RailsApps Project

Devise with CanCan ar Twitter Bootstrap

Ruby on Rails tutorial for a Rails 3.2 application using Devise with CanCan and Twitter Bootstr

Contents

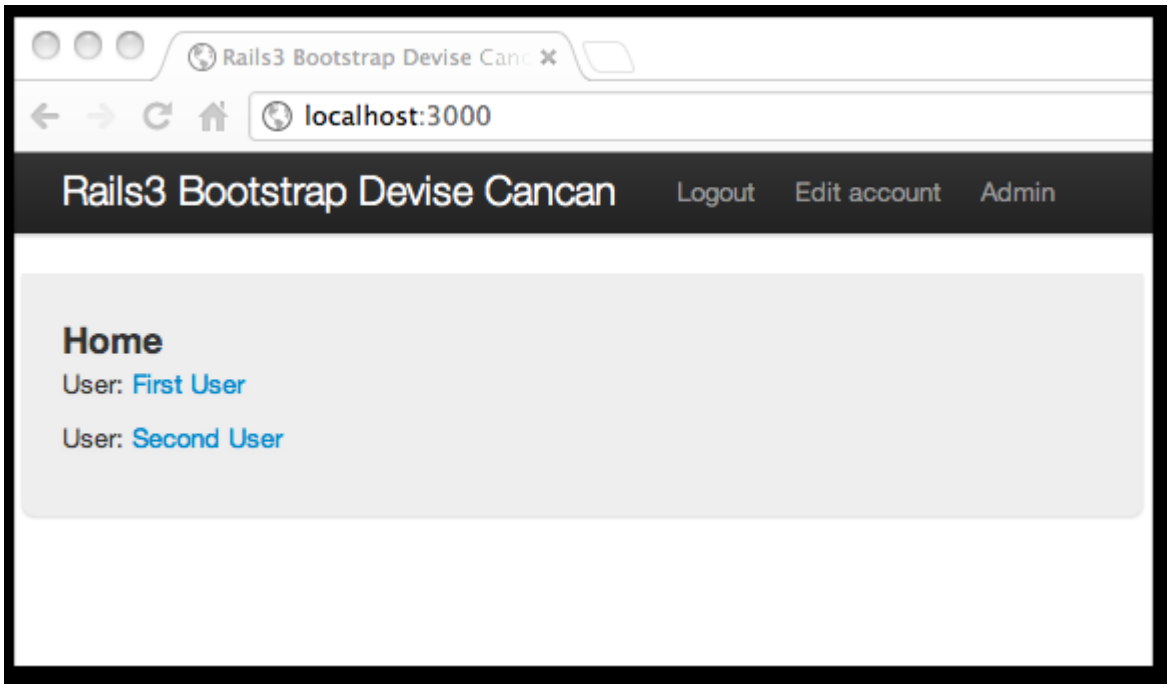
Chapter 1

Introduction

Ruby on Rails tutorial showing how to create a Rails 3.2 application using **Devise** with **CanCan** and **Twitter Bootstrap**.

- Devise gives you ready-made authentication and user management.
- CanCan provides authorization which restricts the resources a user is allowed to access.
- Twitter Bootstrap is a front-end framework for CSS styling.

Screenshot



Is It for You?

This tutorial is for experienced developers as well as startup founders or hobbyist coders who are new to Rails.

Experienced developers will find the complete application on GitHub; this tutorial provides the detail and background to understand the implementation in depth. For Rails beginners, this tutorial describes each step that you must follow to create the application. Every step is

documented concisely, so you can create this application without any additional knowledge. However, the tutorial assumes you've already been introduced to Rails, so if you are a beginner, you may be overwhelmed unless you've been introduced to Rails elsewhere. If you're new to Rails, see recommendations for a Rails tutorial and resources for getting started with Rails.

This is one in a series of Rails example apps and tutorials from the RailsApps Project. See a list of similar Rails examples, tutorials, and starter apps.

This example application is based on the rails3-devise-rspec-cucumber example and tutorial and adds CanCan and Twitter Bootstrap. View the rails3-devise-rspec-cucumber example and tutorial for the basics of setting up an application with RSpec and Cucumber for testing.

This example application uses ActiveRecord and a SQLite database. You can use the Mongoid ORM with the MongoDB datastore instead, for faster development without schemas or migrations. The rails3-mongoid-devise example app and tutorial shows how to set up Devise and Mongoid with RSpec and Cucumber.

For more complex applications that use Devise, CanCan, and Twitter Bootstrap, see:

- rails-stripe-membership-saas example and tutorial
- rails-prelaunch-signup example and tutorial

How to Support the Project

If you haven't subscribed to the RailsApps Tutorials Pro Plan, please consider purchasing a monthly subscription to support the project.

The code for the example applications is open source and unrestricted; your purchase of a subscription plan supports maintenance of the project. Rails applications need frequent updates as Rails and its gems change. Subscription sales support the project so we can keep the applications current so you'll always have an up-to-date reference implementation.

Before You Start

Most of the tutorials from the RailsApps project take about an hour to complete.

If you follow this tutorial closely, you'll have a working application that closely matches the example app in this GitHub repository. The example app in the rails3-bootstrap-devise-cancan repository is your reference implementation. If you find problems with the app you build from this tutorial, download the example app (in Git speak, clone it) and use a file compare tool to identify differences that may be causing errors. On a Mac, good file compare tools are FileMerge, DiffMerge, Kaleidoscope, or Ian Baird's Changes.

If you find problems or wish to suggest improvements, please create a GitHub issue. It's best to clone and check the example application from the GitHub repository before you report an issue, just to make sure the error isn't a result of your own mistake.

The online edition of this tutorial contains a comments section at the end of the tutorial. I encourage you to offer feedback to improve this tutorial.

Assumptions

Before beginning this tutorial, you need to install

- The Ruby language (version 1.9.3)
- Rails 3.2

Check that appropriate versions of Ruby and Rails are installed in your development environment:

```
$ ruby -v
$ rails -v
```

Be sure to read Installing Rails to make sure your development environment is set up properly.

I recommend using rvm, the Ruby Version Manager to manage your Rails versions and create a dedicated gemset for each application you build.

Chapter 2

Create the Application

You have several options for getting the code. You can *copy from the tutorial*, *fork*, *clone*, or *generate*.

Copy from the Tutorial

To create the application, you can cut and paste the code from the tutorial into your own files. It's a bit tedious and error-prone but you'll have a good opportunity to examine the code closely.

Other Options

Fork

If you'd like to add features (or bug fixes) to improve the example application, you can fork the GitHub repo and make pull requests. Your code contributions are welcome!

Clone

If you want to copy and customize the app with changes that are only useful for your own project, you can download or clone the GitHub repo. You'll need to search-and-replace the project name throughout the application. You probably should generate the app instead (see below). To clone:

```
$ git clone git://github.com/RailsApps/rails3-bootstrap-devise-cancan.git
```

You'll need git on your machine. See Rails and Git.

Generate

If you wish to skip the tutorial and build the application immediately, use the Rails Composer tool to generate the complete example app. You'll be able to give it your own project name when you generate the app. Generating the application gives you additional options.

To build the complete example application immediately, see the instructions in the README for the rails3-bootstrap-devise-cancan example application.

Building from Scratch

Beginning here, we show how to create the application from scratch.

Open a terminal, navigate to a folder where you have rights to create files, and type:

```
$ rails new rails3-bootstrap-devise-cancan -T
```

Use the `-T` flag to skip Test::Unit files (since you are using RSpec).

