plataformatec / devise

Flexible authentication solution for Rails with Warden. http://blog.plataformatec.com.br/tag/...

#rails #ruby #authentication #devise #plataformatec

3,430 commits | **20** branches | **125** releases | **491** contributors | MIT

Branch: **master** | New pull request | Find file | Clone or download

tegon Update changelog [ci skip] — Latest commit `451ba53` 3 days ago

app	Fix syntax for MRI 2.5.0-preview1.	2 months ago
bin	Add a `bin/test` executable to use Rails 5 Minitest report.	2 years ago
config/locales	Change email_change => email_changed notification	9 months ago
gemfiles	Remove TODO when Rails 5.0.3 is released	2 months ago
guides/bug_report_templates	Drop `_master` suffix as the bug template doesnt use Rails/Devise mas...	2 years ago
lib	Fix missing validations on Signup (#4674)	3 days ago
test	Fix missing validations on Signup (#4674)	3 days ago
.gitignore	Cache bundle results on Travis to S3	4 years ago

.travis.yml	CI against Ruby 2.2.8, 2.3.5, and 2.4.2	3 months ago
.yardopts	Add config to customize documentation.	5 years ago
CHANGELOG.md	Update changelog [ci skip]	3 days ago
CODE_OF_CONDUCT.md	Add a Code of Conduct	2 years ago
CONTRIBUTING.md	Expand `CONTRIBUTING.md` [ci skip].	a year ago
Gemfile	Removed gem minitest	a month ago
Gemfile.lock	bundle installed	a month ago
ISSUE_TEMPLATE.md	Add issue template	9 days ago
MIT-LICENSE	Update copyright year to 2017	8 months ago
README.md	Provide additional info on devise_scope usage	25 days ago
Rakefile	Formatting	3 years ago
devise.gemspec	Allow Rails 5.1	7 months ago
devise.png	Run image optimizer on the logo (i'm done)	5 years ago

README.md

By Plataformatec.

build passing maintainability A

This README is also available in a friendly navigable format.

Devise is a flexible authentication solution for Rails based on Warden. It:

- Is Rack based;
- Is a complete MVC solution based on Rails engines;
- Allows you to have multiple models signed in at the same time;
- Is based on a modularity concept: use only what you really need.

It's composed of 10 modules:

- Database Authenticatable: hashes and stores a password in the database to validate the authenticity of a user while signing in. The authentication can be done both through POST requests or HTTP Basic Authentication.
- Omniauthable: adds OmniAuth (https://github.com/omniauth/omniauth) support.
- Confirmable: sends emails with confirmation instructions and verifies whether an account is already confirmed during sign in.
- Recoverable: resets the user password and sends reset instructions.
- Registerable: handles signing up users through a registration process, also allowing them to edit and destroy their account.
- Rememberable: manages generating and clearing a token for remembering the user from a saved cookie.
- Trackable: tracks sign in count, timestamps and IP address.
- Timeoutable: expires sessions that have not been active in a specified period of time.
- Validatable: provides validations of email and password. It's optional and can be customized, so you're able to define your own validations.
- Lockable: locks an account after a specified number of failed sign-in attempts. Can unlock via email or after a specified time period.

Information

The Devise wiki

The Devise Wiki has lots of additional information about Devise including many "how-to" articles and answers to the most frequently asked questions. Please browse the Wiki after finishing this README:

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

Bug reports

If you discover a problem with Devise, we would like to know about it. However, we ask that you please review these guidelines before submitting a bug report:

https://github.com/plataformatec/devise/wiki/Bug-reports

If you have discovered a security related bug, please do *NOT* use the GitHub issue tracker. Send an email to opensource@plataformatec.com.br.

StackOverflow and Mailing List

If you have any questions, comments, or concerns, please use StackOverflow instead of the GitHub issue tracker:

http://stackoverflow.com/questions/tagged/devise

The deprecated mailing list can still be read on

https://groups.google.com/group/plataformatec-devise

RDocs

You can view the Devise documentation in RDoc format here:

http://rubydoc.info/github/plataformatec/devise/master/frames

If you need to use Devise with previous versions of Rails, you can always run "gem server" from the command line after you install the gem to access the old documentation.

