Rails-Omniauth

Last updated 08 Aug 2014

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

Rails Authentication with OmniAuth

Introduction

Welcome. This tutorial shows how to build a Rails application with authentication using OmniAuth. Versions:

- OmniAuth 1.2
- Rails 4.1

OmniAuth (https://github.com/intridea/omniauth/wiki) is a gem for authentication. Most web applications need a way for users to sign in, permitting access to some features of the application only for signed-in users. Gems such as Devise (https://github.com/plataformatec/devise) enable authentication for users who register with an email address and password. OmniAuth eliminates the extra step of registration by allowing a user to sign in using an account they already have with a service such as Facebook, Twitter, or GitHub.

What's Here

You'll find:

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

Is It for You?

This tutorial shows how to set up a starter application with OmniAuth and customize it to add a user's email address. 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'll learn how to set up an application for sign-in with a single provider such as Twitter or Facebook. This tutorial doesn't cover the more complex implementation required if you want a user to sign in using a choice of providers.

Concepts

OmniAuth (https://github.com/intridea/omniauth) gives you ready-made authentication using a service provider such as:

- Facebook
- Twitter
- GitHub
- LinkedIn
- and many more (https://github.com/intridea/omniauth)

OmniAuth is a software library that adds third-party authentication to any web application that uses the Ruby-based Rack (https://en.wikipedia.org/wiki/Rack_(web_server_interface)) web server interface, including Rails. OmniAuth is modular, providing a common API that can be used to implement various *strategies* for different providers, which are available as separate gems, such as omniauth-twitter (http://rubygems.org/gems/omniauth-twitter). Many of the OmniAuth strategies implement the OAuth (https://en.wikipedia.org/wiki/OAuth) protocol which allows a web application to receive an access token from an authorization server after a user confirms they wish to gain access.

OAuth was first developed in 2006 by engineers at Twitter who wanted users to be able to sign in to third-party applications using a Twitter account. Twitter provides user registration and authentication, checking the user has the correct password for the account. If the user is signed in to Twitter, the user can gain access to another website if the application uses OAuth to check that the user is signed in to Twitter. Technically speaking, the user is authenticated by Twitter, and Twitter authorizes access to your application.

As the developer of an application, you must decide which providers will be able to authorize access. You can provide access to people who have Twitter accounts, or Facebook accounts, or accounts with other providers. Practically speaking, that means you restrict use of your application to people who have already set up accounts with Twitter or Facebook or another provider (or people who are willing to set up an account with Twitter or Facebook or another provider to gain access to your application). Use OmniAuth for access control if your application will be used by people who have accounts with a provider. For example, if your application helps people send tweets, then use OmniAuth to allow Twitter users to sign in to your application.

When you use OmniAuth, your visitors can only register to use your site if they have an account with a provider you've selected. If they do not have a Twitter, FaceBook, or similar account, they will have to create one to use your site. If you are not targeting Twitter, FaceBook, or similar users as your audience, you will be better off using Devise to manage user registration and authentication with email addresses and passwords.

Should You Combine OmniAuth With Devise?

Devise (https://github.com/plataformatec/devise) provides authentication when a visitor registers with an email address and password. Devise can be combined with OmniAuth, using the Devise Omniauthable module (https://github.com/plataformatec/devise/wiki/OmniAuth:-Overview). When you add the Omniauthable module to Devise, you can offer a user the choice of registering with an email address and password or using a provider such as Twitter or Facebook to sign in.

At first glance, it seems ideal to offer a visitor multiple choices. In practice, it creates confusion. Users frequently forget how they initially accessed the application, and they hesitate to use the application when they are not sure how they are expected to sign in. For the best user experience, limit the visitor's choices. If your audience is Facebook users, and only Facebook users, use OmniAuth. If you need to stay in touch with users by email, for example, sending payment receipts or a newsletter, use Devise alone. You should only combine Devise and OmniAuth if there is a compelling reason to do so. Apparent convenience is not often a good reason.

The Email Problem

You don't need to ask a visitor for an email address when you build an application that allows a user to sign in using a service provider such as Twitter or Facebook. You may consider that an advantage: If a user is signed in with Twitter or Facebook, they don't need to enter an email address and password to access your application. However, the lack of an email address may be a business drawback, if you want the opportunity to stay in contact with the user by email. Some service providers provide the user's email address to your application (Facebook). Some service providers only provide the email address at the user's option (GitHub supplies it if the user has provided a public email address). And other service providers do not provide the email address at all (Twitter, Meetup, LinkedIn).

In this tutorial, we first show a simple way to implement authentication. Then, in the chapter "Customizing the Application," we show how to request an email address from the user when he or she first requests access to your application.

Accounts You Will Need

Before beginning the tutorial, go to the service provider you'll use and obtain any required API keys. The tutorial assumes you will be using Twitter.