You may give the app a different name if you are building it for your own use. For this tutorial, we'll assume the name is "rails3-bootstrap-devise-cancan."

This will create a Rails application that uses a SQLite database for data storage.

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

```
$ cd rails3-bootstrap-devise-cancan
```

Replace the READMEs

Please edit the README files to add a description of the app and your contact info. Changing the README is important if your app will be publicly visible on GitHub. Otherwise, people will think I am the author of your app. If you like, add an acknowledgment and a link to the RailsApps project.

Set Up Source Control (Git)

If you're creating an app for deployment into production, you'll want to set up a source control repository at this point. If you are building a throw-away app for your own education, you may skip this step.

```
$ git init .
$ git add -A
$ git commit -m 'Initial commit'
```

See detailed instructions for Using Git with Rails.

If you want to store your application on GitHub, you should now set up your GitHub repository.

Set Up Gems

It's a good idea to create a new gemset using rvm, the Ruby Version Manager, as described in the article Installing Rails.

Create a gemset:

```
$ rvm use ruby-2.0.0@rails3-bootstrap-devise-cancan --create
```

Open your **Gemfile** and replace the contents with the following:

Gemfile

```
source 'https://rubygems.org'
gem 'rails', '3.2.13'
gem 'sqlite3'
group :assets do
  gem 'sass-rails',   '~> 3.2.3'
  gem 'coffee-rails', '~> 3.2.1'
  gem 'uglifier', '>= 1.0.3'
end
gem 'jquery-rails'
gem "rspec-rails", ">= 2.11.4", :group => [:development, :test]
gem "database_cleaner", ">= 0.9.1", :group => :test
gem "email_spec", ">= 1.4.0", :group => :test
gem "cucumber-rails", ">= 1.3.0", :group => :test, :require => false
gem "launchy", ">= 2.1.2", :group => :test
gem "capybara", ">= 2.0.1", :group => :test
gem "factory_girl_rails", ">= 4.1.0", :group => [:development, :test]
gem "bootstrap-sass", ">= 2.1.1.0"
gem "devise", ">= 2.1.2"
gem "cancan", ">= 1.6.8"
gem "rolify", ">= 3.2.0"
gem "simple_form", ">= 2.0.4"
gem "quiet_assets", ">= 1.0.1", :group => :development
gem "figaro", ">= 0.5.0"
gem "better_errors", ">= 0.2.0", :group => :development
gem "binding_of_caller", ">= 0.6.8", :group => :development
```

This tutorial requires Rails version 3.2.13.

Note: The RailsApps examples are generated with application templates created by the Rails Apps Composer Gem. For that reason, groups such as `:development` or `:test` are specified

inline. You can reformat the Gemfiles to organize groups in an eye-pleasing block style. The functionality is the same.

For more information about the Gemfile, see Gemfiles for Rails 3.2.

Install the Required Gems

Install the required gems on your computer:

```
$ bundle install
```

You can check which gems are installed on your computer with:

```
$ gem list
```

Keep in mind that you have installed these gems locally. When you deploy the app to another server, the same gems (and versions) must be available.

Test the App

You can check that your app runs properly by entering the command

```
$ rails server
```

To see your application in action, open a browser window and navigate to http://localhost:3000/. You should see the Rails default information page.

Stop the server with Control-C.

Chapter 3

Test-Driven Development

This example application uses Cucumber for integration testing and RSpec for unit testing.

Testing is at the center of any robust software development process. Integration tests determine whether the application's features work as expected, testing the application from the point of view of the user. Unit tests confirm that small, discrete portions of the application continue working as developers add features and refactor code. RSpec is a popular choice for unit testing. The rails3-devise-rspec-cucumber tutorial shows how to set up RSpec and provides example specs for use with Devise. Cucumber is a popular choice for integration testing and behavior driven development. The rails3-devise-rspec-cucumber tutorial shows how to set up Cucumber and provides example scenarios for use with Devise.

This tutorial assumes you've learned to write tests elsewhere (see a list of recommended resources for Rails). I won't spend time showing you how to write tests but you can use tests to make sure the application works as expected.

If you want to build the application with RSpec and Cucumber test frameworks, here's how to set up RSpec and Cucumber.

Use the rspec-rails generator to set up files needed for RSpec:

```
$ rails generate rspec:install
```

Use the cucumber-rails generator to set up files needed for Cucumber:

```
$ rails generate cucumber:install --capybara --rspec
```

You can copy the RSpec unit tests and Cucumber integration tests from the rails3-bootstrap-devise-cancan example application. Replace both the **spec** and **features** directories entirely. Copying all the files will include necessary configuration and helper files.

Run `rake -T` to check that rake tasks for RSpec and Cucumber are available. You won't be able to run `rake spec` or `rake cucumber` until the database is set up.

Chapter 4

Configuration

Configuration File

The application uses the figaro gem to set environment variables. See the article Rails Environment Variables for more information.

The figaro gem uses a generator to set up the necessary files. Run:

```
$ rails generate figaro:install
```

This generates a **config/application.yml** file and lists it in your **.gitignore** file.

Credentials for your administrator account and email account are set in the **config/application.yml** file. The **.gitignore** file prevents the **config/application.yml** file from being saved in the git repository so your credentials are kept private.

Modify the file **config/application.yml**:

```
# Add account credentials and API keys here.
# See http://railsapps.github.io/rails-environment-variables.html
# This file should be listed in .gitignore to keep your settings secret!
# Each entry sets a local environment variable and overrides ENV variables in the Unix
shell.
# For example, setting:
# GMAIL_USERNAME: Your_Gmail_Username
# makes 'Your_Gmail_Username' available as ENV["GMAIL_USERNAME"]
# Add application configuration variables here, as shown below.
#
GMAIL_USERNAME: Your_Username
GMAIL_PASSWORD: Your_Password
ADMIN_NAME: First User
ADMIN_EMAIL: user@example.com
ADMIN_PASSWORD: changeme
ROLES: [admin, user, VIP]
```

Set the user name and password needed for the application to send email.

If you wish, set your name, email address, and password for an administrator's account. If you prefer, you can use the default to sign in to the application and edit the account after

deployment. It is always a good idea to change the administrator's password after the application is deployed.

Specify roles in the configuration file. You will need an "admin" role. Change the "user" and "VIP" roles as you wish.

All configuration values in the **config/application.yml** file are available anywhere in the application as environment variables. For example, `ENV["GMAIL_USERNAME"]` will return the string "Your_Username".

If you prefer, you can delete the **config/application.yml** file and set each value as an environment variable in the Unix shell.

Configure Email

This example application doesn't send email messages. However, if you want your application to send email messages (for example, if you plan to install the Devise `:confirmable` module) you must configure the application for your email account. See the article Send Email with Rails.

Chapter 5

Layout and Stylesheets

Rails will use the layout defined in the file **app/views/layouts/application.html.erb** as a default for rendering any page.

You'll want to add navigation links, include flash messages for errors and notifications, and apply CSS styling.

This tutorial shows code using ERB, the default Rails templating language. If you prefer to use Haml, see the detailed guide Rails Default Application Layout for HTML5.

Navigation Links

You'll likely need navigation links on every page of your web application. You'll want a link for Home. You'll want links for Login, Logout, and Sign Up. And a user who is an administrator should see a link for Admin.

You can add navigation links directly to your application layout file but many developers prefer to create a partial template – a "partial" – to better organize the default application layout.

