail, Gmail displays the Gmail account as the sender. Other email service providers may display the `config.mailer_sender` value as the sender.

User Model

Next, Rails Composer uses the Devise generator to create a User model:

```
$ rails generate devise user
```

A User model is only one possibility; you could create an Account or Profile model instead. However, most developers will recognize a User model, as it is the name of the model most often used for authentication and user management.

The Devise generator creates a database migration that populates a `users` table with attributes needed by the Devise default modules:

- email
- encrypted_password
- reset_password_token
- reset_password_sent_at
- remember_created_at
- sign_in_count
- current_sign_in_at
- last_sign_in_at
- current_sign_in_ip
- last_sign_in_ip

The Devise generator won't create a `name` attribute. Rails Composer will add the `name` attribute later.

Here's the default User model created by the Devise generator in the app/models/user.rb file:

```
class User < ActiveRecord::Base
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable
end
```

Devise Routes

Finally, the Devise generator adds a route to the config/routes.rb file:

```
Rails.application.routes.draw do
  devise_for :users
end
```

Devise Views

At this point, developers will often generate Devise view files with the command `rails generate devise:views`. It's not necessary to run the generator to create views, as the Rails engine contains all the view files inside the gem. But the view files in the gem are not very attractive, so most developers want to generate new view files and improve the appearance, manually adding styles for Bootstrap or Foundation.

Rails Composer generates attractive Devise view files for you, applying Bootstrap or Foundation CSS styles, by using the RailsLayout (https://github.com/RailsApps/rails_layout) gem.

```
$ rails generate layout:devise bootstrap3
```

User Pages

Finally, Rails Composer uses the RailsApps Pages (https://github.com/RailsApps/rails_apps_pages) gem to add a User controller, views, routes, and RSpec tests to the application.

At this point, you've generated the rails-devise (https://github.com/RailsApps/rails-devise) example application using Rails Composer. Let's examine the example application to see how Devise is integrated into a Rails application.

Home Page

We'll start with user stories that describe each feature. Then we'll examine the code that implements each user story.

Though Rails Composer has generated all the code, you'll want to examine the code so you can customize the application for your own needs. If you didn't generate the application with Rails Composer, you could copy from this tutorial to build the example application.

User Story

Here's a user story for the home page:

```
Feature: Home page
  As a visitor
  I want to visit a home page
  So I can learn more about the website
```

The user story is simple and obvious.

Rails Composer created a Visitors controller and a home page in the file app/views/visitors/index.html.erb.

Why a "Visitors" controller? Why not a "Home" controller or "Welcome" controller? Those names are acceptable. But the home page often implements a user story for a persona named "visitor," so a "Visitors" controller is appropriate.

Controller

Examine the file app/controllers/visitors_controller.rb:

```
class VisitorsController < ApplicationController
end
```

No added code is needed to render the Visitors#index view. Rails will render the view automatically, even without implementing controller actions, by using default methods inherited from a base application controller.

View

Rails Composer created an app/views/visitors/ directory for our view file.

Examine the file app/views/visitors/index.html.erb:

```
<h3>Welcome</h3>
<p><%= link_to 'Users:', users_path %> <%= User.count %> registered</p>
```

The home page provides a link to a “Users” page and displays how many users are registered.

You can modify the home page for your own application. It’s unlikely you’ll want to display a count of registered users on your home page but it’s useful as a demonstration for our tutorial.

Routing

Rails Composer added a route to the home page.

You’ll see the new route `root :to => "visitors#index"` in the file config/routes.rb:

```
Rails.application.routes.draw do
  root :to => "visitors#index"
  devise_for :users
  resources :users
end
```

Any request to the application root (http://localhost:3000/ (http://localhost:3000/)) will be directed to the VisitorsController `index` action (handled by Rails automatically unless you override it).

There’s nothing specific to Devise in our generic home page. Except, of course, the navigation bar has links for “Sign up” and “Sign in.” Let’s look at the navigation we implement for an application that uses Devise.

Navigation for Devise

The application layout used for Devise is similar to any other Rails application. Rails Composer creates an application layout with partial templates for Rails flash messages and a navigation bar. If you need to know more, refer to the book Learn Ruby on Rails (https://tutorials.railsapps.org/tutorials/learn-rails) or the Bootstrap Quickstart Guide (https://tutorials.railsapps.org/tutorials/bootstrap-quickstart).

With code in place to display flash messages, we will see the acknowledgment and alert messages generated by Devise.

For Devise, we need navigation links for “Sign in” and “Sign up,” as well as “Sign out” and “Edit account.” Rails Composer creates the links for us. Let’s look closely at the navigation links.

Here’s a user story for our navigation feature:

```
Feature: Navigation links
  As a visitor
  I want to see navigation links
  So I can find home, sign in, or sign up
```

Rails Composer has customized the navigation links partial for Devise. The navigation links contain clues to the underlying architecture and API of Devise. Let’s examine the file app/views/layouts/_navigation_links.html.erb:

```
<%# add navigation links to this file %>
<% if user_signed_in? %>
  <li><%= link_to 'Edit account', edit_user_registration_path %></li>
  <li><%= link_to 'Sign out', destroy_user_session_path, :method=>'delete' %></li>
<% else %>
  <li><%= link_to 'Sign in', new_user_session_path %></li>
  <li><%= link_to 'Sign up', new_user_registration_path %></li>
<% end %>
<% if user_signed_in? %>
  <li><%= link_to 'Users', users_path %></li>
<% end %>
```

This is the heart of the user interface for the authentication and user management features of Devise. You’ll want these links on every page of your application, so users can sign up or sign in from the home page and sign out from any page. This partial contains an important Devise view helper as well as route helpers for Devise controller actions.

The key view helper is:

```
user_signed_in?
```

It returns `true` if the user has signed in and `false` if the user is unauthenticated. The logic is simple. Signed-in users see links to “Edit account” and “Sign out”; unauthenticated users see links to “Sign in” and “Sign up.”

For our tutorial application, we have a link to the “Users” page:

```
<% if user_signed_in? %>
  <li><%= link_to 'Users', users_path %></li>
<% end %>
```

For a production application, you could remove this link or make it only visible to an administrator. You can look at the rails-devise-pundit (https://github.com/RailsApps/rails-devise-pundit) example application to see how to limit visibility to an administrator.

If you examine the route helpers, you'll see the names of the controllers that handle the Devise actions:

- "Sign up" – Registrations controller
- "Edit account" – Registrations controller
- "Sign in" – Sessions controller
- "Sign out" – Sessions controller

The Sessions controller and Registrations controller are buried in the Rails engine that is delivered inside the Devise gem. Devise is modular and components can be replaced. If your requirements are not met by Devise, you can override components such as the Sessions controller and Registrations controller.

Let's briefly examine the architecture and API of Devise so we can understand what's available.

Devise Architecture and API

We don't need to understand how Devise implements authentication and user management. To use its default features, we just need to recognize the Devise helper methods.

View and Controller Helpers

We already saw the `user_signed_in?` view helper when we looked at navigation. Here's the method again:

```
user_signed_in?
```

The `user_signed_in?` method returns `true` or `false` so we can build controllers or views that respond appropriately to unauthenticated visitors or authenticated users.

Here's another important method:

```
current_user
```

The `current_user` method returns the user object for the current signed-in user.

Here's a method that you'll add to controllers:

```
authenticate_user!
```

This method is used to prevent visitors from seeing rendered pages if they are not signed in.

