



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.

Table of Contents

- Information

-
- The Devise wiki
 - Bug reports
 - StackOverflow and Mailing List
 - RDocs
 - Example applications
 - Extensions
 - Contributing
 - Starting with Rails?
 - Getting started
 - Controller filters and helpers
 - Configuring Models
 - Strong Parameters
 - Configuring views
 - Configuring controllers
 - Configuring routes
 - I18n
 - Test helpers
 - Controller tests
 - Integration tests
 - OmniAuth
 - Configuring multiple models
 - Active Job Integration
 - Password reset tokens and Rails logs
 - Other ORMs
 - Rails API mode
 - Additional information
 - Warden
 - Contributors
 - License

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/heartcombo/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/heartcombo/devise/wiki/Bug-reports>

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

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/heartcombo/devise/main/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/heartcombo/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/heartcombo/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/heartcombo/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 `bin/test`. Devise works with multiple Ruby and Rails versions, and ActiveRecord and Mongoid ORMs, which means you can run the test suite with some modifiers: `DEVISE_ORM` and `BUNDLE_GEMFILE`.

DEVISE_ORM

Since Devise supports both Mongoid and ActiveRecord, we rely on this variable to run specific code for each ORM. The default value of `DEVISE_ORM` is `active_record`. To run the tests for Mongoid, you can pass `mongoid`:

```
1 DEVISE_ORM=mongoid bin/test
2
3 ==> Devise.orm = :mongoid
```

When running the tests for Mongoid, you will need to have a MongoDB server (version 2.0 or newer) running on your system.

Please note that the command output will show the variable value being used.

BUNDLE_GEMFILE

We can use this variable to tell bundler what Gemfile it should use (instead of the one in the current directory). Inside the `gemfiles` directory, we have one for each version of Rails we support. When you send us a pull request, it may happen that the test suite breaks using some of them. If that's the case, you can simulate the same environment using the `BUNDLE_GEMFILE` variable. For example, if the tests broke using Ruby 3.0.0 and Rails 6.0, you can do the following:

```
1 rbnbv shell 3.0.0 # or rvm use 3.0.0
2 BUNDLE_GEMFILE=gemfiles/Gemfile-rails-6-0 bundle install
3 BUNDLE_GEMFILE=gemfiles/Gemfile-rails-6-0 bin/test
```

You can also combine both of them if the tests broke for Mongoid:

```
1 BUNDLE_GEMFILE=gemfiles/Gemfile-rails-6-0 bundle install
2 BUNDLE_GEMFILE=gemfiles/Gemfile-rails-6-0 DEVISE ORM=mongoid bin/test
```

Running tests

Devise uses Mini Test as test framework.

- Running all tests:

```
1 bin/test
```

- Running tests for an specific file:

```
1 bin/test test/models/trackable_test.rb
```

- Running a specific test given a regex:

```
1 bin/test test/models/trackable_test.rb:16
```

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. Here's a few resources that should help you get started:

- Michael Hartl's online book: https://www.railstutorial.org/book/modeling_users
- Ryan Bates' Railscasts: <http://railscasts.com/episodes/250-authentication-from-scratch> and <http://railscasts.com/episodes/250-authentication-from-scratch-revised>
- Codecademy's Ruby on Rails: Authentication and Authorization: <https://www.codecademy.com/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. :smiley:

Getting started

Devise 4.0 works with Rails 6.0 onwards. Run:

```
1 bundle add devise
```

Next, you need to run the generator:

```
1 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`:

```
1 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.

```
1 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 (this includes stopping spring). 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'):

```
1 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:

```
1 user_signed_in?
```

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

```
1 current_user
```

You can access the session for this scope:

```
1 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:

```
1 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:

```
1 before_action :authenticate_member!  
2  
3 member_signed_in?  
4  
5 current_member  
6  
7 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:

```
1 devise :database_authenticatable, :registerable, :confirmable, :  
    recoverable, stretches: 13
```

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 :warning:

For previous Devise versions see <https://github.com/heartcombo/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 action in your `ApplicationController`:

```
1 class ApplicationController < ActionController::Base  
2   before_action :configure_permitted_parameters, if: :devise_controller?  
3  
4   protected  
5  
6   def configure_permitted_parameters
```

```
7     devise_parameter_sanitizer.permit(:sign_up, keys: [:username])
8   end
9 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:

```
1 class ApplicationController < ActionController::Base
2   before_action :configure_permitted_parameters, if: :devise_controller?
3
4   protected
5
6   def configure_permitted_parameters
7     devise_parameter_sanitizer.permit(:sign_up, keys: [:first_name, :last_name, address_attributes: [:country, :state, :city, :area, :postal_code]])
8   end
9 end
```