Create the file **app/views/layouts/_navigation.html.erb** for the navigation links:

```
<%= link_to "Rails3 Bootstrap Devise Cancan", root_path, :class => 'brand' %>
<ul class="nav">
  <% if user_signed_in? %>
    <li>
    <%= link_to('Logout', destroy_user_session_path, :method=>'delete') %>
    </li>
  <% else %>
    <li>
    <%= link_to('Login', new_user_session_path)  %>
    </li>
  <% end %>
  <% if user_signed_in? %>
    <li>
    <%= link_to('Edit account', edit_user_registration_path) %>
    </li>
    <% if current_user.has_role? :admin %>
      <li>
      <%= link_to('Admin', users_path) %>
      </li>
    <% end %>
  <% else %>
    <li>
    <%= link_to('Sign up', new_user_registration_path)  %>
    </li>
  <% end %>
</ul>
```

Notice the condition `<% if current_user.has_role? :admin %>` that uses a `has_role?` method provided by the Rolify gem. The Admin link will display only if the user is an administrator.

Flash Messages

Rails provides a standard convention to display alerts (including error messages) and other notices (including success messages), called Rails "flash messages" (as in "flash memory", not to be confused with the "Adobe Flash" proprietary web development platform).

You can include code to display flash messages directly in your application layout file or you can create a partial.

Create a partial for flash messages in **app/views/layouts/_messages.html.erb** like this:

```
<% flash.each do |name, msg| %>
  <% if msg.is_a?(String) %>
    <div class="alert alert-<%= name == :notice ? "success" : "error" %>">
      <a class="close" data-dismiss="alert">&#215;</a>
      <%= content_tag :div, msg, :id => "flash_#{name}" %>
    </div>
  <% end %>
<% end %>
```

Rails uses `:notice` and `:alert` as flash message keys. Twitter Bootstrap provides a base class `.alert` with additional classes `.alert-success` and `.alert-error` (see the Bootstrap documentation on alerts). A bit of parsing is required to get a Rails "notice" message to be styled with the Twitter Bootstrap "alert-success" style. Any other message, including a Rails "alert" message, will be styled with the Twitter Bootstrap "alert-error" style.

Twitter Bootstrap provides a jQuery plugin named bootstrap-alert that makes it easy to dismiss flash messages with a click. The data-dismiss property displays an "x" that enables the close functionality. Note that Twitter Bootstrap uses the HTML entity "×" instead of the keyboard letter "x".

By default, Twitter Bootstrap applies a green background to `.alert-success` and a red background to `.alert-error`. Twitter Bootstrap provides a third class `.alert-info` with a blue background. With a little hacking, it's possible to create a Rails flash message with a custom name, such as `:info`, that will display with the Bootstrap `.alert-info` class. However, it's wise to stick with the Rails convention of using only "alert" and "notice."

CSS Styling with SASS

It's a good idea to rename the **app/assets/stylesheets/application.css** file as **app/assets/stylesheets/application.css.scss**.

This will allow you to use the advantages of the SASS syntax and features for your application stylesheet. For more on the advantages of SASS and how to use it, see the SASS Basics RailsCast from Ryan Bates.

CSS Styling with Twitter Bootstrap

Twitter Bootstrap and other CSS front-end frameworks (such as Zurb Foundation) are toolkits that provide the kind of structure and convention that make Rails popular for server-side ("back-end") development. You can use Twitter Bootstrap to quickly add attractive CSS styling to your application.

Several options are available for installing Twitter Bootstrap in a Rails application. Twitter Bootstrap can be installed using either its native LESS CSS language or the SASS language

that is the default for CSS files in Rails. See the article Twitter Bootstrap, Less, and Sass: Understanding Your Options for Rails 3.1. SASS is a default for CSS development in Rails so I recommend you install Thomas McDonald's bootstrap-sass gem.

In your **Gemfile**, you've already added:

```
gem 'bootstrap-sass'
```

and previously run `$ bundle install`.

Include the Twitter Bootstrap Javascript files by modifying the file **app/assets/javascripts/application.js**:

```
//= require jquery
//= require jquery_ujs
//= require bootstrap
//= require_tree .
```

Next, import the Twitter Bootstrap CSS files. You can modify the **app/assets/stylesheets/application.css.scss** file to import Bootstrap. However, I recommend adding a new file **app/assets/stylesheets/bootstrap_and_overrides.css.scss** file. You may wish to modify the Twitter Bootstrap CSS rules; you can do so in the **application.css.scss** file but placing changes to Twitter Bootstrap CSS rules in the **bootstrap_and_overrides.css.scss** file will keep your CSS better organized.

The file **app/assets/stylesheets/bootstrap_and_overrides.css.scss** is automatically included and compiled into your Rails application.css file by the `*= require_tree .` statement in the **app/assets/stylesheets/application.css.scss** file.

Add the file **app/assets/stylesheets/bootstrap_and_overrides.css.scss**:

```
@import "bootstrap";
body { padding-top: 60px; }
@import "bootstrap-responsive";
```

The file will import both basic Bootstrap CSS rules and the Bootstrap rules for responsive design (allowing the layout to resize for various devices and secreen sizes).

The CSS rule `body { padding-top: 60px; }` applies an additional CSS rule to accommodate the "Bootstrap navigation bar" heading created by the `navbar-fixed-top class` in the `header` tag in the layout below.

Finally, to provide an example of adding a CSS rule that will be used on every page of your application, the following code creates a nice gray box as a background to page content.

Add this to your **app/assets/stylesheets/application.css.scss** file for a gray background:

```
.content {
  background-color: #eee;
  padding: 20px;
  margin: 0 -20px; /* negative indent the amount of the padding to maintain the grid system */
  -webkit-border-radius: 0 0 6px 6px;
  -moz-border-radius: 0 0 6px 6px;
  border-radius: 0 0 6px 6px;
  -webkit-box-shadow: 0 1px 2px rgba(0,0,0,.15);
  -moz-box-shadow: 0 1px 2px rgba(0,0,0,.15);
  box-shadow: 0 1px 2px rgba(0,0,0,.15);
}
```

Default Application Layout with Twitter Bootstrap

Generating a new Rails application with the `rails new` command will create a default application layout in the file **app/views/layouts/application.html.erb**. Modify the file to add navigation links, include flash messages, and apply CSS styling. Twitter Bootstrap provides additional elements for a more complex page layout.

Use the code below to incorporate recommendations from the article HTML5 Boilerplate for Rails Developers.

Replace the contents of the file **app/views/layouts/application.html.erb** with this:

```
<!DOCTYPE html>
<html>
  <head>
    <meta name="viewport" content="width=device-width, initial-scale=1.0">
    <title><%= content_for?(:title) ? yield(:title) : "App_Name" %></title>
    <meta name="description" content="<%= content_for?(:description) ?
yield(:description) : "App_Name" %>">
    <%= stylesheet_link_tag "application", :media => "all" %>
    <%= javascript_include_tag "application" %>
    <%= csrf_meta_tags %>
    <%= yield(:head) %>
  </head>
  <body>
    <header class="navbar navbar-fixed-top">
      <nav class="navbar-inner">
        <div class="container">
          <%= render 'layouts/navigation' %>
        </div>
      </nav>
    </header>
    <div id="main" role="main">
      <div class="container">
        <div class="content">
           <div class="row">
            <div class="span12">
              <%= render 'layouts/messages' %>
              <%= yield %>
            </div>
          </div>
          <footer>
          </footer>
        </div>
      </div> <!--! end of .container -->
    </div> <!--! end of #main -->
  </body>
</html>
```

See the article Rails Default Application Layout for an explanation of each of the elements in the application layout.