Finally, Devise provides direct access to the session datastore that is serialized in a cookie. You'll seldom use this facility but you could use it instead of setting a cookie if you wanted to store a simple token for the length of a session without touching the database. You might use this facility if you want to access a snippet of data during the session, such as a referral code, without storing it permanently. This facility is not often used, but it's available if you need it. Here's the method:

```
user_session
```

Here we use it like an ordinary hash to store a tracking code:

```
user_session[:tracking_code] = 'foobar'
```

We can use these helper methods in any controller or view.

Route Helpers

Here are the route helpers you will see most often in views and controllers:

- new_user_session_path – devise/sessions#new
- user_session_path – devise/sessions#create
- destroy_user_session_path – devise/sessions#destroy

- new_user_registration_path – devise/registrations#new
- user_registration_path – devise/registrations#create
- edit_user_registration_path – devise/registrations#edit

- new_user_password_path – devise/passwords#new
- user_password_path – devise/passwords#create
- edit_user_password_path – devise/passwords#edit

Recognizing these route helpers will help you use the default Devise features.

If you wish to go beyond the default Devise features, you'll need to learn about its controllers so you can override key functions as needed. Let's first consider the Sessions controller.

Sessions Controller

To understand the purpose of the Sessions controller, consider the basic workings of the web.

The web, as originally built, was *stateless*. The server did not need to "manage state," keeping track of a sequence of events or staying in sync with the browser. The server only had to respond to a request by delivering a file. To enable ecommerce applications, *cookies* were adopted in 1997 as a way to preserve state. Each time the browser makes a request, it will send a cookie. A web application will check the value of the cookie and, if the value remains the same, the application will recognize the requests as a sequence of actions or a *session*. A session begins with the first request from a browser to a web application and continues until the browser is closed.

Rails does all the work of setting up an encrypted, tamperproof session datastore. By default, session data is saved as a cookie in the browser. You can specify other storage mechanisms but `CookieStore` is the default and the most convenient. Cookie-based sessions give us a way to manage data through multiple browser requests. The data we most often want to persist throughout a session is an object that represents the user.

Devise provides a Sessions controller (https://github.com/plataformatec/devise/blob/master/app/controllers/devise/sessions_controller.rb) that initiates a session when the user signs in. The implementation details are not important; all you need to know is that Devise provides two routes for sign in and sign out:

- Devise::SessionsController#new – `new_user_session_path`
- Devise::SessionsController#destroy – `destroy_user_session_path`

It is seldom necessary to override the Sessions controller. If you need to execute a function after the user logs in, Devise provides an `after_database_authentication` callback method to use in the User model. You could use the callback to send a "Welcome Back" email, for example.

Registrations Controller

The Devise Registrations controller (https://github.com/plataformatec/devise/blob/master/app/controllers/devise/registrations_controller.rb) accommodates creation, editing, and deleting of user accounts. In most applications. you won't need to override the Registrations controller. Older tutorials show how to replace the Registrations controller to accommodate Rails 4.0 strong parameters (http://guides.rubyonrails.org/action_controller_overview.html#strong-parameters). Starting with Devise 3.0, it is not necessary to override the Registrations controller to accommodate strong parameters. We can indicate which parameters to accept by adding an initializer file or adding a method to the parent Application controller. We'll look at the details when we explore how to add additional attributes to a sign-up form.

Other Controllers

Devise contains other controllers:

- ConfirmationsController
- OmniauthCallbacksController
- PasswordsController
- UnlocksController

You'll seldom need to override these controllers.

Overriding a Controller

Here's an example of overriding a controller, should you need to do so. Let's imagine we only let a visitor sign up if they have an invitation code.

First, examine the controller you wish to replace, in this case, the Registrations controller (https://github.com/plataformatec/devise/blob/master/app/controllers/devise/registrations_controller.rb). Create a new controller in the file app/controllers/registrations_controller.rb, overriding the methods you want to change:

```
class RegistrationsController < Devise::RegistrationsController

  def create
    invite_code = params[:invite_code]
    if invite_code == 'RSVP123'
      super
    else
      redirect_to root_path, :notice => 'Your invitation code is not valid'
    end
  end

end
```

Using `super`, we've taken all the code from the default `create` method in the Devise Registrations controller and wrapped it with a simple conditional test. We obtain the `invite_code` from a form using `params[:invite_code]` and then test if the invitation matches "RSVP123".

For this to work, you must modify the file app/views/devise/registrations/new.html.erb, adding `<%= text_field_tag :invite_code %>` to the form. This modifies the sign-up form to include a field for the invitation code.

You also must modify the Devise routes to override the default Registrations controller in the config/routes.rb file:

```
Rails.application.routes.draw do
  devise_for :users, :controllers => { :registrations => 'registrations' }
end
```

This route tells Devise to use our new Registrations controller instead of its built-in Registrations controller.

We've seen a simple example of overriding the Devise Registrations controller. We're not going to use this new Registrations controller in our tutorial application, so remove the file and restore the config/routes.rb file to its original state.

Before we go on to examine our tutorial application in more detail, here's a useful tip that will help you if you replace a Devise controller.

"Resource" in Devise Means "User"

When you examine the implementation of a Devise controller, you'll see references to a "resource." For example, in the Registrations controller, you'll see the method `build_resource` and the variable `resource`. Devise uses metaprogramming tricks to build in flexibility (at the cost of some obscurity). Devise allows you to create a user model named Account, Profile, Member, or anything else (though User is most common). The variable `resource` is a proxy for your user object, accommodating whatever name you've given it.

In simplest terms, when you see the variable `resource` in a Devise controller, assume it means `user` if you have a User model.

Now that we've learned about the architecture and API of Devise, let's look at the practical details of setting up sign in and sign up forms.

Sign In

Let's consider terminology for a moment. Do you prefer "sign in" or "log in"? User interface experts point out that "sign up" and "sign in" are difficult to distinguish at a glance, especially for visitors whose native language is not English. "Log in" has its roots in computer antiquity, when use of a computer terminal was restricted to users who had recorded their access in a system log. Today, you'll see "log in" on some websites (Wikipedia, for example) but "sign in" is standard on major consumer sites (Google, Facebook, LinkedIn). With Devise, you can change the text to your own preferences. We've used "sign up," "sign in," and "sign out" for our tutorial application.

Here's the user story for the sign-in feature:

```
Feature: Sign in
  As a user
  I want to sign in
  So I can visit protected areas of the site
```

Examine the file app/views/devise/sessions/new.html.erb:

```
<div class="authform">
  <%= form_for(resource, :as => resource_name, :url => session_path(resource_name), :html => {
:role => 'form'}) do |f| %>
    <h3>Sign in</h3>
    <%= devise_error_messages! %>
    <div class="form-group">
      <%- if devise_mapping.registerable? %>
        <%= link_to 'Sign up', new_registration_path(resource_name), class: 'right'  %>
      <% end -%>
      <%= f.label :email %>
      <%= f.email_field :email, :autofocus => true, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%- if devise_mapping.recoverable? %>
        <%= link_to "Forgot password?", new_password_path(resource_name), class: 'right' %>
      <% end -%>
      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>
    </div>
      <%= f.submit 'Sign in', :class => 'button right' %>
      <% if devise_mapping.rememberable? -%>
        <div class="checkbox">
          <%= f.check_box :remember_me %> <%= f.label :remember_me %>
        </div>
      <% end -%>
  <% end %>
</div>
```

This is the form that appears when the visitor clicks the “Sign in” link in the navigation bar.