Example applications

There are a few example applications available on GitHub that demonstrate various features of Devise with different versions of Rails. You can view them here:

https://github.com/plataformatec/devise/wiki/Example-Applications

Extensions

Our community has created a number of extensions that add functionality above and beyond what is included with Devise. You can view a list of available extensions and add your own here:

https://github.com/plataformatec/devise/wiki/Extensions

Contributing

We hope that you will consider contributing to Devise. Please read this short overview for some information about how to get started:

https://github.com/plataformatec/devise/wiki/Contributing

You will usually want to write tests for your changes. To run the test suite, go into Devise's top-level directory and run "bundle install" and "rake". For the tests to pass, you will need to have a MongoDB server (version 2.0 or newer) running on your system.

Starting with Rails?

If you are building your first Rails application, we recommend you *do not* use Devise. Devise requires a good understanding of the Rails Framework. In such cases, we advise you to start a simple authentication system from scratch. Today, we have three resources that should help you get started:

- Michael Hartl's online book: https://www.railstutorial.org/book/modeling_users
- Ryan Bates' Railscast: http://railscasts.com/episodes/250-authentication-from-scratch
- Codecademy's Ruby on Rails: Authentication and Authorization: http://www.codecademy.com/en/learn/rails-auth

Once you have solidified your understanding of Rails and authentication mechanisms, we assure you Devise will be very pleasant to work with. 😃

Getting started

Devise 4.0 works with Rails 4.1 onwards. You can add it to your Gemfile with:

```
gem 'devise'
```

Then run `bundle install`

Next, you need to run the generator:

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

At this point, a number of instructions will appear in the console. Among these instructions, you'll need to set up the default URL options for the Devise mailer in each environment. Here is a possible configuration for `config/environments/development.rb` :

```
config.action_mailer.default_url_options = { host: 'localhost', port: 3000 }
```

The generator will install an initializer which describes ALL of Devise's configuration options. It is *imperative* that you take a look at it. When you are done, you are ready to add Devise to any of your models using the generator.

In the following command you will replace `MODEL` with the class name used for the application’s users (it’s frequently `User` but could also be `Admin`). This will create a model (if one does not exist) and configure it with the default Devise modules. The generator also configures your `config/routes.rb` file to point to the Devise controller.

```
$ rails generate devise MODEL
```

Next, check the MODEL for any additional configuration options you might want to add, such as confirmable or lockable. If you add an option, be sure to inspect the migration file (created by the generator if your ORM supports them) and uncomment the appropriate section. For example, if you add the confirmable option in the model, you'll need to uncomment the Confirmable section in the migration.

Then run `rails db:migrate`

You should restart your application after changing Devise's configuration options. Otherwise, you will run into strange errors, for example, users being unable to login and route helpers being undefined.

Controller filters and helpers

Devise will create some helpers to use inside your controllers and views. To set up a controller with user authentication, just add this before_action (assuming your devise model is 'User'):

```
before_action :authenticate_user!
```

For Rails 5, note that `protect_from_forgery` is no longer prepended to the `before_action` chain, so if you have set `authenticate_user` before `protect_from_forgery`, your request will result in "Can't verify CSRF token authenticity." To resolve this, either change the order in which you call them, or use `protect_from_forgery prepend: true`.

If your devise model is something other than User, replace "_user" with "_yourmodel". The same logic applies to the instructions below.

To verify if a user is signed in, use the following helper:

```
user_signed_in?
```

For the current signed-in user, this helper is available:

```
current_user
```

You can access the session for this scope:

```
user_session
```

After signing in a user, confirming the account or updating the password, Devise will look for a scoped root path to redirect to. For instance, when using a `:user` resource, the `user_root_path` will be used if it exists; otherwise, the default `root_path` will be used. This means that you need to set the root inside your routes:

```
root to: 'home#index'
```

You can also override `after_sign_in_path_for` and `after_sign_out_path_for` to customize your redirect hooks.

Notice that if your Devise model is called `Member` instead of `User`, for example, then the helpers available are:

```
before_action :authenticate_member!

member_signed_in?

current_member

member_session
```

Configuring Models

The Devise method in your models also accepts some options to configure its modules. For example, you can choose the cost of the hashing algorithm with:

```
devise :database_authenticatable, :registerable, :confirmable, :recoverable, stretches: 12
```

Besides `:stretches`, you can define `:pepper`, `:encryptor`, `:confirm_within`, `:remember_for`, `:timeout_in`, `:unlock_in` among other options. For more details, see the initializer file that was created when you invoked the "devise:install" generator described above. This file is usually located at `/config/initializers/devise.rb`.