Your default application layout defines the look-and-feel of your application. You now have the basics with navigation links, messages for alerts and notices, and CSS styling using Twitter Bootstrap.

Set up SimpleForm with Twitter Bootstrap

We want to use the styling that Twitter Bootstrap provides for forms. The easiest way to integrate Twitter Bootstrap styles with our forms is to use the SimpleForm gem. There's a great SimpleForm and Twitter Bootstrap demo app that shows what can be accomplished.

You should have the following gem in your **Gemfile**:

```
gem 'simple_form'
```

If you haven't already, run:

```
$ bundle install
```

Run the generator to install SimpleForm with a Twitter Bootstrap option:

```
$ rails generate simple_form:install --bootstrap
```

which installs several configuration files:

```
config/initializers/simple_form.rb
config/initializers/simple_form_bootstrap.rb
config/locales/simple_form.en.yml
lib/templates/erb/scaffold/_form.html.erb
```

Later we'll add forms that use the SimpleForm view helpers.

Base Errors Helper for SimpleForm

SimpleForm provides excellent handling of validation errors for any attributes of a model. However, some applications set model errors that are not matched to a specific form field. These are called "base errors." For example, a model might set an error like this: `errors.add :base, "Credit card declined"`. SimpleForm does not provide a helper to display base errors; if your application sets base errors, you'll need to add the following view helper.

Add this view helper to **app/helpers/application_helper.rb**:

```
module ApplicationHelper

  def display_base_errors resource
    return '' if (resource.errors.empty?) or (resource.errors[:base].empty?)
    messages = resource.errors[:base].map { |msg| content_tag(:p, msg) }.join
    html = <<-HTML
    <div class="alert alert-error alert-block">
      <button type="button" class="close" data-dismiss="alert">&#215;</button>
      #{messages}
    </div>
    HTML
    html.html_safe
  end

end
```

Use the view helper in a form like this:

```
<%= simple_form_for ... %>
  <%= f.error_notification %>
  <%= display_base_errors resource %>
.
.
.
<% end %>
```

This extra helper is not needed in most applications; however, it can save time-consuming debugging for applications that use the `errors.add :base` method.

Chapter 6

Authentication

This app uses Devise for user management and authentication.

Set Up Configuration for Devise

You should have the following gem in your **Gemfile**:

```
gem 'devise'
```

If you haven't already, run:

```
$ bundle install
```

Run the generator to install Devise:

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

which installs a configuration file **config/initializers/devise.rb** and a localization file **config/locales/devise.en.yml**.

Configure Devise for Email

If you will be using the Devise Confirmable module to send confirmation emails, complete your email configuration by modifying the **config/initializers/devise.rb** file and setting the `config.mailer_sender` option for the return email address for messages that Devise sends from the application.

Generate a Model and Routes for Users

Use Devise to generate a model and routes for a User.

```
$ rails generate devise User
```

Devise will create a database migration and a User model.

Devise will try to create a spec file for the User model. If you've already downloaded the example app spec files, don't let the Devise generator overwrite the **spec/models/user_spec.rb** file.

Devise will try to create factories for testing the User model. If you've already downloaded the **spec/factories/users.rb** file, don't let the Devise generator overwrite the **spec/factories/users.rb** file (enter "n" to prevent overwriting).

Devise will modify the **config/routes.rb** file to add:

```
devise_for :users
```

which provides a complete set of routes for user signup and login. If you run `rake routes` you can see the routes that this line of code creates.

Accommodate Cucumber Testing for "Sign Out"

By default, Devise uses an http DELETE request for sign out requests (`destroy_user_session_path`). Rails uses Javascript to implement http DELETE requests. Prior to Devise 1.4.1 (27 June 2011), Devise used an http GET request for sign out requests. Jose Valim explained the change: "GET requests should not change the state of the server. When sign out is a GET request, CSRF can be used to sign you out automatically and things that preload links can eventually sign you out by mistake as well."

However, Cucumber wants to test GET requests not DELETE requests. If you intend to use Cucumber with Devise, you must change the Devise default from DELETE to GET in **/config/initializers/devise.rb** for the Rails test environment. You may see a suggestion elsewhere to tweak the routes.rb file or change the log_out link to make the fix. It isn't necessary if you change the **/config/initializers/devise.rb** file.

```
# The default HTTP method used to sign out a resource. Default is :delete.
config.sign_out_via = Rails.env.test? ? :get : :delete
```

Since you only use Cucumber during testing, switching the default is only needed for testing.

If you're not going to use Cucumber, leave Devise's default (DELETE) in place.

Prevent Logging of Passwords

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

Modify the file **config/application.rb** to include:

```
config.filter_parameters += [:password, :password_confirmation]
```

Note that filter_parameters is an array.

Improve the Sign In Form

We want the form for signing in (login) to use the Twitter Bootstrap form styles. Devise provides this form (hidden in the Devise gem) but we'll override the Devise default form with our own form that uses the SimpleForm view helpers that integrate with Twitter Bootstrap.

Create a new **app/views/devise/sessions/new.html.erb** file:

```
<h2>Sign in</h2>
<%= simple_form_for(resource, :as => resource_name, :url => session_path(resource_name),
:html => {:class => 'form-vertical' }) do |f| %>
  <%= f.input :email, :autofocus => true %>
  <%= f.input :password %>
  <%= f.input :remember_me, :as => :boolean if devise_mapping.rememberable? %>
  <%= f.button :submit, "Sign in", :class => 'btn-primary' %>
<% end %>
<%= render "devise/shared/links" %>
```

The new form uses Twitter Bootstrap styles.

Chapter 7

Authorization

Devise provides authentication, a system to securely identify users, making sure the user is really who he represents himself to be. We need to add a system for authorization to determine if an authenticated user should have access to secured resources. This app uses CanCan for authorization, to restrict access to pages that should only be viewed by an administrator. Cancan is by far the most popular gem used to implement authorization (see the Rails Authorization category on The Ruby Toolbox site) . The author of CanCan, Ryan Bates, offers a RailsCast on Authorization with CanCan to show how to use it.

There are many ways to implement authorization in a web application. CanCan offers an architecture that centralizes all authorization rules (permissions or "abilities") in a single location, the CanCan `Ability` class. For a discussion of the benefits of using a single, consolidated location for the permissions, see the article Don't Do Role-Based Authorization Checks; Do Activity-Based Checks.

CanCan provides a mechanism for limiting access at the level of controller and controller method and expects you to set permissions based on user attributes you define. CanCan doesn't provide default user attributes such as user roles; you must implement this outside of CanCan. There are many ways to implement role-based authorization for use with CanCan. For this example, we use Florent Monbillard's Rolify gem to create a Role model, add methods to a User model, and generate a migration for a roles table.

Set Up CanCan

CanCan provides a Rails generator to create the CanCan `Ability` class. Run the generator to create the file **app/models/ability.rb**:

```
$ rails generate cancan:ability
```

Edit the file **app/models/ability.rb** to define a simple rule for granting permission to an administrator to access any page:

```
class Ability
  include CanCan::Ability

  def initialize(user)
    user ||= User.new # guest user (not logged in)
    if user.has_role? :admin
      can :manage, :all
    end
  end

end
```

Rules defined in the `Ability` class can become quite complex. See the CanCan wiki Defining Abilities for details.

Note that the `user.has_role?` method doesn't yet exist. We'll add the method when we set up Rolify.