To understand how views work in Devise, rename the file so it is no longer available as app/views/devise/sessions/new.html.erb. You could delete it, but you’ll need to restore it in a minute. Click the “Sign in” link to revisit the page. You’ll still see a sign-in form, but it won’t have the attractive Bootstrap styling and layout. By renaming the file, you’ve removed the file we’ve used to override the default Devise forms. The Devise forms are always available inside the gem, but Rails Composer has given us more attractive replacements. To replace a default Devise view, we simply add a new file in the appropriate folder. Just as we overrode the Devise registrations controller by replacing it with our own, Rails Composer generates a view file that replaces the view file hidden in the Devise gem.

Go ahead and restore our replacement file as app/views/devise/sessions/new.html.erb with the code shown above.

The CSS class `authform` is supplied by Rails Composer (actually the RailsLayout gem (https://github.com/RailsApps/rails_layout)) in the app/assets/stylesheets/framework_and_overrides.css.scss file. It simply applies some style and layout.

The `form_for` helper is a standard Rails form helper that binds a form to an object (http://guides.rubyonrails.org/form_helpers.html#binding-a-form-to-an-object). Earlier, I mentioned that `resource` in Devise means `user`. The form wrapper could be rewritten as:

```
<%= form_for(@user, :url => session_path(@user), :html => { :role => 'form'}) do |f| %>
  .
  .
  .
<% end %>
```

Try it if you like; you’ll see the sign-in form still works. Devise adds a layer of mystery by using `resource`, but it adds flexibility if the user model is named something other than User.

The `devise_error_messages!` method will display alerts set by Devise.

The `devise_mapping.registerable?` method checks if a sign-up form is available and displays a “Sign up” link. Notice the Devise route helper; `new_registration_path` is the same as the link for “Sign up” in the navigation bar.

The form contains conventional form helpers for `email` and `password` fields.

The `devise_mapping.recoverable?` method checks if the “forgot password” features is enabled (it is enabled by default) and displays a link to the “forgot password” form.

We have a “Sign in” button with some CSS styling supplied by the RailsLayout gem.

Finally the `devise_mapping.rememberable?` method checks if the “remember me” feature is enabled (again, enabled by default) and provides a checkbox for “Remember me.”

When the user submits the form, the request is handled by the Sessions controller. You can see the route helper `session_path(resource_name)` specified as the URL parameter for the `form_for` method. Devise does the work of checking if the user is registered, if the password is correct, and creating the cookie-based `user_session` object. There's nothing you need to implement to authenticate a user if you use the supplied form.

Sign Up

The "sign up" feature could also be called "register," "join," or "create account." The fundamental function is to gather the user's email address and a password and create a database record for the user.

Here's the user story for the sign-up feature:

```
Feature: Sign up
  As a visitor
  I want to sign up
  So I can visit protected areas of the site
```

Examine the file app/views/devise/registrations/new.html.erb:

```
<div class="authform">
  <%= form_for(resource, :as => resource_name, :url => registration_path(resource_name), :html
=> { :role => 'form'}) do |f| %>
    <h3>Sign up</h3>
    <%= devise_error_messages! %>
    <div class="form-group">
      <%= f.label :name %>
      <%= f.text_field :name, :autofocus => true, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :password_confirmation %>
      <%= f.password_field :password_confirmation, class: 'form-control' %>
    </div>
     <%= f.submit 'Sign up', :class => 'button right' %>
  <% end %>
</div>
```

The form is very similar to the form we examined for the sign-in view.

The most significant difference is the URL parameter for the `form_for` method. When the user submits the form, the request is handled by the Registrations controller. The route helper is `registration_path(resource_name)`.

You can try entering invalid credentials to see how the `devise_error_messages!` view helper displays alerts. Try submitting the form with an email address that is already registered, such as `user@example.com` or a blank password. You'll see an alert box within the form that warns "Email has already been taken" or "Password can't be blank."

Notice that the form contains fields for both "password" and "password confirmation." Devise does the work of checking that the contents of the two fields match before creating a new user account.

If you remove or rename this file and view the default Devise sign-up form, you'll see our form has a `name` field that is not present in the default view. Rails Composer adds the extra field because it is a common requirement to ask for the user's name. Later in this tutorial you'll learn how to add additional attributes to the sign-up form.

Edit Account

If Devise only offered a sign-in and sign-out feature, it would be purely an *authentication tool*. Because Devise can create, update, and delete a user account, it is also a *user management tool.*

The "edit account" page provides the interface for both updating and deleting a user account.

Here are the relevant user stories:

```
Feature: User edit
  As a user
  I want to edit my user profile
  So I can change my email address or password
```

```
Feature: User delete
  As a user
  I want to delete my user profile
  So I can close my account
```

Examine the file app/views/devise/registrations/edit.html.erb:

```
<div class="authform">
  <h3>Edit <%= resource_name.to_s.humanize %></h3>
  <%= form_for(resource, :as => resource_name, :url => registration_path(resource_name), :html
=> { :method => :put, :role => 'form'}) do |f| %>
    <%= devise_error_messages! %>
    <div class="form-group">
      <%= f.label :name %>
      <%= f.text_field :name, :autofocus => true, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>
      <% if devise_mapping.confirmable? && resource.pending_reconfirmation? %>
        <div>Currently waiting confirmation for: <%= resource.unconfirmed_email %></div>
      <% end %>
    </div>
    <fieldset>
      <p>Leave these fields blank if you don't want to change your password.</p>
      <div class="form-group">
        <%= f.label :password %>
        <%= f.password_field :password, :autocomplete => 'off', class: 'form-control' %>
      </div>
      <div class="form-group">
        <%= f.label :password_confirmation %>
        <%= f.password_field :password_confirmation, class: 'form-control' %>
      </div>
    </fieldset>
    <fieldset>
      <p>You must enter your current password to make changes.</p>
      <div class="form-group">
        <%= f.label :current_password %>
        <%= f.password_field :current_password, class: 'form-control' %>
      </div>
    </fieldset>
    <%= f.submit 'Update', :class => 'button right' %>
  <% end %>
</div>
<div class="authform">
<h3>Cancel Account</h3>
<p>Unhappy? We'll be sad to see you go.</p>
<%= button_to "Cancel my account", registration_path(resource_name), :data => { :confirm => "Ar
e you sure?" }, :method => :delete, :class => 'button right' %>
</div>
```

There are two forms on the page. The first form is for updating the account information. The second is for deleting the account.

When you view the page, you'll see the headline "Edit User." The statement `resource_name.to_s.humanize` obtains the name of the "resource" and capitalizes it. You'd see "Edit Member" if you'd named your user model Member.

The URL parameter for the `form_for` helper is `registration_path(resource_name)` with the additional `:method => :put`, which routes the form to the Registrations controller's `edit` action.

The `devise_mapping.confirmable?` method checks if the Devise Confirmable module is installed and verifies that the user account has been confirmed by the user. By default, the Devise Confirmable module is not installed. You may recall that Confirmable module is an option that requires the user to click a link in an email notification to create the account. The "edit account" form is flexible enough to accommodate the Devise Confirmable module if you decide to install it.

The form's remaining fields allow user to change names, email addresses, or passwords. The user can only make a change by entering the current password.

The second form allows the user to delete the account. It implements the standard Rails delete feature (http://guides.rubyonrails.org/form_helpers.html#how-do-forms-with-patch-put-or-delete-methods-work-questionmark), which uses jQuery to pop up a dialog box for confirmation before submitting the request to the Registrations controller.

The default Devise implementation will destroy the user record when the Registrations controller receives the delete request. You'll lose all records of the user's activity. It will be easy for the user to return and create a new account with the same email address. In some situations, you may wish to suspend the account or disable sign in without destroying the user record. You'll need to override the Registrations controller, implementing a new method such as `cancel_account`.