Strong Parameters

The Parameter Sanitizer API has changed for Devise 4

For previous Devise versions see https://github.com/plataformatec/devise/tree/3-stable#strong-parameters

When you customize your own views, you may end up adding new attributes to forms. Rails 4 moved the parameter sanitization from the model to the controller, causing Devise to handle this concern at the controller as well.

There are just three actions in Devise that allow any set of parameters to be passed down to the model, therefore requiring sanitization. Their names and default permitted parameters are:

- `sign_in` (`Devise::SessionsController#create`) - Permits only the authentication keys (like `email`)
- `sign_up` (`Devise::RegistrationsController#create`) - Permits authentication keys plus `password` and `password_confirmation`
- `account_update` (`Devise::RegistrationsController#update`) - Permits authentication keys plus `password` , `password_confirmation` and `current_password`

In case you want to permit additional parameters (the lazy way™), you can do so using a simple before filter in your `ApplicationController` :

```
class ApplicationController < ActionController::Base
  before_action :configure_permitted_parameters, if: :devise_controller?

  protected

  def configure_permitted_parameters
    devise_parameter_sanitizer.permit(:sign_up, keys: [:username])
  end
end
```

The above works for any additional fields where the parameters are simple scalar types. If you have nested attributes (say you're using `accepts_nested_attributes_for`), then you will need to tell devise about those nestings and types. Devise allows you to completely change Devise defaults or invoke custom behaviour by passing a block:

To permit simple scalar values for username and email, use this

```ruby
def configure_permitted_parameters
  devise_parameter_sanitizer.permit(:sign_in) do |user_params|
    user_params.permit(:username, :email)
  end
end
```

If you have some checkboxes that express the roles a user may take on registration, the browser will send those selected checkboxes as an array. An array is not one of Strong Parameters' permitted scalars, so we need to configure Devise in the following way:

```ruby
def configure_permitted_parameters
  devise_parameter_sanitizer.permit(:sign_up) do |user_params|
    user_params.permit({ roles: [] }, :email, :password, :password_confirmation)
  end
end
```

For the list of permitted scalars, and how to declare permitted keys in nested hashes and arrays, see

https://github.com/rails/strong_parameters#nested-parameters

If you have multiple Devise models, you may want to set up a different parameter sanitizer per model. In this case, we recommend inheriting from `Devise::ParameterSanitizer` and adding your own logic:

```ruby
class User::ParameterSanitizer < Devise::ParameterSanitizer
  def initialize(*)
    super
    permit(:sign_up, keys: [:username, :email])
  end
end
```

And then configure your controllers to use it:

```
class ApplicationController < ActionController::Base
  protected

  def devise_parameter_sanitizer
    if resource_class == User
      User::ParameterSanitizer.new(User, :user, params)
    else
      super # Use the default one
    end
  end
end
```

The example above overrides the permitted parameters for the user to be both `:username` and `:email`. The non-lazy way to configure parameters would be by defining the before filter above in a custom controller. We detail how to configure and customize controllers in some sections below.

Configuring views

We built Devise to help you quickly develop an application that uses authentication. However, we don't want to be in your way when you need to customize it.

Since Devise is an engine, all its views are packaged inside the gem. These views will help you get started, but after some time you may want to change them. If this is the case, you just need to invoke the following generator, and it will copy all views to your application:

```
$ rails generate devise:views
```

If you have more than one Devise model in your application (such as `User` and `Admin`), you will notice that Devise uses the same views for all models. Fortunately, Devise offers an easy way to customize views. All you need to do is set `config.scoped_views = true` inside the `config/initializers/devise.rb` file.

After doing so, you will be able to have views based on the role like `users/sessions/new` and `admins/sessions/new`. If no view is found within the scope, Devise will use the default view at `devise/sessions/new`. You can also use the generator to generate scoped views:

```
$ rails generate devise:views users
```

If you would like to generate only a few sets of views, like the ones for the `registerable` and `confirmable` module, you can pass a list of modules to the generator with the `-v` flag.

```
$ rails generate devise:views -v registrations confirmations
```

Configuring controllers

If the customization at the views level is not enough, you can customize each controller by following these steps:

1. Create your custom controllers using the generator which requires a scope:

   ```
   $ rails generate devise:controllers [scope]
   ```

   If you specify `users` as the scope, controllers will be created in `app/controllers/users/`. And the sessions controller will look like this:

   ```
   class Users::SessionsController < Devise::SessionsController
     # GET /resource/sign_in
     # def new
     #   super
     # end
     ...
   end
   ```

2. Tell the router to use this controller:

   ```
   devise_for :users, controllers: { sessions: 'users/sessions' }
   ```

3.