Configure CanCan Exception Handling

If user authorization fails, a `CanCan::AccessDenied` exception will be raised. See the CanCan wiki Exception Handling for ways to handle authorization exceptions.

For this example, we'll handle the `CanCan::AccessDenied` exception in the ApplicationController. We'll set an error message and redirect to the home page. Modify the file **app/controllers/application_controller.rb** like this:

```
class ApplicationController < ActionController::Base
  protect_from_forgery
  rescue_from CanCan::AccessDenied do |exception|
    redirect_to root_path, :alert => exception.message
  end
end
```

Set Up User Roles

We'll use the Rolify gem to implement user roles. Rolify provides a Rails generator to create a Role model, add methods to a User model, and generate a migration for a roles table. Run the command:

```
$ rails generate rolify:role
```

If you're using Mongoid with the MongoDB datastore, add parameters to the command:

```
$ rails generate rolify:role Role User mongoid
```

The generator will insert a `rolify` method in **app/models/users.rb** and it will create several files:

- **app/models/role.rb**
- **config/initializers/rolify.rb**
- **db/migrate/…rolify_create_roles.rb**

The **app/models/role.rb** looks like this:

```
class Role < ActiveRecord::Base
  has_and_belongs_to_many :users, :join_table => :users_roles
  belongs_to :resource, :polymorphic => true

  scopify
end
```

These few steps with CanCan and Rolify implement role-based authorization in our application. We'll use the authorization options provided by CanCan and Rolify when we add an administrative page with corresponding links.

Chapter 8

User Management

By default, Devise uses an email address to identify users. We'll add a "name" attribute as well. Your application may not require a user to provide a name. But showing you how to add a name will help you see what you need to do if you decide to make changes to the default Devise user model.

Add a Migration

Devise created a migration file to establish the schema for the SQLite database with a migration file named something like **db/migrate/xxxxxx_devise_create_users.rb**. We won't modify the migration file. Instead we'll add an additional migration that adds the "name" field to the User record.

```
$ rails generate migration AddNameToUsers name:string
```

Run the migration and prepare the test database to pick up the "name" field:

```
$ rake db:migrate
$ rake db:test:prepare
```

Modify the User Model

You'll want to prevent malicious hackers from creating fake web forms that would allow changing of passwords through the mass-assignment operations of `update_attributes(attrs)` or `new(attrs)`. Devise already added this to the **models/user.rb** file:

```
attr_accessible :email, :password, :password_confirmation, :remember_me
```

but you'll need to add the "name" attribute:

```
attr_accessible :name, :email, :password, :password_confirmation, :remember_me
```

If you wish, you can modify the user model to validate the presence and uniqueness of the "name" attribute. Modify the file **app/models/user.rb** and add:

```
validates_presence_of :name
validates_uniqueness_of :name, :email, :case_sensitive => false
```

This will allow users to be created (or edited) with a name attribute. When a user is created, a name and email address must be present and must be unique (not used before). Note that Devise (by default) will check that the email address and password are not blank and that the email address is unique.

Create Custom Views for User Registration

Devise provides a controller and views for registering users. It is called the "registerable" module. The controller and views are hidden in the Devise gem so we don't need to create anything. However, because we want our users to provide a name when registering, we will create custom views for creating and editing a user. Our custom views will override the Devise gem defaults. We'll use the SimpleForm view helpers (we already installed SimpleForm in a previous step).

Create a new **app/views/devise/registrations/new.html.erb** file:

```
<h2>Sign up</h2>
<%= simple_form_for(resource, :as => resource_name, :url =>
registration_path(resource_name), :html => {:class => 'form-vertical' }) do |f| %>
  <%= f.error_notification %>
  <%= display_base_errors resource %>
  <%= f.input :name, :autofocus => true %>
  <%= f.input :email, :required => true %>
  <%= f.input :password, :required => true %>
  <%= f.input :password_confirmation, :required => true %>
  <%= f.button :submit, 'Sign up', :class => 'btn-primary' %>
<% end %>
<%= render "devise/shared/links" %>
```

Create a new **app/views/devise/registrations/edit.html.erb** file:

```
<h2>Edit <%= resource_name.to_s.humanize %></h2>
<%= simple_form_for(resource, :as => resource_name, :url =>
registration_path(resource_name), :html => { :method => :put, :class => 'form-vertical'
}) do |f| %>
  <%= f.error_notification %>
  <%= display_base_errors resource %>
  <%= f.input :name, :autofocus => true %>
  <%= f.input :email, :required => true %>
  <%= f.input :password, :autocomplete => "off", :hint => "leave it blank if you don't
want to change it", :required => false %>
  <%= f.input :password_confirmation, :required => false %>
  <%= f.input :current_password, :hint => "we need your current password to confirm your
changes", :required => true %>
  <%= f.button :submit, 'Update', :class => 'btn-primary' %>
<% end %>
<h3>Cancel my account</h3>
<p>Unhappy? <%= link_to "Cancel my account", registration_path(resource_name), :data =>
{ :confirm => "Are you sure?" }, :method => :delete %>.</p>
<%= link_to "Back", :back %>
```

We do not need to add a controller with methods to create a new user or edit or delete a user. We use the existing "registerable" module from Devise which provides a controller with methods to create, edit or delete a user.

Note that Devise's default behaviour allows any logged-in user to edit or delete his or her own record (but no one else's). When you access the edit page you are editing just your info, and not info of other users.

If you are using Haml, you can convert the ERB files using the online tool Html2Haml. You'll need to remove the **.erb** files and replace them with **app/views/devise/registrations/edit.html.haml** and **app/views/devise/registrations/new.html.haml**.

Chapter 9

Home Page

Remove the Default Home Page

Delete the default home page from your application:

```
$ rm public/index.html
```

Create a Home Controller and View

Create the first page of the application. Use the Rails generate command to create a "home" controller and a "views/home/index" page:

```
$ rails generate controller home index --no-controller-specs --skip-stylesheets --skip-javascripts
```

Specify `--no-controller-specs`. You can add the RSpec files from the example application later. We'll use `--skip-stylesheets --skip-javascripts` to avoid cluttering our application with stylesheet and JavaScript files we don't need.

If you're using the default template engine, you'll find an **erb** file with placeholder content:

app/views/home/index.html.erb

Next, set a route to your home page.

Modify the file **config/routes.rb** so it looks like this:

```
Rails3BootstrapDeviseCancan::Application.routes.draw do
  authenticated :user do
    root :to => 'home#index'
  end
  root :to => "home#index"
  devise_for :users
end
```

If you examine this code, you'll see that authenticated users (those who have an account and are logged in) will see the home/index page as the application root (or home) page. And all

other users (those who don't have an account or who are not logged in) will see the same home page. The redundancy serves a didactic purpose: If you decide you want users to see a different page when they log in, you now know exactly where to change it.

By default, Devise will redirect to the root_path after successful sign in or sign out. It is easy to change the root_path as shown in the **config/routes.rb** file. Alternatively, you can override the Devise methods `after_sign_in_path_for` and `after_sign_out_path_for` as described in the Devise wiki article How To Redirect to a Specific Page.

Test the App

You can check that your app runs properly by entering the command

```
$ rails server
```

To see your application in action, open a browser window and navigate to http://localhost:3000/. You should see your new home page.

Stop the server with Control-C.

Display Users on the Home Page

Modify the file **app/controllers/home_controller.rb** and add:

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

Modify the file **app/views/home/index.html.erb** and add:

```
<h3>Home</h3>
<% @users.each do |user| %>
  <p>User: <%= user.name %> </p>
<% end %>
```

This code is not appropriate for deployment in a real application. You likely will not want to display a list of users on the home page. However, it is convenient for our example.