We've seen how Devise provides user management features to create, update, and delete a user account. There are two common user management features that Devise doesn't provide. First, the user may want to view their user profile. Second, an administrator may want to view a list of users. Next we'll look at how Rails Composer adds these features.

User Pages

Devise implements the basic requirements of user management. It creates a user account, provides a page for the user to update their email address or password, and allows the user to delete their account. Here are the corresponding RESTful controller actions:

- devise/registrations#create
- devise/registrations#edit
- devise/registrations#destroy

We've already seen how the "Sign up" and "Edit account" links execute these actions.

Two functions are missing, however. Typically, a web application will provide a user profile page for each user. And an administrator may want to see a list of all the registered users. That means we need the following RESTful controller actions:

- users#show – the user profile
- users#index – a list of users

User Story

Here's a user story for the user profile page:

```
Feature: User profile page
  As a user
  I want to visit my user profile page
  So I can see my personal account data
```

And for the user index page:

```
Feature: User index page
  As a user
  I want to see a list of users
  So I can see who has registered
```

Typically, only an administrator will be allowed access to a list of users. The rails-devise (https://github.com/RailsApps/rails-devise) example application doesn't provide this restriction. We'll see how we can add rudimentary access control. Alternatively, you can use the rails-devise-pundit (https://github.com/RailsApps/rails-devise-pundit) example application for role-based authorization.

Rails Composer created a Users controller and two views in the folder app/views/users/, as well as the necessary routes.

Controller

Examine the file app/controllers/users_controller.rb:

```
class UsersController < ApplicationController
  before_filter :authenticate_user!

  def index
    @users = User.all
  end

  def show
    @user = User.find(params[:id])
    unless @user == current_user
      redirect_to :back, :alert => "Access denied."
    end
  end

end
```

Authentication

To limit access to signed-in users, we add a Devise helper to our controller. We force authentication before any of the actions are called and pages rendered. Devise performs its magic with this simple helper:

```
authenticate_user!
```

We wrap the helper in a `before_filter` to check if the user is authenticated before executing any controller action.

If you wish, you can require authentication for specific controller actions, allowing unauthenticated access to all other actions.

```
before_filter :authenticate_user!, :only => [:show]
```

Or, as an alternative:

```
before_filter :authenticate_user!, :except => [:index]
```

You must remember to add `authenticate_user!` to every controller that renders pages you wish to restrict to authenticated users. If you wish to "lock down" every action in every controller, you can add `before_filter :authenticate_user!` to the ApplicationController that is the parent of all controllers. However, if you do so, you probably will want visitors to see the home page. You can do so by modifying the app/controllers/visitors_controller.rb file:

```
class VisitorsController < ApplicationController
  skip_before_filter :authenticate_user!
end
```

It is more common to see `authenticate_user!` in every controller, because it makes it obvious to developers that authentication is required in the controller.

Action: Index

The `index` action simply sets an instance variable with a collection of all the users.

```
def index
  @users = User.all
end
```

This is code you will want to improve for a production application. As implemented, any signed-in user can see a list of all registered users.

For the most rudimentary access control, you could assume the first registered user is an administrator (presumably you, when you first deploy the application). Only allow access to the page for the first registered user:

```
def index
  @users = User.all
  @admin = User.first
  unless @admin == current_user
    redirect_to :back, :alert => "Access denied."
  end
end
```

This is adequate for a simple application, but for robust role-based access control, build the rails-devise-pundit (https://github.com/RailsApps/rails-devise-pundit) starter application which only allows an administrator to access the “Users” page.

Action: Show

The `show` action restricts access to only the signed-in user.

```
def show
  @user = User.find(params[:id])
  unless @user == current_user
    redirect_to :back, :alert => "Access denied."
  end
end
```

Devise provides a helper method `current_user`. It can be used anywhere in controllers or view templates.

In this case, the `:id` parameter from the URL indicates which user profile we want to see. We find the user in the database and instantiate the `@user` instance variable. Then we check if the `current_user` is the `@user`. It’s a simple but powerful check. If the signed-in user (the current user) tries to view someone else’s profile, they’ll see an “Access denied” error message. If the user wants to see his or her own profile, Rails will render the user view.

View: User Profile

Rails Composer created an app/views/users/ directory for our view files.

Examine the file app/views/users/show.html.erb:

```
<h3>User</h3>
<p>Name: <%= @user.name if @user.name %></p>
<p>Email: <%= @user.email if @user.email %></p>
```

The page simply displays the user’s name and email address. You could add other attributes from in the User model, for example:

```
<h3>User</h3>
<p>Name: <%= @user.name if @user.name %></p>
<p>Email: <%= @user.email if @user.email %></p>
<p>Created: <%= @user.created_at if @user.created_at %></p>
```

Check the file db/schema.rb to see a list of all the attributes available in the User model:

- email
- encrypted_password
- reset_password_token
- reset_password_sent_at
- remember_created_at
- sign_in_count
- current_sign_in_at
- last_sign_in_at
- current_sign_in_ip
- last_sign_in_ip
- created_at
- updated_at
- name

View: List of Users

We use a Rails partial (http://guides.rubyonrails.org/layouts_and_rendering.html#using-partials) to generate a listing for each user. Examine the file app/views/users/_user.html.erb:

```
<td><%= user.name %></td>
<td><%= link_to user.email, user %></td>
```

You could add the creation date for the account:

```
<td><%= link_to user.name, user %></td>
<td><%= user.email %></td>
<td><%= user.created_at %></td>
```

Then we display each partial in a table. Examine the file app/views/users/index.html.erb:

```
<div class="container">
  <div class="row">
    <h3>Users</h3>
    <div class="column">
      <table class="table">
        <tbody>
          <% @users.each do |user| %>
            <tr>
              <%= render user %>
            </tr>
          <% end %>
        </tbody>
      </table>
    </div>
  </div>
</div>
```

Before you deploy this application, you'll likely want to add pagination to this view using a gem such as Kaminari (https://github.com/amatsuda/kaminari) or will_paginate (https://github.com/mislav/will_paginate). If you have only a small number of users, they'll easily fit on one page. But a single page with a thousand users takes a long time to render and display. If you have hundreds or thousands of users, adding pagination will break the list into multiple pages that load quickly.

You may want to use Haml (http://haml.info/) or Slim (http://slim-lang.com/) if you dislike all the HTML clutter.

Routing

Rails Composer has added resourceful routes `resources :users` for the user views.

You'll see the routes in the file config/routes.rb:

```
Rails.application.routes.draw do
  root :to => "visitors#index"
  devise_for :users
  resources :users
end
```

Notice that we don't have a route to our own Registrations controller; that was just an example we looked at earlier.

Route Order is Important

The order of routes in the file is important. The Devise routes must appear above the routes for the user pages. If the user routes are listed in the file above the Devise routes, you'll get an error message, "This webpage has a redirect loop," when you try to sign in.

We've now seen the full feature set for user management, as provided by Devise and enhanced by Rails Composer. Next we'll explore how to add additional attributes to the user model.

Adding Attributes

Without modification, Devise provides everything needed to create a user account with an email address and a password. Most real-world applications will need additional attributes. Rails Composer adds a `name` attribute to demonstrate how to add an additional attribute. Here we'll learn how to modify the rails-devise (https://github.com/RailsApps/rails-devise) example application to replace the `name` attribute with `first_name` and `last_name` attributes. You can add any other attributes with the same process.

We'll start by changing the database schema with a migration.

Migration

If you're using SQLite, PostgreSQL, or another SQL database, you'll need a migration to add columns to the user table. First remove the `name` column. Then add `first_name` and `last_name`. Finally, run `rake db:migrate`.

```
$ rails generate migration RemoveNameFromUsers name:string
$ rails generate migration AddFirstlastToUsers first_name:string last_name:string
$ rake db:migrate
```

You can take a look at the db/schema.rb to see how the database schema has changed.