Twitter

Visit https://apps.twitter.com/app/new (https://apps.twitter.com/app/new) to register your application. You need to sign in to Twitter to access the "Create Application" page. When you register your application, use the following values:

- Application Website: http://example.com
- Callback URL: http://127.0.0.1:3000/
- Note: *http://localhost:3000/ doesn't work*

The values above are suitable for development. When you deploy your application to production, you'll need to update with a custom domain name.

Facebook

Visit http://developers.facebook.com/setup (http://developers.facebook.com/setup) to register your application.

GitHub

Visit https://github.com/settings/applications/new
(https://github.com/settings/applications/new) to register your application.

Other Service Providers

OmniAuth supports many other service providers. See the OmniAuth wiki for a List of Provider Strategies (https://github.com/intridea/omniauth/wiki/List-of-Strategies).

Create an Application

We'll use the Rails Composer (http://railsapps.github.io/rails-composer/) tool to build a starter app. For this tutorial, we'll start with the rails-bootstrap (https://github.com/RailsApps/rails-bootstrap) example application. Using a starter application will allow us to focus on the code that adds authentication without spending time configuring a basic Rails application. If you want to know more about creating a basic Rails application, see the Learn Ruby on Rails (https://tutorials.railsapps.org/tutorials/learn-rails) book.

To build the starter app, Rails 4.1 must be installed in your development environment. See the article Install Rails (http://railsapps.github.io/installing-rails.html) to set up your development environment.

Run the command:

```
$ rails new rails-omniauth -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  Choose a starter application.
      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-bootstrap**. Be careful: Don't choose "rails-omniauth." We want to build the basic "rails-bootstrap" application and then add the custom code we need. If you select "rails-omniauth" all the work will be done for you and you won't have any fun!

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

- Web server for development? **WEBrick (default)**
- Web server for production? **Same as development**
- Template engine? **ERB**
- Test framework? **RSpec with Capybara**
- Continuous testing? **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**

The README (https://github.com/RailsApps/rails-composer) for Rails Composer has more information about the options.

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.

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

```
$ cd rails-omniauth
```

You now have a Rails application that uses a SQLite database for data storage, the Bootstrap front-end framework, and the RSpec test framework. We'll customize it to add OmniAuth for authentication.

Gems

You'll need to add the following gems to the application Gemfile:

- omniauth (http://rubygems.org/gems/omniauth)
- omniauth-twitter (http://rubygems.org/gems/omniauth-twitter)

OmniAuth depends on external gems that implement access strategies for specific service providers. For example, if you want your users to authenticate using a Twitter account, you'll need the `omniauth-twitter` gem. See the OmniAuth wiki for a List of Provider Strategies (https://github.com/intridea/omniauth/wiki/List-of-Strategies) to determine if gems are available for the services you wish to use. If you don't see a service you want, try a RubyGems.org search (https://rubygems.org/search?utf8=%E2%9C%93&query=omniauth) for what you need.

This tutorial assumes you will be using Twitter so we'll add the omniauth-twitter gem to the Gemfile. You'll add a different gem if you want to use a different service provider.

Open your **Gemfile** and add the following:

```
gem 'omniauth'
gem 'omniauth-twitter'
```

Install the Required Gems

Install the required gems on your computer:

```
$ bundle install
```

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 (this happens automatically on Heroku).

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/ (http://localhost:3000). You should see the home page for our simple Rails Bootstrap starter application.

Configuration

This tutorial assumes you want to authenticate using Twitter. If you want to use other service providers (such as Facebook), you'll need to install a different gem. See the OmniAuth wiki for a List of Provider Strategies (https://github.com/intridea/omniauth/wiki/List-of-Strategies) if you want to use a different service provider.

OmniAuth Initializer

You'll need an OmniAuth initialization file **config/initializers/omniauth.rb** configured for the service provider you'll use. For most service providers, you must register your application and obtain API keys to use their authentication service. The arguments that you supply may not be the same for each OmniAuth strategy you use (check the documentation for each gem).

Create a file **config/initializers/omniauth.rb** with information for the service provider you'll use. For Twitter:

```
Rails.application.config.middleware.use OmniAuth::Builder do
  provider :twitter, Rails.application.secrets.omniauth_provider_key, Rails.application.secr
ets.omniauth_provider_secret
end
```

This initializer configures OmniAuth to use Twitter as a service provider, obtaining API keys from the **config/secrets.yml** file.

Keep Your API Keys Secret

To consolidate configuration settings in a single location, we obtain credentials from the **config/secrets.yml** file. To keep the credentials private, we use Unix environment variables to set your credentials.

Add credentials to the **config/secrets.yml** file:

```
development:
  omniauth_provider_key: <%= ENV["OMNIAUTH_PROVIDER_KEY"] %>
  omniauth_provider_secret: <%= ENV["OMNIAUTH_PROVIDER_SECRET"] %>
  domain_name: example.com
  secret_key_base: (not shown)

test:
  secret_key_base: (not shown)

# Do not keep production secrets in the repository,
# instead read values from the environment.
production:
  omniauth_provider_key: <%= ENV["OMNIAUTH_PROVIDER_KEY"] %>
  omniauth_provider_secret: <%= ENV["OMNIAUTH_PROVIDER_SECRET"] %>
  domain_name: <%= ENV["DOMAIN_NAME"] %>
  secret_key_base: <%= ENV["SECRET_KEY_BASE"] %>
```

Use the long `secret_key_base` that was generated for your application. Add the additional credentials to the file.

All configuration values in the **config/secrets.yml** file are available anywhere in the application as variables, for example as `Rails.application.secrets.omniauth_provider_key` .

If you don't want to use Unix environment variables, you can set each value directly in the **config/secrets.yml** file. The file must be in your git repository when you deploy to Heroku. However, you shouldn't save the file to a public GitHub repository where other people can see your credentials. If you've created a public repository and you commit the file with your API Keys, anyone who browses your repo will be able to obtain your API keys and masquerade as your application.

Set Environment Variables

Unix gives you a choice of shell programs; each has a slightly different way to set environment variables. If you are using the bash shell, you can set environment variables in your **.bashrc** or **.bash_profile** file:

```
export OMNIAUTH_PROVIDER_KEY="your_key"
export OMNIAUTH_PROVIDER_SECRET="your_secret"
```

See the article Rails Environment Variables (http://railsapps.github.io/rails-environment-variables.html) for more information.

Home Page

Our starter application already has a simple home page. Let's take a minute to examine the code that Rails Composer has generated. Later we'll customize the home page for a demonstration of OmniAuth.

Rails Composer created a Visitors controller in **app/controllers/visitors_controller.rb**, a home page in **app/views/visitors/index.html.erb**, and set up the necessary routes.

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.

View

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

Here is the home page file **app/views/visitors/index.html.erb**:

```
<h3>Welcome</h3>
```

Later we'll modify the home page to provide a link to a "Users" page and display how many users are registered.

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"
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).

User Model

Use the Rails generator to generate a model for a User.

The User will have name and email attributes. You can add other attributes that you may need. To keep the example simple, we'll initially create a User model without other attributes, controller, routes, or views.

```
$ rails generate model user provider:string uid:string name:string email:string
```

The file **app/models/user.rb** will look like this:

```
class User < ActiveRecord::Base
end
```

Run the migration to add the new table to the database:

```
$ rake db:migrate
```

Initially we won't use the email attribute because Twitter doesn't provide the user's email address. Later, in the chapter "Customizing the Application," we'll obtain the email address from the user.

Sessions

A Sessions controller is key to implementing authentication. 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.

Cookie-based sessions give us a way to manage data through multiple browser requests. Rails does all the work of setting up an encrypted, tamperproof session datastore. The data we most often want to persist throughout a session is an object that represents the user.

Sessions Controller

We'll use a SessionsController to create and destroy sessions to accommodate signing in and signing out the user.

Create a file **app/controllers/sessions_controller.rb**:

```
class SessionsController < ApplicationController
  def create
    raise request.env["omniauth.auth"].to_yaml
  end
end
```

For now, we'll set up the SessionsController to simply raise an exception. In development, Rails handles exceptions by displaying an error page, so by calling `raise request.env["omniauth.auth"].to_yaml`, we'll see the information returned by the service provider when authentication is successful.

Set Up Routes

OmniAuth supplies built-in routes for all its supported service providers. We'll add the OmniAuth routes now.

Edit the file **config/routes.rb**:

```
Rails.application.routes.draw do
  root to: 'visitors#index'
  get '/auth/:provider/callback' => 'sessions#create'
  get '/signin' => 'sessions#new', :as => :signin
  get '/signout' => 'sessions#destroy', :as => :signout
  get '/auth/failure' => 'sessions#failure'
end
```

Let's examine the routes more closely.

Here's the route for the callback URL:

```
get '/auth/:provider/callback' => 'sessions#create'
```

The path `/auth/twitter` will redirect to Twitter's website and allow the user to authenticate using Twitter. After successful authentication, OmniAuth triggers the callback URL `/auth/twitter/callback`. This route redirects to the Sessions controller `create` method.

It could be simplified as `match '/auth/twitter/callback' => 'sessions#create'` but by providing a `:provider` parameter we can easily enhance the application to handle more than one provider if necessary.

We also want to offer a feature that allows an authenticated user to sign out. We've added the following route to the file **config/routes.rb**:

```
get '/signout' => 'sessions#destroy', :as => :signout
```

The URL http://localhost:3000/signout (http://localhost:3000/signout) will call the `destroy` action of the Sessions controller. Adding `:as => :signout` to the route gives us a `signout_path` helper to use in our views.

The next route isn't strictly necessary. We use it for signing in to the application. OmniAuth allows us to directly initiate authentication by clicking a URL link such as `/auth/twitter` in any view. However, for a more completely RESTful Sessions controller, we'll have a `new` method that redirects to `/auth/twitter` in the Sessions controller. We've added the corresponding route to the file **config/routes.rb**:

```
get '/signin' => 'sessions#new', :as => :signin
```

The URL http://localhost:3000/signin (http://localhost:3000/signin) will call the `new` action of the Sessions controller. Adding `:as => :signin` to the route gives us a `signin_path` helper to use in our views.

We also must handle authentication failures. OmniAuth provides a failure URL `/auth/failure` that we can route to our Sessions controller. We've added the following route to the file **config/routes.rb**:

```
get '/auth/failure' => 'sessions#failure'
```

Soon we'll add a `failure` method to the Sessions controller to show an alert when authentication fails.

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/ (http://localhost:3000). You should see the default page.

Enter the URL http://localhost:3000/auth/twitter (http://localhost:3000/auth/twitter) and sign in with Twitter. You should be redirected to an error page that shows information from the Twitter authentication. The error message will include information similar to this:

```
--- !ruby/hash:OmniAuth::AuthHash
provider: twitter
uid: '123456789'
info: !ruby/hash:OmniAuth::AuthHash::InfoHash
  nickname: rails_apps
```

Though we see an error message, we've successfully authenticated with Twitter. Next we need to implement the callback processing to use the data returned by Twitter.

Authentication

Authentication with a service such as Twitter returns a hash to our application that provides information about the user. The dataset varies with each service provider. We will extract what we need from the hash we receive.

We'll modify the User model and the Sessions controller to capture data provided by the service provider.

Modify the User Model

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

```
class User < ActiveRecord::Base

  def self.create_with_omniauth(auth)
    create! do |user|
      user.provider = auth['provider']
      user.uid = auth['uid']
      if auth['info']
         user.name = auth['info']['name'] || ""
      end
    end
  end

end
```

We add a method to the User model that saves a new user to the database. Each service provider delivers user data in a slightly different format, so you may need to modify the file **app/models/user.rb** accordingly if you are using another service provider. In this case, we set the user's name using data provided by Twitter.

Modify the Sessions Controller

Replace the file **app/controllers/sessions_controller.rb**:

```
class SessionsController < ApplicationController

  def new
    redirect_to '/auth/twitter'
  end

  def create
    auth = request.env["omniauth.auth"]
    user = User.where(:provider => auth['provider'],
                      :uid => auth['uid'].to_s).first || User.create_with_omniauth(auth)
    reset_session
    session[:user_id] = user.id
    redirect_to root_url, :notice => 'Signed in!'
  end

  def destroy
    reset_session
    redirect_to root_url, :notice => 'Signed out!'
  end

  def failure
    redirect_to root_url, :alert => "Authentication error: #{params[:message].humanize}"
  end

end
```

Let's look at the methods in more detail.

New

We've already added a `signin` route to the file **config/routes.rb**. The following method implements the "Sign In" feature:

```
def new
  redirect_to '/auth/twitter'
end
```

It simply redirects to the OmniAuth `/auth/twitter` action.

Create

Previously, we just raised an exception after authenticating. Now we have a method that creates a User and stores the user id in the session.

```
def create
  auth = request.env["omniauth.auth"]
  user = User.where(:provider => auth['provider'],
                    :uid => auth['uid'].to_s).first || User.create_with_omniauth(auth)
  reset_session
  session[:user_id] = user.id
  redirect_to root_url, :notice => 'Signed in!'
end
```

When a visitor authenticates, we check the database to determine if the user's data has been recorded previously and, if not, we'll create a new user record. In either case, we'll add the user to the session and redirect to the application home page with a "Signed In" message.

The `reset_session` method is a Rails method that clears all objects stored in the session. The Ruby on Rails Security Guide (http://guides.rubyonrails.org/security.html#sessions) recommends resetting the session id upon successful authentication to protect against session fixation (https://www.owasp.org/index.php/Session_fixation) vulnerabilities.

Destroy

We've already added a `signout` route to the file **config/routes.rb**. This implements a "Sign Out" feature for the user:

```
def destroy
  reset_session
  redirect_to root_url, :notice => 'Signed out!'
end
```

The `reset_session` method is a Rails method that clears all objects stored in the session. Resetting the session is all that is needed to sign out the user.

Failure

We've already accommodated OmniAuth's `/auth/failure` route in the file **config/routes.rb** with a redirect to the Sessions controller `failure` method. This method handles authentication failure:

```
def failure
  redirect_to root_url, :alert => "Authentication error: #{params[:message].humanize}"
end
```

We simply redirect to the home page and display an alert.

Authentication Helpers

If you've created applications that use Devise for authentication, you have seen helper methods `current_user` and `user_signed_in?`. Devise provides these helper methods so we can obtain the current user and check if the current user is signed in.

OmniAuth doesn't provide these convenient helper methods so we'll create them for our application. We'll use the method names used by Devise, though we can give the helpers other names if we wish. We'll also create a `correct_user?` method so we can limit a user to only view his or her own profile when we implement a User#show action.

We want the helper methods to be available in any controller, so we'll add them to the Application controller.

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

  helper_method :current_user
  helper_method :user_signed_in?
  helper_method :correct_user?

  private
    def current_user
      begin
        @current_user ||= User.find(session[:user_id]) if session[:user_id]
      rescue Exception => e
        nil
      end
    end

    def user_signed_in?
      return true if current_user
    end

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

    def authenticate_user!
      if !current_user
        redirect_to root_url, :alert => 'You need to sign in for access to this page.'
      end
    end

end
```

The `current_user` helper method allows us to access the currently signed-in user from any controller. We obtain the User id from the session and retrieve the User object from the database, raising an exception if we can't find it.

The `user_signed_in?` helper allows us to distinguish anonymous visitors from users who are signed in.

We can use the `correct_user?` helper method to prevent a user from seeing any user profile but their own.

The `authenticate_user!` helper can be added to any controller to prevent access by visitors who are not signed in.

Navigation

Modify the file **app/views/layouts/_navigation_links.html.erb**:

```
<%# add navigation links to this file %>
<li><%= link_to 'About', page_path('about') %></li>
<% if user_signed_in? %>
  <li><%= link_to 'Sign out', signout_path %></li>
<% else %>
  <li><%= link_to 'Sign in', signin_path %></li>
<% end %>
<% if user_signed_in? %>
  <li><%= link_to 'Users', users_path %></li>
<% end %>
```

Previously, the application navigation bar contained only a link to the "About" page. Now we've added links to sign in and sign out, using our authentication helpers and the routes we've set up in the file **config/routes.rb**.

In a real-world application, you will not want any user to see a list of all registered users. But for our tutorial application, we'll use it to demonstrate access control.

User Pages

We've implemented everything needed for authentication with OmniAuth. Now let's see it in action by adding a user profile page. With the access control provided by OmniAuth, only the authenticated user will see his or her own profile.

Users Controller

Create a file **app/controllers/users_controller.rb**:

```
class UsersController < ApplicationController
  before_filter :authenticate_user!
  before_filter :correct_user?, :except => [:index]

  def index
    @users = User.all
  end

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

end
```

This is a standard RESTful controller. The `index` action will display a list of all users. The `show` action displays a user profile page.

We set a `before_filter` with the `authenticate_user!` helper method from the Application controller to require authentication before any controller action.

We set a `before_filter` with the `correct_user?` helper method for all controller actions except `index`. That means any signed-in user can see a list of users.

For the user profile page, we use the `correct_user?` helper method to limit access so only the signed-in user can view his or or her own profile.

List of Users

Add a 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>
```

We set a list of users in table (with lots of HTML) and render a partial that displays details for each user.

Add a partial **app/views/users/_user.html.erb**:

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

The partial provides individual user details for the list of users.

In a real-world application, you will not want any user to see a list of all registered users. But for our tutorial application, it is a useful feature.

User Profile View

Add a file **app/views/users/show.html.erb**:

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

This is a simple user profile, displaying the user's name.

Modify the Home Page

Modify the file **app/views/visitors/index.html.erb**:

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

For our tutorial application, we'll display a count of users with a link to the list of users on the home page.

Routes

Modify the file **config/routes.rb**:

```
Rails.application.routes.draw do
  resources :users
  root to: 'visitors#index'
  get '/auth/:provider/callback' => 'sessions#create'
  get '/signin' => 'sessions#new', :as => :signin
  get '/signout' => 'sessions#destroy', :as => :signout
  get '/auth/failure' => 'sessions#failure'
end
```

We've added routes for our RESTful User controller.

Test the App

Test the app with the command:

```
$ rails server
```

Open a browser window and navigate to http://localhost:3000/ (http://localhost:3000/).

When you sign in, you should be able to click through to see your user profile page.

Sign out and visit the URL http://localhost:3000/users (http://localhost:3000/users) and you'll see a message "You need to sign in for access to this page."

OmniAuth 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.

We'll look here at issues unique to testing OmniAuth. As you build your application, you'll want to test all the features that require a user to sign in. However, for purposes of testing, you won't want to actually authenticate using Twitter or another provider. Instead, you'll need to *mock* the authentication process. In this context, "to mock" means to bypass the actual authentication and fake it programmatically. Here we'll create the mocks we need for OmniAuth testing with RSpec.

First, be sure you are familiar with the configuration requirements of RSpec 3.0.

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 an **.rspec** file in the project folder. 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.

Mocks

OmniAuth has a built-in mocking mechanism, described on the OmniAuth wiki (https://github.com/intridea/omniauth/wiki/Integration-Testing).

To create a mock for authentication, create a helper file **spec/support/helpers/omniauth.rb**:

```
module Omniauth

  module Mock
    def auth_mock
      OmniAuth.config.mock_auth[:twitter] = {
        'provider' => 'twitter',
        'uid' => '123545',
        'user_info' => {
          'name' => 'mockuser'
        },
        'credentials' => {
          'token' => 'mock_token',
          'secret' => 'mock_secret'
        }
      }
    end
  end

end
```

We create a Ruby module (a collection of methods and constants, similar to a class) named `Mock`. To avoid collisions with other modules that might be named `Mock`, we create a namespace by nesting it within another module named `Omniauth`.

Our method `auth_mock` impersonates the hash returned by authentication with Twitter. In this tutorial, we are using Twitter as an authentication provider with the omniauth-twitter gem. If you are using a different authentication provider, check the gem documentation for details about the hash returned by the provider. As described in the OmniAuth wiki (https://github.com/intridea/omniauth/wiki/Integration-Testing), you can create a hash with the `:default` key to return a suitable hash for providers that haven't been specified.

We can use the `auth_mock` method in any test to replace actual authentication with Twitter.

Session Helper

For feature tests, we must simulate the user clicking the sign-in link, authenticating with Twitter, filling in the email field, and clicking the "Sign in" button. To avoid duplicating these steps for every feature test, we can create a helper method to use whenever we need to simulate signing in.

Let's add the session helper to the file **spec/support/helpers/omniauth.rb**:

```
module Omniauth

  module Mock
    def auth_mock
      OmniAuth.config.mock_auth[:twitter] = {
        'provider' => 'twitter',
        'uid' => '123545',
        'user_info' => {
          'name' => 'mockuser'
        },
        'credentials' => {
          'token' => 'mock_token',
          'secret' => 'mock_secret'
        }
      }
    end
  end

  module SessionHelpers
    def signin
      visit root_path
      expect(page).to have_content("Sign in")
      auth_mock
      click_link "Sign in"
    end
  end

end
```

The session helper includes Capybara actions and matchers combined in a Ruby module named `SessionHelpers`.

We visit the home page, call our `auth_mock` method to simulate authentication, click the "Sign in" link, and complete the sign-in process by providing an email address and clicking the "Sign in" button.

However, before we call the `signin` method in a feature test to obtain a signed-in user, we need to configure RSpec to make the helper method and mock available during testing.

Configuration

To make the helper method and mock available during testing, we must include them in the RSpec **spec/rails_helper.rb** file, or alternatively, include them in a **spec/support/helpers.rb** file:

```
RSpec.configure do |config|
  config.include Omniauth::Mock
  config.include Omniauth::SessionHelpers, type: :feature
end
OmniAuth.config.test_mode = true
```

This configuration file adds the method in the `Mock` module for use by all tests. We specify `type: :feature` to only add the `SessionHelpers` module to feature tests.

Finally, we enable OmniAuth test mode with `OmniAuth.config.test_mode = true`. In test mode, all requests to OmniAuth will be bypassed to use the mock authentication. A request to `/auth/provider` will redirect immediately to `/auth/provider/callback` and the mock hash will be available to the session.

Now that we've configured RSpec for OmniAuth testing, we can write actual tests. Let's look at feature tests first.

Sign In

To test the "Sign in" feature, we'll create test scenarios for a signing in with a valid account as well as signing in with an invalid account.

Create a file **spec/features/users/sign_in_spec.rb**:

```
# Feature: Sign in
#   As a user
#   I want to sign in
#   So I can visit protected areas of the site
feature 'Sign in', :omniauth do

  # Scenario: User can sign in with valid account
  #   Given I have a valid account
  #   And I am not signed in
  #   When I sign in
  #   Then I see a success message
  scenario "user can sign in with valid account" do
    signin
    expect(page).to have_content("Sign out")
  end