Chapter 10

Initial Data

Set Up a Database Seed File

You'll want to set up default users so you can test the application.

Replace the file **db/seeds.rb** with:

```
puts 'ROLES'
YAML.load(ENV['ROLES']).each do |role|
  Role.find_or_create_by_name({ :name => role }, :without_protection => true)
  puts 'role: ' << role
end
puts 'DEFAULT USERS'
user = User.find_or_create_by_email :name => ENV['ADMIN_NAME'].dup, :email =>
ENV['ADMIN_EMAIL'].dup, :password => ENV['ADMIN_PASSWORD'].dup, :password_confirmation
=> ENV['ADMIN_PASSWORD'].dup
puts 'user: ' << user.name
user.add_role :admin
user2 = User.find_or_create_by_email :name => 'Second User', :email =>
'user2@example.com', :password => 'changeme', :password_confirmation => 'changeme'
puts 'user: ' << user2.name
user2.add_role :VIP
```

The **db/seeds.rb** file initializes the database with default values. To keep some data private, and consolidate configuration settings in a single location, we use the **config/application.yml** file to set environment variables and then use the environment variables in the **db/seeds.rb** file.

The **db/seeds.rb** file reads a list of roles from the **config/application.yml** file and adds the roles to the database. In fact, any new role can be added to the roles datatable with a statement such `user.add_role :superhero`. Setting the roles in the **db/seeds.rb** file simply makes sure each role is listed and available should a user wish to change roles. Notice we have to supply the argument `:without_protection => true` to override the mass-assignment protection configured in the Role class.

Then we add two default users. The first will be an administrator. The second will be designated a VIP.

You can change the administrator name, email, and password in this file but it is better to make the changes in the **config/application.yml** file to keep the credentials private. If you

decide to include your private password in the **db/seeds.rb** file, be sure to add the filename to your **.gitignore** file so that your password doesn't become available in your public GitHub repository.

Note that it's not necessary to personalize the **db/seeds.rb** file before you deploy your app. You can deploy the app with an example user and then use the application's "Edit Account" feature to change name, email address, and password after you log in. Use this feature to log in as an administrator and change the user name and password to your own.

Seed the Database

Initialize the database by running:

```
$ rake db:migrate
$ rake db:seed
```

You can run `$ rake db:reset` whenever you need to recreate the database:

```
$ rake db:reset
```

You should set up the database for the test environment:

```
$ rake db:test:prepare
```

If you're not using rvm, you should preface each rake command with `bundle exec`. You don't need to use `bundle exec` if you are using rvm version 1.11.0 or newer.

If the task fails with "Validation failed: Name can't be blank" you should check that the file **models/user.rb** allows the "name" attribute to be mass updated:

```
attr_accessible :name, :email, :password, :password_confirmation, :remember_me
```

Test the App

You can check that your app runs properly by entering the command

```
$ rails server
```

To see your application in action, open a browser window and navigate to http://localhost:3000/. You should see your new home page.

Stop the server with Control-C.

If you test the app by starting the web server and then leave the server running while you install new gems, you'll have to restart the server to see any changes. The same is true for changes to configuration files in the config folder. This can be confusing to new Rails developers because you can change files in the app folders without restarting the server. Stop the server each time after testing and you will avoid this issue.

Chapter 11

Administrative Page #1

Add Links to Users on the Home Page

You've already modified the file **app/controllers/home_controller.rb** to include this:

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

Now we'll add links to each user's profile.

Modify the file **app/views/home/index.html.erb** to look like this:

```
<h3>Home</h3>
<% @users.each do |user| %>
  <p>User: <%=link_to user.name, user %></p>
<% end %>
```

This code is not appropriate for deployment in a real application. You likely will not want to display a list of users on the home page. However, it is convenient for our example.

The links to the user's profile page will not yet work; in the next section we'll create a users controller, routes, and views.

Create a Users Controller

Use the Rails generate command to create a "users" controller and a "views/user/show" page. You can specify `--no-controller-specs` if you've already downloaded RSpec files for the example application.

```
$ rails generate controller users index show --no-controller-specs --skip-stylesheets
--skip-javascripts
```

Note that "users" is plural when you create the controller.

Set Up the Users Routes

The generator command has changed the file **config/routes.rb** to include:

```
get "users/index"
get "users/show"
```

Remove that and change the file **config/routes.rb** to look like this:

```
Rails3BootstrapDeviseCancan::Application.routes.draw do
  authenticated :user do
    root :to => 'home#index'
  end
  root :to => "home#index"
  devise_for :users
  resources :users
end
```

Important note: The `devise_for :users` route must be placed above `resources :users`.

Set Up the Users#Show Page

Modify the file **app/views/users/show.html.erb** and add:

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

In a typical application, this page might provide additional details about the user's account.

Set Up the Users#Index Page

For the purposes of our example, this page will be accessible only to administrators. This will be our "Administrator's Dashboard." It will display a list of all users of the application.

Modify the file **app/views/users/index.html.erb** and add:

```
<ul class="users">
  <% @users.each do |user| %>
    <li>
      <%= link_to user.name, user %> signed up <%= user.created_at.to_date %>
    </li>
  <% end %>
</ul>
```

We want to restrict access to this page, the **Users#index** page at http://localhost:3000/users. In the next section, we will set up authorization so the page is accessible only to administrators.

Chapter 12

Restrict Access

You'll want to see how CanCan is used to limit access to only the administrator.

First Authorization Example: Users Controller with CanCan "@authorize!"

Modify the file **app/controllers/users_controller.rb** to look like this:

```
class UsersController < ApplicationController
  before_filter :authenticate_user!

  def index
    authorize! :index, @user, :message => 'Not authorized as an administrator.'
    @users = User.all
  end

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

end
```

The `before_filter :authenticate_user!` uses Devise to make sure a user is logged in.

Then we add one line of code to limit access to the **Users# index** page at http://localhost:3000/users:

```
authorize! :index, @user, :message => 'Not authorized as an administrator.'
```

The CanCan `authorize!` method will check the CanCan `Ability` class definition to determine if the user has permission to execute the `index` method. We've previously set a rule in the `Ability` class definition that gives a user in an administrator role the ability to execute all methods:

```
if user.has_role? :admin
  can :manage, :all
end
```

No rule is present for other users, so by default, other users are restricted from executing the `index` method.

This approach is the most obvious way to implement authorization using CanCan. We'll use this approach throughout this tutorial.

Before we implement additional features, it is worthwhile to take a look at other options available for authorization. CanCan offers convenience methods that can eliminate superfluous controller code in some applications. The next example shows how to use CanCan convenience methods to reduce the amount of code in your controllers. We won't use this approach in this tutorial but is worth reviewing.

Second Authorization Example: Users Controller with CanCan "load_and_authorize_resource"

CanCan provides a convenience method `authorize_resource` that applies the `authorize!` method to every action in the controller. Another convenience method `load_resource` queries the database and loads the resources required by each action (for example, `users = User.all` for the `index` action). A third convenience method combines the two as `load_and_authorize_resource`.

Using `load_and_authorize_resource`, you can set up the file **app/controllers/users_controller.rb** like this:

```
class UsersController < ApplicationController
  before_filter :authenticate_user!
  load_and_authorize_resource :only => :index

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

end
```

The `index` action does not need to be declared because Rails provides it by default.