Model

There's no need to change the User model. However, as an exercise, let's add a convenience method that returns the user's full name.

Modify the app/models/user.rb file:

```
class User < ActiveRecord::Base
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable

  def name
    [first_name, last_name].compact.join(' ')
  end

end
```

We removed the database field that stores the user's name but our new convenience method constructs a `name` from `first_name` and `last_name`. There are several ways to implement this, including `first_name + ' ' + last_name`. In the example implementation, we create an array with `first_name` and `last_name`, apply the `compact` method to remove all nil values from the array, and then join each array value with a space.

Adding the `name` convenience method offers a benefit. We won't need to modify the application views that use `user.name` (for example, the user profile or list of users).

Sign-up Form

The sign-up form is where the user enters a name, email address, and password. Let's modify the form.

Replace the contents of the file app/views/devise/registrations/new.html.erb:

```
<div class="authform">
  <%= form_for(resource, :as => resource_name, :url => registration_path(resource_name), :html
=> { :role => 'form'}) do |f| %>
    <h3>Sign up</h3>
    <%= devise_error_messages! %>
    <div class="form-group">
      <%= f.label :first_name %>
      <%= f.text_field :first_name, :autofocus => true, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :last_name %>
      <%= f.text_field :last_name, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :password_confirmation %>
      <%= f.password_field :password_confirmation, class: 'form-control' %>
    </div>
     <%= f.submit 'Sign up', :class => 'button right' %>
  <% end %>
</div>
```

We've simple removed the `name` field and added `first_name` and `last_name` fields.

Strong Parameters

It would be very pleasant if all we had to do was run a migration and add a field to a form. Interestingly, everything appears to work without errors. Go ahead and try the application. You'll see you can enter the first and last names in the form and successfully create an account. However, there's one problem. If you look for the user's name, it will be blank. The `first_name` and `last_name` fields were simply ignored when you submitted the form. The only clue, buried in the development log, is the message "Unpermitted parameters: first_name, last_name".

This behavior can be very frustrating and it has confused and irritated many Rails newcomers. But it is absolutely necessary.

Rails protects us from a class of security exploits called “mass-assignment vulnerabilities.” Rails won’t let us initialize a model with just any parameters submitted on a form. Suppose we were creating a new user and one of the user attributes was a flag allowing administrator access. A malicious hacker could create a fake form that provides a user name and sets the administrator status to “true.” Rails forces us to “white list” each of the parameters used to initialize the model. This security feature, introduced in Rails 4.0, is called “strong parameters.” If you’ve built Rails applications before, you may have created controllers that process form submissions with a `secure_params` method to screen the parameters sent from the browser.

There are three different ways to accommodate strong parameters in Devise:

- override the default Devise registrations controller
- add code to the Application controller (called “the lazy way”)
- add an initializer file

I recommend the third approach, using an initializer file, but we’ll briefly consider the other two approaches.

Override the Registrations Controller

Overriding the Registrations controller is the most obvious way to permit Devise to accept `first_name` and `last_name` parameters. With this approach, you add a `secure_params` method to a Registrations controller to allow `first_name` and `last_name` parameters. You may find older tutorials that take this approach. We’ve already looked at an example of overriding the Registrations controller. It requires adding a Registrations controller as well as modifying the config/routes.rb file. Since there’s a simpler approach, I won’t show this approach here.

Modify the Application Controller

Starting with Devise 3.0, the Devise developers offered a convenient method to permit parameters they call “the lazy way.” I don’t recommend this approach, but here’s how you do it.

Modify the file app/controllers/application_controller.rb:

```
class ApplicationController < ActionController::Base
  # Prevent CSRF attacks by raising an exception.
  # For APIs, you may want to use :null_session instead.
  protect_from_forgery with: :exception
  before_action :configure_permitted_parameters, if: :devise_controller?

  protected

  def configure_permitted_parameters
    devise_parameter_sanitizer.for(:sign_up) << :first_name << :last_name
    devise_parameter_sanitizer.for(:account_update) << :first_name << :last_name
  end

end
```

The Devise “lazy way” requires a `before_action :configure_permitted_parameters` directive that will be applied to all Devise controllers. It is a `protected` method, indicating it should not be called from other classes. For each controller action that handles an additional attribute, it uses the `devise_parameter_sanitizer` method, which acts like an array. The Ruby `<<` “shovel operator” adds additional attributes to the hash. The only Devise controller actions that need the “white list” are the `sign_up` and `account_update` methods.

There’s a disadvantage to “the lazy way.” Every request to your web application must be evaluated to determine if it uses a Devise controller. It adds unnecessary overhead. Let’s look at a better way.

Use an Initializer File

Rails Composer installs an initializer file where you can specify permitted parameters for Devise. The approach was first suggested by Ben Caldwell (https://coderwall.com/p/memu-g).

Take a look at the file config/initializers/devise_permitted_parameters.rb:

```
module DevisePermittedParameters
  extend ActiveSupport::Concern

  included do
    before_filter :configure_permitted_parameters
  end

  protected

  def configure_permitted_parameters
    devise_parameter_sanitizer.for(:sign_up) << :name
    devise_parameter_sanitizer.for(:account_update) << :name
  end

end

DeviseController.send :include, DevisePermittedParameters
```

To add `first_name` and `last_name` attributes as permitted parameters, modify two lines:

```
devise_parameter_sanitizer.for(:sign_up) << :first_name << :last_name
devise_parameter_sanitizer.for(:account_update) << :first_name << :last_name
```

These two lines tell Devise to accommodate additional attributes. If you want to add other attributes, or different attributes, modify these two statements.

The rest of the file implements a Rails *concern*. Concerns are modules that can be mixed into models and controllers to add shared code. Typically, these modules go in a app/controllers/concerns/ folder and are added to a controller with an `include` keyword. In this case, we use the Ruby `send` method to add our mixin to the `DeviseController` object, adding `include DevisePermittedParameters` to the DeviseController without actually editing the code. If this seems incomprehensible, don’t worry, as it is advanced Ruby metaprogramming that most beginners don’t encounter.

After making these changes and trying the application, you'll see the user account includes the first and last names.

Edit Account Form

We've successfully updated the application to create a user account with a first and last name. But we need to accommodate the users who want to change their names. Most likely, it won't be users who have suddenly decided to adopt the moniker Shamus Beaglehole (a real name, see the 2014 Name of the Year (http://www.nameoftheyear.com/)). More commonly, users need to correct an entry error.

Replace the contents of the file app/views/devise/registrations/new.html.erb:

```
<div class="authform">
  <h3>Edit <%= resource_name.to_s.humanize %></h3>
  <%= form_for(resource, :as => resource_name, :url => registration_path(resource_name), :html
=> { :method => :put, :role => 'form'}) do |f| %>
    <%= devise_error_messages! %>
    <div class="form-group">
      <%= f.label :first_name %>
      <%= f.text_field :first_name, :autofocus => true, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :last_name %>
      <%= f.text_field :last_name, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>
      <% if devise_mapping.confirmable? && resource.pending_reconfirmation? %>
        <div>Currently waiting confirmation for: <%= resource.unconfirmed_email %></div>
      <% end %>
    </div>
    <fieldset>
      <p>Leave these fields blank if you don't want to change your password.</p>
      <div class="form-group">
        <%= f.label :password %>
        <%= f.password_field :password, :autocomplete => 'off', class: 'form-control' %>
      </div>
      <div class="form-group">
        <%= f.label :password_confirmation %>
        <%= f.password_field :password_confirmation, class: 'form-control' %>
      </div>
    </fieldset>
    <fieldset>
      <p>You must enter your current password to make changes.</p>
      <div class="form-group">
        <%= f.label :current_password %>
        <%= f.password_field :current_password, class: 'form-control' %>
      </div>
    </fieldset>
    <%= f.submit 'Update', :class => 'button right' %>
  <% end %>
</div>
<div class="authform">
<h3>Cancel Account</h3>
<p>Unhappy? We'll be sad to see you go.</p>
<%= button_to "Cancel my account", registration_path(resource_name), :data => { :confirm => "Ar
e you sure?" }, :method => :delete, :class => 'button right' %>
</div>
```

As we did with the sign-up form, we've simple removed the `name` field and added `first_name` and `last_name` fields.