Copy the views from `devise/sessions` to `users/sessions`. Since the controller was changed, it won't use the default views located in `devise/sessions`.

4. Finally, change or extend the desired controller actions.

   You can completely override a controller action:

   ```ruby
   class Users::SessionsController < Devise::SessionsController
     def create
       # custom sign-in code
     end
   end
   ```

   Or you can simply add new behaviour to it:

   ```ruby
   class Users::SessionsController < Devise::SessionsController
     def create
       super do |resource|
         BackgroundWorker.trigger(resource)
       end
     end
   end
   ```

   This is useful for triggering background jobs or logging events during certain actions.

Remember that Devise uses flash messages to let users know if sign in was successful or unsuccessful. Devise expects your application to call `flash[:notice]` and `flash[:alert]` as appropriate. Do not print the entire flash hash, print only specific keys. In some circumstances, Devise adds a `:timedout` key to the flash hash, which is not meant for display. Remove this key from the hash if you intend to print the entire hash.

Configuring routes

Devise also ships with default routes. If you need to customize them, you should probably be able to do it through the devise_for method. It accepts several options like :class_name, :path_prefix and so on, including the possibility to change path names for I18n:

```
devise_for :users, path: 'auth', path_names: { sign_in: 'login', sign_out: 'logout', password: 'secret', confirmatior
```

Be sure to check `devise_for` documentation for details.

If you have the need for more deep customization, for instance to also allow "/sign_in" besides "/users/sign_in", all you need to do is create your routes normally and wrap them in a `devise_scope` block in the router:

```
devise_scope :user do
  get 'sign_in', to: 'devise/sessions#new'
end
```

This way, you tell Devise to use the scope `:user` when "/sign_in" is accessed. Notice `devise_scope` is also aliased as `as` in your router.

Please note: You will still need to add `devise_for` in your routes in order to use helper methods such as `current_user` .

```
devise_for :users, skip: :all
```

I18n

Devise uses flash messages with I18n, in conjunction with the flash keys :notice and :alert. To customize your app, you can set up your locale file:

```
en:
  devise:
    sessions:
      signed_in: 'Signed in successfully.'
```

You can also create distinct messages based on the resource you've configured using the singular name given in routes:

```yaml
en:
  devise:
    sessions:
      user:
        signed_in: 'Welcome user, you are signed in.'
      admin:
        signed_in: 'Hello admin!'
```

The Devise mailer uses a similar pattern to create subject messages:

```yaml
en:
  devise:
    mailer:
      confirmation_instructions:
        subject: 'Hello everybody!'
        user_subject: 'Hello User! Please confirm your email'
      reset_password_instructions:
        subject: 'Reset instructions'
```

Take a look at our locale file to check all available messages. You may also be interested in one of the many translations that are available on our wiki:

https://github.com/plataformatec/devise/wiki/I18n

Caution: Devise Controllers inherit from ApplicationController. If your app uses multiple locales, you should be sure to set I18n.locale in ApplicationController.

Test helpers

Devise includes some test helpers for controller and integration tests. In order to use them, you need to include the respective module in your test cases/specs.

Controller tests

Controller tests require that you include `Devise::Test::ControllerHelpers` on your test case or its parent `ActionController::TestCase` superclass.

```ruby
class PostsControllerTest < ActionController::TestCase
  include Devise::Test::ControllerHelpers
end
```

If you're using RSpec, you can put the following inside a file named `spec/support/devise.rb` or in your `spec/spec_helper.rb` (or `spec/rails_helper.rb` if you are using `rspec-rails`):

```ruby
RSpec.configure do |config|
  config.include Devise::Test::ControllerHelpers, type: :controller
  config.include Devise::Test::ControllerHelpers, type: :view
end
```

Just be sure that this inclusion is made *after* the `require 'rspec/rails'` directive.