If a non-administrator tries to view the at http://localhost:3000/users he or she will be redirected to the home page (as specified by the `rescue_from CanCan::AccessDenied` method in the ApplicationController) and will see CanCan's generic exception message, "You are not authorized to access this page." You can customize the exception message in the ApplicationController, if you wish.

Some developers will like this approach; others will feel it dries up code at the expense of introducing layers of black magic. If you prefer a more explicit approach, the next example shows how to implement simple role-based authorization without using CanCan at all.

Third Authorization Example: Users Controller without CanCan

The purported benefit of using CanCan is the advantage of maintaining authorization rules in one location, the `Ability` class. This may be a matter of taste; you may prefer to confine authorization code to only the controllers that need it.

Here is an example of limiting access to the **Users#index** page using only methods provided by the Rolify gem. CanCan is not used.

```
class UsersController < ApplicationController
  before_filter :authenticate_user!
  before_filter :only_allow_admin, :only => [ :index ]

  def index
    @users = User.all
  end

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

  private

  def only_allow_admin
    redirect_to root_path, :alert => 'Not authorized as an administrator.' unless
current_user.has_role? :admin
  end

end
```

If multiple controllers will use the `only_allow_admin` method, it can be moved to the ApplicationController so all controllers will inherit the method.

In contrast to CanCan, all the authorization logic is defined in the controller. For a small application, this approach is simpler and less confusing. You may find this approach preferable to using CanCan, if your application only requires simple role-based authorization. However, for a large or complex application with multiple roles and many constrained activities, CanCan offers better separation of concerns.

Now that we've explored a variety of approaches to implementing authorization, we'll implement additional features that require authorization.

Chapter 13

Administrative Page #2

The **Users# index** page is our administrative dashboard. We're using the CanCan authorization system to restrict access to this page to only users in the "admin" role. Right now, the page lists users and shows the date each registered.

We'll change the listing of users to an HTML table, display email addresses and roles, and add buttons to change roles and delete users.

Modify the Users#index Page

Change the file **app/views/users/index.html.erb** to look like this:

```
<h3>Users</h3>
<div class="span8">
<table class="table table-condensed">
    <thead>
      <tr>
          <th>Username</th>
          <th>Email</th>
          <th>Registered</th>
          <th>Role</th>
          <th></th>
          <th></th>
      </tr>
  </thead>
  <tbody>
  <% @users.each do |user| %>
    <tr>
        <td><%= link_to user.name, user %></td>
        <td><%= user.email %></td>
        <td><%= user.created_at.to_date %></td>
        <td><%= user.roles.first.name.titleize unless user.roles.first.nil? %></td>
        <td>
          <a data-toggle="modal" href="#role-options-<%= user.id %>" class="btn
btn-mini" type="button">Change role</a>
          <%= render user %>
        </td>
        <td><%= link_to("Delete user", user_path(user), :data => { :confirm => "Are you
sure?" }, :method => :delete, :class => 'btn btn-mini') unless user == current_user
%></td>
    </tr>
  <% end %>
  </tbody>
</table>
</div>
```

We've replaced a simple listing with an HTML table and applied styles from Twitter Bootstrap.

We display the user's email address and the date he or she registered.

We also display the user's role, if one has been assigned. When we use Rolify, each user can have multiple roles. We only assign a single role in this application, so we ask for the first role associated with the user. When a role object is displayed as a string, it will be a number, so we ask for the name attribute of the role. Then we capitalize the role name with the `titleize` method.

We've added a "Change role" link that is styled as a Twitter button. The link will open a Twitter Bootstrap modal window labeled "role-options" with an appended integer that corresponds to the user id (it will look like `role-options-1` in the HTML source code). You won't see the code for the modal window and form in this template file; instead we'll use the

template partial `<%= render user %>` to incorporate an individualized "Change role" form for each listed user. We'll add the partial soon.

We've added a "Delete user" link. The link directs to the User controller and passes the user id as a parameter. Before the action completes, the user must confirm the request in an alert dialog. We include the `:method => :delete` parameter so Rails will call the controller's `destroy` method. We apply Twitter Bootstrap styles for a miniature button with the `class` argument. Finally, we don't display the "Delete user" link for the current user (we don't allow the administrator to delete self).

We'll need to add controller actions before the new buttons will work.

First we'll add a template partial that contains a "Change role" form.

Add a "Change Role" Partial

Our administrative dashboard contains a "Change role" button. We'll add a partial that contains a "Change role" form.

Create a file (don't overlook the leading underscore in the filename) **app/views/users/_user.html.erb** :

```
<div id="role-options-<%= user.id %>" class="modal" style="display: none;">
  <%= simple_form_for user, :url => user_path(user), :html => {:method => :put, :class =>
'form-horizontal' } do |f| %>
    <div class="modal-header">
      <a class="close" data-dismiss="modal">&#215;</a>
      <h3>Change Role</h3>
    </div>
    <div class="modal-body">
      <%= f.input :role_ids, :collection => Role.all, :as => :radio_buttons,
:label_method => lambda {|t| t.name.titleize}, :label => false, :item_wrapper_class =>
'inline', checked: user.role_ids.first %>
    </div>
    <div class="modal-footer">
      <%= f.submit "Change Role", :class => "btn" %>
      <a class="btn" data-dismiss="modal" href="#">Close</a>
    </div>
  <% end %>
</div>
```

We wrap this form in a div designated with the Twitter Bootstrap "modal" class. It will be displayed as an overlay when the administrator clicks the "Change role" button.

We bind the form to the User object. Binding the form to the object means the values for the form fields will be set with the attributes stored in the database. The `user_path(user)` route

helper will send the submitted form to the Users controller supplying the user id in the form parameters. The `put` method ensures that the form is processed as an HTTP PUT request. We style the form with the Twitter Bootstrap "form-horizontal" class.

The form has a section named "modal-header" that contains a link to close the modal window. We follow Twitter Bootstrap's example and use HTML entity #215 (an "x" character) for the link.

The next section is named "modal-body" and it contains some of the most complex code we'll use in this application.

We want to display a set of radio buttons that display a collection of all the subscription plans and, when selected, set a role id. A user's subscription plan is encoded as a role id. Role ids are nested attributes of a User object and can be set as `user.role_ids`. So `role_ids` is the first parameter we pass to the input field helper.

The input field helper wants a collection as the second parameter. We supply `Role.all` to provide a list of all the possible roles.

We tell the input field helper we want to display the selection field as radio buttons. The `:item_wrapper_class => 'inline'` will display the radio buttons and labels horizontally.

The simple_form `label_method` parameter allows us to set a label for each radio button. If we didn't set the `label_method`, the radio buttons would be labeled with an integer. We need to obtain the name attribute of each role and then apply the `titleize` method to display titlecase. The `label_method` parameter does not take a block but we can use a programming construct called an anonymous function (a Ruby language "lambda") to manipulate the Role instance, obtaining the name attribute and making it titlecase, before passing it to the `label_method` parameter. This is a bit of advanced Ruby magic that is particularly useful here.

Confusingly, we set the `label` parameter to false. If we didn't do this, the entire field would be automatically labelled "Roles."

Finally, the parameter `checked: user.role_ids.first` makes sure the current role is preselected when the form is displayed.

The form has a section named "modal-footer" that contains a submit button and another link to close the modal window.

With the "Change role" form in place, we can modify the User model and controller to handle the form submission.

Modify the User Model for Administrative Updates

The **models/user.rb** file contains an `attr_accessible` statement that prevents malicious hackers from creating fake web forms that could change passwords or user roles through the mass-assignment operations of `update_attributes` or `new`:

```
attr_accessible :name, :email, :password, :password_confirmation, :remember_me
```

We need to modify the User model to give an administrator the power to change a user's role. Rails gives us a mechanism to override the `attr_accessible` limits.