We've already modified the config/initializers/devise_permitted_parameters.rb initializer file to allow the additional parameters for the Registrations controller `account_update` action.

```
devise_parameter_sanitizer.for(:account_update) << :first_name << :last_name
```

This concludes our look at additional attributes. Use this as a reference when you need to set up a new application with Devise. It's very likely you will need additional attributes for any application.

Localization

When we created the tutorial application, we saw that the Devise generator created a file:

- config/locales/devise.en.yml – English language messages

This file provides the default localization settings, including the messages we've seen such as, "Welcome! You have signed up successfully" and "Signed in successfully."

Non-English Languages

If English is not the primary language of your website visitors, you can add the devise-i18n (https://github.com/tigrish/devise-i18n) gem and set your default locale in the config/application.rb file with the setting `config.i18n.default_locale = :en`, replacing `:en` with your language designation. Devise provides support for dozens of languages (http://www.localeapp.com/projects/377).

Custom Messages

Here are some of the messages found in the config/locales/devise.en.yml localization file:

```
.
.
.
registrations:
  destroyed: "Bye! Your account was successfully cancelled. We hope to see you again soon."
  signed_up: "Welcome! You have signed up successfully."
  signed_up_but_inactive: "You have signed up successfully. However, we could not sign you in b
ecause your account is not yet activated."
  signed_up_but_locked: "You have signed up successfully. However, we could not sign you in bec
ause your account is locked."
  signed_up_but_unconfirmed: "A message with a confirmation link has been sent to your email ad
dress. Please open the link to activate your account."
  update_needs_confirmation: "You updated your account successfully, but we need to verify your
 new email address. Please check your email and click on the confirm link to finalize confirmin
g your new email address."
  updated: "You updated your account successfully."
sessions:
  signed_in: "Signed in successfully."
  signed_out: "Signed out successfully."
.
.
.
```

If there's a message that you don't like, simply replace the message string in the localization file.

Devise Testing with RSpec

If you are new to testing, the RSpec Quickstart Guide (https://tutorials.railsapps.org/tutorials/rspec-quickstart) gives an introduction and explains how to get started.

As explained in RSpec Quickstart Guide, developers place a priority on *unit testing* of models plus feature testing (also called *integration testing* or *acceptance testing*).

Experienced developers know there's no need to write tests for the functionality of gems, if the gems are popular and well-supported. That's certainly the case with Devise. The Devise engine is maintained by some of the most experienced developers in the Rails community and thousands of developers use it in production applications. We can safely assume that Devise works as advertised and any problems will be quickly identified and fixed.

That said, we should test our interaction with Devise to make sure our application works as expected. That means feature specs to test the implementation of all our user stories. Feature tests with Devise are not different from feature tests in general, but we'll look closely at a few tricky issues.

Rails Composer installs a full set of tests for authentication and user management. You'll find the tests in these folders:

- spec/features/visitors/ – feature tests for a "visitors" persona
- spec/features/users/ – feature tests for a "users" persona
- spec/models/user_spec.rb – unit test for the User model

First we'll look at tricky issues with the feature tests.

Setting up RSpec

Prior to RSpec 3.0, every RSpec test file had to include `require 'spec_helper'`. With RSpec 3.0 and later versions, an additional `require 'rails_helper'` became necessary. Fortunately, starting with RSpec 3.0, to eliminate clutter and duplication, these requirements can be specified in the project setting .rspec file. Your .rspec file should look like this:

```
--color
--format documentation
--require spec_helper
--require rails_helper
```

The .rspec file is required for the examples in this tutorial.

Session Helpers

To test the sign-up feature, we'll need a half-dozen scenarios. Each requires visiting the sign-up page and filling in the email, password, and password confirmation before clicking the "Sign up" button. Rather than duplicate these steps for each scenario, Rails Composer gives us session helpers to dry up the code.

A similar session helper is needed for the sign-in and sign-out feature tests. For each of these scenarios, we must simulate the user visiting the sign-in form, filling in the email and password fields, and clicking the "Sign in" button.

Rails Composer installs the session helpers in the file spec/support/helpers/session_helpers.rb:

```
module Features
  module SessionHelpers
    def sign_up_with(email, password, confirmation)
      visit new_user_registration_path
      fill_in 'Email', with: email
      fill_in 'Password', with: password
      fill_in 'Password confirmation', :with => confirmation
      click_button 'Sign up'
    end

    def signin(email, password)
      visit new_user_session_path
      fill_in 'Email', with: email
      fill_in 'Password', with: password
      click_button 'Sign in'
    end
  end
end
```

The session helpers are Capybara actions and matchers combined in a Ruby module. Rails Composer includes the Capybara gem in the Gemfile so we can implement these feature tests.

To make the session helpers available to the feature tests, we must include them in the RSpec spec/rails_helper.rb file, or alternatively, add them to the spec/support/ folder as Rails Composer does here with a spec/support/helpers.rb file:

```
RSpec.configure do |config|
  config.include Features::SessionHelpers, type: :feature
end
```

With these two files in place, our session helpers are available to all our feature tests.

Sign Up

Let's examine one of the test scenarios for the sign-up feature.

In the file "spec/features/visitors/sign_up_spec.rb" you'll find this scenario:

```
# Scenario: Visitor can sign up with valid email address and password
#   Given I am not signed in
#   When I sign up with a valid email address and password
#   Then I see a successful sign up message
scenario 'visitor can sign up with valid email address and password' do
  sign_up_with('test@example.com', 'please123', 'please123')
  expect(page).to have_content 'Welcome! You have signed up successfully.'
end
```

You'll see the use of the `sign_up_with` session helper. By supplying parameters for email, password, and password confirmation, we can easily test for expected errors when the user enters invalid credentials. Here's another scenario from the same file:

```
# Scenario: Visitor cannot sign up with mismatched password and confirmation
#   Given I am not signed in
#   When I sign up with a mismatched password confirmation
#   Then I should see a mismatched password message
scenario 'visitor cannot sign up with mismatched password and confirmation' do
  sign_up_with('test@example.com', 'please123', 'mismatch')
  expect(page).to have_content "Password confirmation doesn't match"
end
```

The tests are very simple but verify that our sign-up feature works as intended. Our tests show that we have integrated Devise correctly in our application.

Sign In

The file spec/features/users/sign_in_spec.rb contains four scenarios for the sign-in feature.

Here's the test of a successful sign in:

```
# Scenario: User can sign in with valid credentials
#   Given I exist as a user
#   And I am not signed in
#   When I sign in with valid credentials
#   Then I see a success message
scenario 'user can sign in with valid credentials' do
  user = FactoryGirl.create(:user)
  signin(user.email, user.password)
  expect(page).to have_content 'Signed in successfully.'
end
```

You'll see the use of the `signin` session helper which takes the email address and password as parameters. FactoryGirl creates a user. We use the FactoryGirl `create` method. FactoryGirl offers a faster `build` method, but the `build` method doesn't save the user in the database. We cannot use the `build` method for a test of the sign-in feature because the user must be saved as a record in the database before we attempt to sign in with the user's email address and password.