Now you are ready to use the `sign_in` and `sign_out` methods on your controller tests:

```ruby
sign_in @user
sign_in @user, scope: :admin
```

If you are testing Devise internal controllers or a controller that inherits from Devise's, you need to tell Devise which mapping should be used before a request. This is necessary because Devise gets this information from the router, but since controller tests do not pass through the router, it needs to be stated explicitly. For example, if you are testing the user scope, simply use:

```ruby
test 'GET new' do
  # Mimic the router behavior of setting the Devise scope through the env.
  @request.env['devise.mapping'] = Devise.mappings[:user]
```

```
    # Use the sign_in helper to sign in a fixture `User` record.
    sign_in users(:alice)

    get :new

    # assert something
  end
```

Integration tests

Integration test helpers are available by including the `Devise::Test::IntegrationHelpers` module.

```ruby
class PostsTests < ActionDispatch::IntegrationTest
  include Devise::Test::IntegrationHelpers
end
```

Now you can use the following `sign_in` and `sign_out` methods in your integration tests:

```ruby
sign_in users(:bob)
sign_in users(:bob), scope: :admin

sign_out :user
```

RSpec users can include the `IntegrationHelpers` module on their `:feature` specs.

```ruby
RSpec.configure do |config|
  config.include Devise::Test::IntegrationHelpers, type: :feature
end
```

Unlike controller tests, integration tests do not need to supply the `devise.mapping` `env` value, as the mapping can be inferred by the routes that are executed in your tests.

You can read more about testing your Rails 3 - Rails 4 controllers with RSpec in the wiki:

- https://github.com/plataformatec/devise/wiki/How-To:-Test-controllers-with-Rails-3-and-4-%28and-RSpec%29

OmniAuth

Devise comes with OmniAuth support out of the box to authenticate with other providers. To use it, simply specify your OmniAuth configuration in `config/initializers/devise.rb` :

```ruby
config.omniauth :github, 'APP_ID', 'APP_SECRET', scope: 'user,public_repo'
```

You can read more about OmniAuth support in the wiki:

- https://github.com/plataformatec/devise/wiki/OmniAuth:-Overview

Configuring multiple models

Devise allows you to set up as many Devise models as you want. If you want to have an Admin model with just authentication and timeout features, in addition to the User model above, just run:

```ruby
# Create a migration with the required fields
create_table :admins do |t|
  t.string :email
  t.string :encrypted_password
  t.timestamps null: false
end

# Inside your Admin model
devise :database_authenticatable, :timeoutable

# Inside your routes
devise_for :admins

# Inside your protected controller
before_action :authenticate_admin!

# Inside your controllers and views
```

```
admin_signed_in?
current_admin
admin_session
```

Alternatively, you can simply run the Devise generator.

Keep in mind that those models will have completely different routes. They **do not** and **cannot** share the same controller for sign in, sign out and so on. In case you want to have different roles sharing the same actions, we recommend that you use a role-based approach, by either providing a role column or using a dedicated gem for authorization.

ActiveJob Integration

If you are using Rails 4.2 and ActiveJob to deliver ActionMailer messages in the background through a queuing back-end, you can send Devise emails through your existing queue by overriding the `send_devise_notification` method in your model.

```ruby
def send_devise_notification(notification, *args)
  devise_mailer.send(notification, self, *args).deliver_later
end
```

Password reset tokens and Rails logs

If you enable the Recoverable module, note that a stolen password reset token could give an attacker access to your application. Devise takes effort to generate random, secure tokens, and stores only token digests in the database, never plaintext. However the default logging behavior in Rails can cause plaintext tokens to leak into log files:

1. Action Mailer logs the entire contents of all outgoing emails to the DEBUG level. Password reset tokens delivered to users in email will be leaked.
2. Active Job logs all arguments to every enqueued job at the INFO level. If you configure Devise to use `deliver_later` to send password reset emails, password reset tokens will be leaked.

Rails sets the production logger level to DEBUG by default. Consider changing your production logger level to WARN if you wish to prevent tokens from being leaked into your logs. In `config/environments/production.rb` :

```
config.log_level = :warn
```

Other ORMs

Devise supports ActiveRecord (default) and Mongoid. To select another ORM, simply require it in the initializer file.

Additional information

Heroku

Using Devise on Heroku with Ruby on Rails 3.2 requires setting:

```
config.assets.initialize_on_precompile = false
```

Read more about the potential issues at http://guides.rubyonrails.org/asset_pipeline.html

Warden

Devise is based on Warden, which is a general Rack authentication framework created by Daniel Neighman. We encourage you to read more about Warden here:

https://github.com/hassox/warden

Contributors

We have a long list of valued contributors. Check them all at:

https://github.com/plataformatec/devise/graphs/contributors

License