Modify the **models/user.rb** file:

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

  # Setup accessible (or protected) attributes for your model
  attr_accessible :role_ids, :as => :admin
  attr_accessible :name, :email, :password, :password_confirmation, :remember_me

end
```

We add the statement:

```
attr_accessible :role_ids, :as => :admin
```

This statement allows us to add a matching `:as => :admin` argument to the User's `update_attributes` method called in a controller. It means we can "special case" the role id mass-assignment restriction as necessary.

We'll use the special case when we add an `update` action to the Users controller.

Add Update and Destroy Actions to the Users Controller

We'll add `update` and `destroy` actions to the Users controller.

Modify the file **app/controllers/users_controller.rb** like this:

```
class UsersController < ApplicationController
  before_filter :authenticate_user!

  def index
    authorize! :index, @user, :message => 'Not authorized as an administrator.'
    @users = User.all
  end

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

  def update
    authorize! :update, @user, :message => 'Not authorized as an administrator.'
    @user = User.find(params[:id])
    if @user.update_attributes(params[:user], :as => :admin)
      redirect_to users_path, :notice => "User updated."
    else
      redirect_to users_path, :alert => "Unable to update user."
    end
  end

  def destroy
    authorize! :destroy, @user, :message => 'Not authorized as an administrator.'
    user = User.find(params[:id])
    unless user == current_user
      user.destroy
      redirect_to users_path, :notice => "User deleted."
    else
      redirect_to users_path, :notice => "Can't delete yourself."
    end
  end
end
```

We add an `update` method to the Users controller. The `update` action responds to the "Change role" form submission. We use the CanCan `authorize!` method to limit use to only an administrator. We find the user specified by a parameter. Then we call the user model's `update_attributes` method. We pass the parameters from the form with the special argument `:as => :admin`. This is how we let the User model know that we want to override the mass-assignment restrictions set by the `attr_accessible` statement in the User model.

We add a `destroy` method to the Users controller. We use the CanCan `authorize!` method to limit use to only an administrator. We find the user specified by a parameter. Then we call the user model's `destroy` method. The "Delete user" link on the administrative dashboard won't display for the administrator's own account but we take the precaution of making sure we are not deleting the logged-in administrator with the `unless user == current_user` condition.

Either action will result in a return to the administrative dashboard page with an appropriate confirmation message or error message.

Routes for the Update and Destroy Actions

There's no need to modify the routes file to include the new actions. The `destroy` and `update` actions are automatically generated by the `resources :users` route in the file **config/routes.rb**.

That's everything we need to give the administrator the power to delete users or change roles.

Your application is ready to customize and deploy!

Chapter 14

Deploy

Cleanup

Several unneeded files are generated in the process of creating a new Rails application.

Additionally, you may want to prevent search engines from indexing your website if you've deployed it publicly while still in development.

See instructions for cleaning up unneeded files in Rails and banning spiders.

Test the App

You can check that your app runs properly by entering the command

```
$ rails server
```

To see your application in action, open a browser window and navigate to http://localhost:3000/. You should see the default user listed on the home page. When you click on the user's name, you should be required to log in before seeing the user's detail page.

To sign in as the first user (the administrator), (unless you've changed it) use

- email: user@example.com
- password: changeme

You'll see a navigation link for Admin. Clicking the link will display a page with a list of users at http://localhost:3000/users.

As an administrator, you can change any user's role or delete a user.

To sign in as the second user, (unless you've changed it) use

- email: user2@example.com
- password: changeme

The second user will not see the Admin navigation link and will not be able to access the page at
http://localhost:3000/users.

Stop the server with Control-C.

Testing

If you've copied the RSpec unit tests and Cucumber integration tests from the rails3-bootstrap-devise-cancan example application, and set up RSpec and Cucumber, you can run `rake -T` to check that rake tasks for RSpec and Cucumber are available.

Run `rake spec` to run RSpec tests.

Run `rake cucumber` (or more simply, `cucumber`) to run Cucumber scenarios.

Deploy to Heroku

For your convenience, here is a Tutorial for Rails on Heroku. Heroku provides low cost, easily configured Rails application hosting.

Be sure to set up SSL before you make your application available in production. See the Heroku documentation on SSL.

Add this configuration parameter to the **config/application.rb** file:

```
# Heroku requires this to be false
config.assets.initialize_on_precompile=false
```

Then precompile assets, commit to git, and push to Heroku:

```
$ rake assets:precompile
$ git add -A
$ git commit -m "assets compiled for Heroku"
$ git push heroku master
```

You'll need to set the configuration values from the **config/application.yml** file as Heroku environment variables. See the article Rails Environment Variables for more information.

With the figaro gem, just run:

```
$ rake figaro:heroku
```

Alternatively, you can set Heroku environment variables directly.

Here's how to set environment variables directly on Heroku with `heroku config:add`.

```
$ heroku config:add GMAIL_USERNAME='myname@gmail.com' GMAIL_PASSWORD='secret'
$ heroku config:add 'ROLES=[admin, user, VIP]'
$ heroku config:add ADMIN_NAME='First User' ADMIN_EMAIL='user@example.com'
ADMIN_PASSWORD='changeme'
```

Complete Heroku deployment with:

```
$ heroku run rake db:migrate
$ heroku run rake db:seed
```

See the Tutorial for Rails on Heroku for details.

Chapter 15

Additional Features

You've created a fully functional web application that allows visitors to register and sign in, with special access for an administrator.

Here are other example applications that add features to this application:

Tutorial	GitHub Repo	Description
Rails Membership Site with Stripe	rails-stripe-membership-saas	Site with subscription billing using Stripe
Rails Membership Site with Recurly	rails-recurly-subscription-saas	Site with subscription billing using Recurly

Assign a Default Role

As implemented, a new user is not assigned a role. You may want to assign a default role when a user is created. Here's how to do it.

Modify the **models/user.rb** file:

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

  # Setup accessible (or protected) attributes for your model
  attr_accessible :role_ids, :as => :admin
  attr_accessible :name, :email, :password, :password_confirmation, :remember_me
  before_create :assign_role

  def assign_role
    # assign a default role if no role is assigned
    self.add_role :user if self.roles.first.nil?
  end

end
```

We add a filter that calls an `assign_role` method before the user is created:

```
before_create :assign_role
```

We add the `assign_role` method to assign a default role:

```
def assign_role
  # assign a default role if no role is assigned
  self.add_role :user if self.roles.first.nil?
end
```

Keep in mind that the Rolify gem allows a user to have multiple roles. In this application, we've only given a single role for each user. You may be surprised when you add additional users using the **db/seeds.rb** file. Each user will be created and given a default role of "user." Then, if you use `user.add_role :admin` or `user2.add_role :VIP`, the `add_role` will add an *additional* role. This will happen whenever you use the `add_role` method unless you delete previous roles with `user.role_ids = []`.

Feature Requests

If you have suggestions for additional features, please create an issue on GitHub.

Chapter 16

Comments

Credits

Daniel Kehoe implemented the application and wrote the tutorial.

Did You Like the Tutorial?

Was this useful to you? Follow rails_apps on Twitter and tweet some praise. I'd love to know you were helped out by the tutorial.

Any issues? Please create an issue on GitHub. Reporting (and patching!) issues helps everyone.

Devise with CanCan and Twitter Bootstrap