Here is another scenario from the same file:

```
# Scenario: User cannot sign in with wrong email
#   Given I exist as a user
#   And I am not signed in
#   When I sign in with a wrong email
#   Then I see an invalid email message
scenario 'user cannot sign in with wrong email' do
  user = FactoryGirl.create(:user)
  signin('invalid@email.com', user.password)
  expect(page).to have_content 'Invalid email or password.'
end
```

In this case, we intentionally pass an email address that is different from the email address stored in the database. The test affirms that a user sees an "Invalid email or password" message when they use an email address that is unknown to the application.

You can examine the test of the sign-out feature in the file spec/features/users/sign_out_spec.rb. It is similar to the tests above.

Shortcutting Sign In

We've used session helpers to automate the repetitive steps of sign up and sign in. To test our integration of Devise, there is no substitute for actually stepping through the sign-up and sign-in processes. However, for a speedy test suite, we don't want to actually step through the sign-in process for every feature test we write. We only need a dozen (or so) tests to make sure the sign-up and sign-in processes work correctly. But we may have hundreds of feature tests in a large application that require the user to sign in during the setup phase of the test. After all, if you are testing a user's interaction with any feature, the user must be signed in.

It is time-consuming for every test to actually log in before testing a feature. That's why most feature tests will use a shortcut suggested by the Devise developers. Our test of the list of users is a good example. We know from our previous tests that our integration with Devise works correctly. Now we simply need an authenticated user to test a feature that is independent of Devise.

Let's examine the file spec/features/users/user_index_spec.rb:

```
include Warden::Test::Helpers
Warden.test_mode!

# Feature: User index page
#   As a user
#   I want to see a list of users
#   So I can see who has registered
feature 'User index page', :devise do

  after(:each) do
    Warden.test_reset!
  end

  # Scenario: User listed on index page
  #   Given I am signed in
  #   When I visit the user index page
  #   Then I see my own email address
  scenario 'user sees own email address' do
    user = FactoryGirl.create(:user)
    login_as(user, scope: :user)
    visit users_path
    expect(page).to have_content user.email
  end

end
```

This test uses a shortcut to obtain an authenticated user. The Devise wiki gives details about How To Test with Capybara (https://github.com/plataformatec/devise/wiki/How-To:-Test-with-Capybara). Warden (https://github.com/hassox/warden/wiki) provides the underlying authentication mechanism that is used by Devise. Warden::Test::Helpers (https://github.com/hassox/warden/wiki/Testing) quickly produce the end result, an authenticated user, that is produced by the Devise sign-in process. Using Warden::Test::Helpers speeds up testing by replacing the slow Capybara simulation of the Devise sign-in process.

To use Warden::Test::Helpers, include them at the top of the file and set Warden in test mode:

```
include Warden::Test::Helpers
Warden.test_mode!
```

Then you can use the Warden `login_as` method in the test scenario:

```
include Warden::Test::Helpers
Warden.test_mode!

  after(:each) do
    Warden.test_reset!
  end

  scenario 'user sees own email address' do
    user = FactoryGirl.create(:user)
    login_as(user, scope: :user)
    visit users_path
    expect(page).to have_content user.email
  end

end
```

For any feature test that requires an authenticated user, use the Warden `login_as` method. If you use it in many tests, you can dry up your code by moving these two steps to a helper:

```
user = FactoryGirl.create(:user)
login_as(user, scope: :user)
```

Note that you must reset Warden test mode after each test has run:

```
after(:each) do
  Warden.test_reset!
end
```

The test of the “User edit” feature in the file spec/features/users/user_edit_spec.rb is very similar to the test above, so you can examine it on your own.

User Show

The test of the “User profile page” contains one oddity that is worth examining.

Here is the file spec/features/users/user_show_spec.rb:

```
# Scenario: User cannot see another user's profile
#   Given I am signed in
#   When I visit another user's profile
#   Then I see an 'access denied' message
scenario "user cannot see another user's profile" do
  me = FactoryGirl.create(:user)
  other = FactoryGirl.create(:user, email: 'other@example.com')
  login_as(me, :scope => :user)
  Capybara.current_session.driver.header 'Referer', root_path
  visit user_path(other)
  expect(page).to have_content 'Access denied.'
end
```

The Users controller contains this code:

```
def show
  @user = User.find(params[:id])
  unless @user == current_user
    redirect_to :back, :alert => "Access denied."
  end
end
```

In production or development, a referrer is available so the user can be redirected to the previous page if access is denied. In test mode, using the Warden::Test::Helpers, no referrer is set, so we must use a Capybara trick to set the referrer for the test to run. This line of code sets a referrer before attempting to visit the prohibited page:

```
Capybara.current_session.driver.header 'Referer', root_path
```

This little trick can be used for any test where a controller redirects to a previous page with `:back`.

Don't be confused by the misspelling of referrer; in code, you'll often see "Referer" (with three R's) instead of the correct English-language spelling "referrer" (with four R's). "Referer" is a misspelling that was set in stone in 1996 (http://en.wikipedia.org/wiki/HTTP_referer) by the standards document RFC 1945.

User Delete

For a simple application, you'd think testing would be simple. It seems that is not always the case. Consider the "User delete" feature. You may recall it uses the standard Rails delete feature (http://guides.rubyonrails.org/form_helpers.html#how-do-forms-with-patch-put-or-delete-methods-work-questionmark), which uses jQuery to pop up a dialog box for confirmation before submitting the request to the Registrations controller. That means you can't test it without invoking JavaScript in the browser. Luckily, Rails Composer sets up Capybara with the Selenium driver that uses Firefox to run actual browser-based test sessions. The RSpec Quickstart Guide (https://tutorials.railsapps.org/tutorials/rspec-quickstart) provides the details.

In the file spec/features/users/user_delete_spec.rb, you'll see the use of the Capybara JavaScript option.

```
include Warden::Test::Helpers
Warden.test_mode!

# Feature: User delete
#   As a user
#   I want to delete my user profile
#   So I can close my account
feature 'User delete', :devise, :js do

  after(:each) do
    Warden.test_reset!
  end

  # Scenario: User can delete own account
  #   Given I am signed in
  #   When I delete my account
  #   Then I should see an account deleted message
  scenario 'user can delete own account' do
    skip 'skip a slow test'
    user = FactoryGirl.create(:user)
    login_as(user, :scope => :user)
    visit edit_user_registration_path(user)
    click_button 'Cancel my account'
    page.driver.browser.switch_to.alert.accept
    expect(page).to have_content 'Bye! Your account was successfully cancelled. We hope to see
you again soon.'
  end

end
```

The flag `js: true` invokes the Selenium driver and Firefox. You'll see a web browser window open (if you have installed Firefox) when the test runs.

Here's the code that forces the test to click the accept button in the JavaScript pop-up dialog box:

```
page.driver.browser.switch_to.alert.accept
```

You may not want your test suite to run the slow JavaScript-based test every time, so it includes the line:

```
skip 'skip a slow test'
```

To run the test, comment out the line that forces a skip. Firefox must be installed on your computer for this test to run without errors.

User Model

We haven't created a User model that requires extensive unit tests. As you add methods to your User model, you'll likely add unit tests. Here's the simple User model spec installed by Rails Composer in the file spec/models/user_spec.rb:

```
describe User do

  before(:each) { @user = User.new(email: 'user@example.com') }

  subject { @user }

  it { should respond_to(:email) }

  it "#email returns a string" do
    expect(@user.email).to match 'user@example.com'
  end

end
```

As you see, it only expects to find an email address. You could add a test for the user's name or any additional attributes.