Devise allows you to completely change Devise defaults or invoke custom behavior by passing a block:

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

```
1 def configure_permitted_parameters
2   devise_parameter_sanitizer.permit(:sign_in) do |user_params|
3     user_params.permit(:username, :email)
4   end
5 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:

```
1 def configure_permitted_parameters
2   devise_parameter_sanitizer.permit(:sign_up) do |user_params|
3     user_params.permit({ roles: [] }, :email, :password, :password_confirmation)
4   end
5 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:

```
1 class User::ParameterSanitizer < Devise::ParameterSanitizer
2   def initialize(*)
3     super
4     permit(:sign_up, keys: [:username, :email])
5   end
6 end
```

And then configure your controllers to use it:

```
1 class ApplicationController < ActionController::Base
2   protected
3
4   def devise_parameter_sanitizer
5     if resource_class == User
6       User::ParameterSanitizer.new(User, :user, params)
7     else
8       super # Use the default one
9     end
10  end
11 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:

```
1 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:

```
1 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 views to the generator with the `-v` flag.

```
1 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:

```
1 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:

```
1 class Users::SessionsController < Devise::SessionsController
2   # GET /resource/sign_in
3   # def new
4   #   super
5   # end
6   ...
7 end
```

Use the `-c` flag to specify one or more controllers, for example: `rails generate devise:controllers users -c sessions`)

2. Tell the router to use this controller:

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

3. Recommended but not required: copy (or move) the views from `devise/sessions` to `users/sessions`. Rails will continue using the views from `devise/sessions` due to inheritance if you skip this step, but having the views matching the controller(s) keeps things consistent.
4. Finally, change or extend the desired controller actions.

You can completely override a controller action:

```
1 class Users::SessionsController < Devise::SessionsController
2   def create
3     # custom sign-in code
```

```
4   end
5   end
```

Or you can simply add new behavior to it:

```
1  class Users::SessionsController < Devise::SessionsController
2    def create
3      super do |resource|
4        BackgroundWorker.trigger(resource)
5      end
6    end
7  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:

```
1  devise_for :users, path: 'auth', path_names: { sign_in: 'login',
  sign_out: 'logout', password: 'secret', confirmation: 'verification'
  , unlock: 'unblock', registration: 'register', sign_up: '
  cmon_let_me_in' }
```

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:

```
1  devise_scope :user do
2    get 'sign_in', to: 'devise/sessions#new'
3  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`.

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

Hotwire/Turbo

Devise integrates with Hotwire/Turbo by treating such requests as navigational, and configuring certain responses for errors and redirects to match the expected behavior. New apps are generated with the following response configuration by default, and existing apps may opt-in by adding the config to their Devise initializers:

```
1 Devise.setup do |config|
2   # ...
3   # When using Devise with Hotwire/Turbo, the http status for error
   responses
4   # and some redirects must match the following. The default in Devise
   for existing
5   # apps is `200 OK` and `302 Found` respectively, but new apps are
   generated with
6   # these new defaults that match Hotwire/Turbo behavior.
7   # Note: These might become the new default in future versions of
   Devise.
8   config.responder.error_status = :unprocessable_entity
9   config.responder.redirect_status = :see_other
10 end
```

Important: these custom responses require the `responders` gem version to be 3.1.0 or higher, please make sure you update it if you're going to use this configuration. Check this upgrade guide for more info.

Note: the above statuses configuration may become the default for Devise in a future release.

There are a couple other changes you might need to make in your app to work with Hotwire/Turbo, if you're migrating from rails-ujs:

- The `data-confirm` option that adds a confirmation modal to buttons/forms before submission needs to change to `data-turbo-confirm`, so that Turbo handles those appropriately.
- The `data-method` option that sets the request method for link submissions needs to change to `data-turbo-method`. This is not necessary for `button_to` or `forms` since Turbo can handle those.

If you're setting up Devise to sign out via `:delete`, and you're using links (instead of buttons wrapped in a form) to sign out with the `method: :delete` option, they will need to be updated as described above. (Devise does not provide sign out links/buttons in its shared views.)

Make sure to inspect your views looking for those, and change appropriately.

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:

```
1 en:
2   devise:
3     sessions:
4       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:

```
1 en:
2   devise:
3     sessions:
4       user:
5         signed_in: 'Welcome user, you are signed in.'
6       admin:
7         signed_in: 'Hello admin!'
```

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

```
1 en:
2   devise:
3     mailer:
4       confirmation_instructions:
5         subject: 'Hello everybody!'
6       user_subject: 'Hello User! Please confirm your email'
7       reset_password_instructions:
8         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/heartcombo/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::IntegrationHelpers` on your test case or its parent `ActionController::TestCase` superclass. For Rails versions prior to 5, include `Devise::Test::ControllerHelpers` instead, since the superclass for controller tests was changed to `ActionDispatch::IntegrationTest` (for more details, see the Integration tests section).