  # Scenario: User cannot sign in with invalid account
  #   Given I have no account
  #   And I am not signed in
  #   When I sign in
  #   Then I see an authentication error message
  scenario 'user cannot sign in with invalid account' do
    OmniAuth.config.mock_auth[:twitter] = :invalid_credentials
    visit root_path
    expect(page).to have_content("Sign in")
    click_link "Sign in"
    expect(page).to have_content('Authentication error')
  end

end
```

The comments contain simple user stories to describe the expected behavior.

We name the feature `Sign in` and assign the tag `omniauth` as a Ruby symbol. A tag allows us to run tests selectively, if we wish. The tag only serves as metadata to select which tests to run; if you don't include the tag, you can still run the tests.

The first test uses the `signin` helper method. We expect the page to contain the email address we specified in the `signin` method.

The second test doesn't use the `signin` helper method because we want to sign in using an invalid account. Instead of using the mock we created to simulate a valid account, we create a new mock that doesn't return a hash. As described in the OmniAuth wiki (https://github.com/intridea/omniauth/wiki/Integration-Testing), if we set a provider's mock to a symbol instead of a hash, it will fail, displaying the symbol as a string. It is a convenient trick to simulate failed authentication.

Sign Out

The test for the "Sign out" feature is very simple.

Create a file **spec/features/users/sign_out_spec.rb**:

```
# Feature: Sign out
#   As a user
#   I want to sign out
#   So I can protect my account from unauthorized access
feature 'Sign out', :omniauth do

  # Scenario: User signs out successfully
  #   Given I am signed in
  #   When I sign out
  #   Then I see a signed out message
  scenario 'user signs out successfully' do
    signin
    click_link 'Sign out'
    expect(page).to have_content 'Signed out'
  end

end
```

We use the `signin` helper method to sign in. Then we click the "Sign out" link and look for the "Signed out" message.

Sessions Controller

In principle, our feature tests for sign in and sign out are adequate to assure that the Sessions controller works as expected. However, we don't really know that the sign in feature has created a new user and established a new session. We can test the Sessions controller for that.

Create a file **spec/controllers/sessions_controller_spec.rb**:

```
describe SessionsController, :omniauth do

  before do
    request.env['omniauth.auth'] = auth_mock
  end

  describe "#create" do

    it "creates a user" do
      expect {post :create, provider: :twitter}.to change{ User.count }.by(1)
    end

    it "creates a session" do
      expect(session[:user_id]).to be_nil
      post :create, provider: :twitter
      expect(session[:user_id]).not_to be_nil
    end

    it "redirects to the home page" do
      post :create, provider: :twitter
      expect(response).to redirect_to root_url
    end

  end

  describe "#destroy" do

    before do
      post :create, provider: :twitter
    end

    it "resets the session" do
      expect(session[:user_id]).not_to be_nil
      delete :destroy
      expect(session[:user_id]).to be_nil
    end

    it "redirects to the home page" do
      delete :destroy
      expect(response).to redirect_to root_url
    end

  end

end
```

Our tests follow the RSpec conventions for testing a controller. We tag the group of tests with the `omniauth` tag in case we want to run tests selectively.

We use a `before` block to set an environment variable in the HTTP request to contain the hash provided by our `auth_mock` method, as recommended by the OmniAuth wiki (https://github.com/intridea/omniauth/wiki/Integration-Testing) article.

We test the `create` action with three scenarios. In each scenario, we start by initiating an HTTP request with the POST verb to the `create` method, sending the `provider` attribute as a parameter.

We create a user and check if the User count has increased by one.

Next, we test if the `create` action creates a new session. We make sure no user id exists in the session, initiate the HTTP request, and check if the action has added the user id to the session.

Finally, we check if the `create` action redirects to the page where we request the email address (the edit user page).

We test the `destroy` action with two scenarios. For each, we use a `before` block to set create a new session. We initiate an HTTP request with the DELETE verb to the `destroy` method. Then we check if the user id has been deleted from the session and the response has been redirected to the home page.

As you see, testing the Sessions controller gets closer to the underlying plumbing than our feature tests. Both kinds of tests are useful.

Now that you've seen the techniques unique to testing OmniAuth, you'll be able to build your application with a full suite of tests.

Run Tests

Run all tests using the `rspec` command:

```
$ rspec
```

The tests should pass.

Customizing the Application

Our tutorial application is simple but it provides functionality that can be added to any web application. You can add the code to another web application where you want users to sign in with Twitter or another provider. Or you can use the example application as a basis for your own custom application.

Email Option

Any business that wants to stay in contact with its customers will need its users' email addresses. Each service provider has a different policy about providing user details. For example, Twitter and LinkedIn do not provide a user's email address. GitHub only provides an email address if the user specified a public email address.

If you need a user's email address, you can add a feature to request it when they authenticate for the first time. This feature is an option. It is often required, so I've included it here and in the example application.

User Model

Previously, when we created the migration for the User model, we included an email attribute. We'll begin using the email attribute here.

For providers that supply a user's email address, we could modify the User model to obtain the email address from the provider. A modified **app/models/user.rb** file would look like this:

```
class User < ActiveRecord::Base

  def self.create_with_omniauth(auth)
    create! do |user|
      user.provider = auth['provider']
      user.uid = auth['uid']
      if auth['info']
         user.name = auth['info']['name'] || ""
         user.email = auth['info']['email'] || ""
      end
    end
  end

end
```

However, Twitter doesn't provide the user's email address so we'll have to ask the user for an email address when he or she signs in for the first time.

Edit View

Create a file **app/views/users/edit.html.erb**:

```
<div class="authform">
  <%= form_for(@user) do |f| %>
    <div class="form-group">
      <%= f.label :email %>
      <%= f.email_field :email, class: 'form-control' %>
    </div>
    <%= f.submit 'Sign in', class: 'button right' %>
  <% end %>
</div>
```

Functionally, the form is an email field and a submit button. Additional markup adds Bootstrap styling for a more attractive appearance.

Modify the Sessions Controller

Change the `create` method in the file **app/controllers/sessions_controller.rb**:

```
class SessionsController < ApplicationController

  def new
    redirect_to '/auth/twitter'
  end

  def create
    auth = request.env["omniauth.auth"]
    user = User.where(:provider => auth['provider'],
                      :uid => auth['uid'].to_s).first || User.create_with_omniauth(auth)
    reset_session
    session[:user_id] = user.id
    if user.email.blank?
      redirect_to edit_user_path(user), :alert => "Please enter your email address."
    else
      redirect_to root_url, :notice => 'Signed in!'
    end

  end

  def destroy
    reset_session
    redirect_to root_url, :notice => 'Signed out!'
  end

  def failure
    redirect_to root_url, :alert => "Authentication error: #{params[:message].humanize}"
  end

end
```

The additional code checks if the user email address is blank. If there is no email address, the method redirects to the user edit page and asks the user for an email address.

Modify the Users Controller

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

```
class UsersController < ApplicationController
  before_filter :authenticate_user!
  before_filter :correct_user?, :except => [:index]

  def index
    @users = User.all
  end

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

  def update
    @user = User.find(params[:id])
    if @user.update_attributes(secure_params)
      redirect_to @user
    else
      render :edit
    end
  end

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

  private

  def secure_params
    params.require(:user).permit(:email)
  end

end
```

We've added `edit` and `update` actions as well as a `secure_params` method. The `secure_params` method prevents mass assignment vulnerabilities when we obtain data from a submitted form (this is the security feature known as "strong parameters," introduced in Rails 4).

Modify the Users Partial

Modify the file **app/views/users/_user.html.erb**:

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

We'll display the user's email address in the list of users.

Modify the Users Profile

Modify 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>
```

We'll display the user's email address in the user profile view.

Test the App

Test the app with the command:

```
$ rails server
```

Open a browser window and navigate to http://localhost:3000/ (http://localhost:3000/).

Sign out and then sign in. If you sign in with Twitter, you should see a page requesting your email address.

Comments

Credits

Daniel Kehoe implemented the application and wrote the tutorial.

Did You Like the Tutorial?

Was the article 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/).

Any issues? Please create an issue (http://github.com/RailsApps/rails-omniauth/issues) on GitHub. Reporting (and patching!) issues helps everyone.

i think i got the above to work. now i get this error:
http://postimg.org/image/fpypw...

Reply · Share ›

Daniel Kehoe RailsApps ➚ Jay Rocc · 3 months ago

The secret_key_base variable is present in the config/secrets.yml file whenever a Rails application is created. Check the file and you should see it.

You can put credentials in either the config/initializers/omniauth.rb file or the config/secrets.yml file. Use Unix ENV variables to avoid revealing the credentials if you commit to a public GitHub repo.

Reply · Share ›

Joe Mellin · 3 months ago

Sorry really confused about the environmental variables. Okay with FB it seems that I have to have 2 separate apps (one for dev on localhost and the other on heroku). So I call the :facebook, ENV['FACEBOOK_KEY'], ENV['FACEBOOK_SECRET'] in omniauth.rb

Okay, then I tells rails that there is one to use in production and one in development in secret.yml

---
development:
FACEBOOK_KEY: <%= ENV["FACEBOOK_KEY"] %>
FACEBOOK_SECRET: <%= ENV["FACEBOOK_SECRET"] %>
domain_name: example.com

test:
Do not keep production secrets in the repository,
instead read values from the environment.

production:
FACEBOOK_KEY: <%= ENV["FACEBOOK_KEY"] %>

see more

Reply · Share ›

Daniel Kehoe RailsApps ➚ Joe Mellin · 3 months ago

In the file config/initializers/omniauth.rb, use Rails.application.secrets.omniauth_provider_key to obtain the key from the config/secrets.yml file. That way, your app will use the right key whether in development or production. If you prefer, you can skip the config/secrets.yml file and just use <%= ENV["OMNIAUTH_PROVIDER_KEY"] %> in the file config/initializers/omniauth.rb. It works just as well. The only advantage of the config/secrets.yml file is to keep all the secrets in one central location.

Reply · Share ›