RSpec Tags

Rails Composer installs the test suite with RSpec tags for all the feature tests of Devise. You can use RSpec tags to selectively run portions of the test suite.

Here's what an RSpec tag looks like:

```
feature 'Sign in', :devise do
```

Without a tag, it looks like this:

```
feature 'Sign in' do
```

If you run RSpec without a tag, every test will be executed:

```
$ rspec
```

If you only want to run Devise tests, use the command:

```
$ rspec --tag devise
```

If you only want to skip all tests tagged `:devise`, use the command:

```
$ rspec --tag ~devise
```

RSpec tags are a convenient way to save time and narrow your focus when testing.

Customizing the Application

This guide describes the most common use of Devise. Most websites that require authentication will ask a user to register with an email address and immediately provide access on sign up. This guide shows how to add a custom attribute to the sign up form, which is a common requirement not provided by a default Devise installation. Though most websites need nothing more than this, there are many websites that need something different from the default Devise installation.

For example, you might wish to confirm an email address before allowing access to a new user (use the Devise Confirmable module). Or you might wish to invite new users, setting up an account, sending an invitation, and letting a new user set a password when they respond to an invitation (use the devise_invitable (https://github.com/scambra/devise_invitable) gem). These variations are described in the pages of the Devise wiki (https://github.com/plataformatec/devise/wiki) and you can look on Stack Overflow (http://stackoverflow.com/questions/tagged/devise) for help.

It can be difficult to figure out how to customize Devise for a less common authentication approach. If you're having trouble tweaking Devise to your needs, first check the Devise wiki (https://github.com/plataformatec/devise/wiki) and Stack Overflow (http://stackoverflow.com/questions/tagged/devise) for help. If you can't find what you need, leave a note in the comments at the end of this guide. I can't provide code for every custom requirement, but I'll try to address some of the more common variations here.

Replace Email With Account Number

With Devise, the convention is to use an email address for sign up and sign in. Everyone has an email address (or can easily get one), every email address is unique, and an email address can be used to send messages to users. But you may not want to use an email address to identify your users.

Let's look at how to change Devise to use an account number instead of an email address for sign up and sign in. For example, you might have a business application that lets existing customers sign in using an account number.

To make this customization, it's not enough to search for "email" and replace it with "account_number" throughout the application. With Devise, not all code that relies on an email attribute is visible; some is buried in the gem itself. Devise can accommodate the change, though the necessary tweaks are not clearly documented.

First, look for the database migration that Devise created. In the folder db/migrate/ there is a filename that begins with a unique number and ends in …_devise_create_users.rb. Change the string `email` to `account_number`. The string appears twice, setting an attribute and creating an index.

After you change the migration, reset the database and apply the new migration:

```
$ rake db:migrate:reset
```

Change the file app/models/user.rb:

```
class User < ActiveRecord::Base
  # Include default devise modules. Others available are:
  # :confirmable, :lockable, :timeoutable and :omniauthable
  devise :database_authenticatable, :registerable,
         :recoverable, :rememberable, :trackable, :validatable,
         :authentication_keys => [:account_number]

  def email_required?
    false
  end

  def email_changed?
    false
  end

end
```

We've added a configuration setting `:authentication_keys => [:account_number]`. Alternatively, the default can be changed in the file config/initializers/devise.rb. I think it's wise to change the User model, as it makes the change obvious.

We also add two methods `email_required?` and `email_changed?`, overriding inherited methods and setting the values to false. These methods are hooks into the Devise gem that the Devise developers provide for disabling use of the email attribute.

Change the file app/views/devise/registrations/new.html.erb:

```
<div class="authform">
  <%= form_for(resource, :as => resource_name, :url => registration_path(resource_name), :html
=> { :role => 'form'}) do |f| %>
    <h3>Sign up</h3>
    <%= devise_error_messages! %>
    <div class="form-group">
      <%= f.label :name %>
      <%= f.text_field :name, :autofocus => true, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :account_number %>
      <%= f.text_field :account_number, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :password %>
      <%= f.password_field :password, class: 'form-control' %>
    </div>
    <div class="form-group">
      <%= f.label :password_confirmation %>
      <%= f.password_field :password_confirmation, class: 'form-control' %>
    </div>
     <%= f.submit 'Sign up', :class => 'button right' %>
  <% end %>
</div>
```

We've changed the Devise registration view to replace `email` with `account_number`.

You'll also need to replace `email` with `account_number` in the following files:

- app/views/devise/passwords/new.html.erb
- app/views/devise/registrations/edit.html.erb
- app/views/devise/sessions/new.html.erb

In our example application, we use `email` in several files:

- app/views/users/_user.html.erb
- app/views/users/show.html.erb

You'll need to make changes in those files, as well as throughout the tests.

With these changes, you'll be able to use an account number instead of an email address for sign up and sign in.

Resources for Devise

Devise may be the best-documented gem used by Rails developers. Unlike many gem developers, the Devise developers at Plataformatec (http://plataformatec.com.br/) are committed to expanding and updating the gem documentation. In addition, since Devise is used by so many developers, Stack Overflow has thousands of questions and answers about Devise.

Of course, documentation can become a problem when Rails and its gems change frequently. That's the challenge with Devise; you'll need to carefully review whether information you find on the web is up to date.

Devise Wiki

The Devise wiki (https://github.com/plataformatec/devise/wiki) is the best single source of documentation for the gem, including over a hundred How Tos (https://github.com/plataformatec/devise/wiki/How-Tos) that cover options for customization. It's worthwhile to scan the wiki to see the scope of issues and features for Devise, but some of the wiki pages have gathered cobwebs and may be out of date.

Stack Overflow

Check Stack Overflow for questions tagged 'devise' (http://stackoverflow.com/questions/tagged/devise). Every question about Devise has been asked (and often answered) on Stack Overflow. Be sure to look closely at the date of the answers, as many answers are no longer current.

Devise Mailing List

There is a Devise Google Group (https://groups.google.com/forum/#!forum/plataformatec-devise) for questions and discussion. You should check Stack Overflow (http://stackoverflow.com/questions/tagged/devise) first. If you need clarification, the experts will answer your questions on the Google Group mailing list.

Devise Changelog and Issues

Check the Devise changelog (https://github.com/plataformatec/devise/blob/master/CHANGELOG.md) to see how Devise has changed from version to version.

You can search the GitHub issues for Devise (https://github.com/plataformatec/devise/issues) for open and closed issues that may shed light on any problems you have. The GitHub issues are not a place to ask to ask "how to" questions, however.

Books

There is very little coverage of Devise in books.

Obie Fernandez's The Rails 4 Way (https://leanpub.com/tr4w) has a brief section on Devise. Michael Hartl's acclaimed Rails Tutorial (http://ruby.railstutorial.org/) doesn't cover Devise, in favor of showing how to build authentication from scratch.

Packt Publishing offers a book, Learning Devise for Rails (http://www.packtpub.com/learning-devise-for-rails/book) that is comparable in quality to other books from the same publisher.

Comments

Credits

Daniel Kehoe wrote the guide.

Did You Like This Guide?

Was the guide useful to you? Follow @rails_apps (http://twitter.com/rails_apps) on Twitter and tweet some praise. I'd love to know you were helped out by the article.

You can also find me on Facebook (https://www.facebook.com/daniel.kehoe.sf) or Google+ (https://plus.google.com/+DanielKehoe/).