```
1 class PostsControllerTest < ActionController::TestCase
2   include Devise::Test::IntegrationHelpers # Rails >= 5
3 end
```

```
1 class PostsControllerTest < ActionController::TestCase
2   include Devise::Test::ControllerHelpers # Rails < 5
3 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`):

```
1 RSpec.configure do |config|
2   config.include Devise::Test::ControllerHelpers, type: :controller
3   config.include Devise::Test::ControllerHelpers, type: :view
4 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:

```
1 sign_in @user
2 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:

```
1 test 'GET new' do
2   # Mimic the router behavior of setting the Devise scope through the
   env.
3   @request.env['devise.mapping'] = Devise.mappings[:user]
4
5   # Use the sign_in helper to sign in a fixture `User` record.
6   sign_in users(:alice)
7
8   get :new
9
10  # assert something
11 end
```

Integration tests

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

```
1 class PostsTests < ActionDispatch::IntegrationTest
2   include Devise::Test::IntegrationHelpers
3 end
```

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

```
1 sign_in users(:bob)
2 sign_in users(:bob), scope: :admin
3
4 sign_out :user
```

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

```
1 RSpec.configure do |config|
2   config.include Devise::Test::IntegrationHelpers, type: :feature
3 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 controllers with RSpec in the wiki:

- [https://github.com/heartcombo/devise/wiki/How-To:-Test-controllers-with-Rails-\(and-RSpec\)](https://github.com/heartcombo/devise/wiki/How-To:-Test-controllers-with-Rails-(and-RSpec))

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`:

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

You can read more about OmniAuth support in the wiki:

- <https://github.com/heartcombo/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:

```
1 # Create a migration with the required fields
2 create_table :admins do |t|
3   t.string :email
4   t.string :encrypted_password
5   t.timestamps null: false
6 end
7
8 # Inside your Admin model
9 devise :database_authenticatable, :timeoutable
10
11 # Inside your routes
12 devise_for :admins
13
14 # Inside your protected controller
15 before_action :authenticate_admin!
16
17 # Inside your controllers and views
18 admin_signed_in?
19 current_admin
20 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.

Active Job Integration

If you are using Active Job to deliver Action Mailer 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.

```
1 def send_devise_notification(notification, *args)
2   devise_mailer.send(notification, self, *args).deliver_later
3 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 INFO 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`:

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

Other ORMs

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

Rails API Mode

Rails 5+ has a built-in API Mode which optimizes Rails for use as an API (only). Devise is *somewhat* able to handle applications that are built in this mode without additional modifications in the sense that it should not raise exceptions and the like. But some issues may still arise during `development/testing`, as we still don't know the full extent of this compatibility. (For more information, see issue #4947)

Supported Authentication Strategies API-only applications don't support browser-based authentication via cookies, which is devise's default. Yet, devise can still provide authentication out of the box in those cases with the `http_authenticatable` strategy, which uses HTTP Basic Auth and authenticates the user on each request. (For more info, see this wiki article for How To: Use HTTP Basic Authentication)

The devise default for HTTP Auth is disabled, so it will need to be enabled in the devise initializer for the database strategy:

```
1 config.http_authenticatable = [:database]
```

This restriction does not limit you from implementing custom warden strategies, either in your application or via gem-based extensions for devise. A common authentication strategy for APIs is token-based authentication. For more information on extending devise to support this type of authentication and others, see the wiki article for Simple Token Authentication Examples and alternatives or this blog post on Custom authentication methods with Devise.

Testing API Mode changes the order of the middleware stack, and this can cause problems for `Devise::Test::IntegrationHelpers`. This problem usually surfaces as an `undefined method `[]=' for nil:NilClass` error when using integration test helpers, such as `#sign_in`. The solution is simply to reorder the middlewares by adding the following to `test.rb`:

```
1 Rails.application.config.middleware.insert_before Warden::Manager,  
  ActionController::Cookies  
2 Rails.application.config.middleware.insert_before Warden::Manager,  
  ActionController::Session::CookieStore
```

For a deeper understanding of this, review this issue.

Additionally be mindful that without views supported, some email-based flows from Confirmable, Recoverable and Lockable are not supported directly at this time.

Additional information

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/wardencommunity/warden>

Contributors

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

<https://github.com/heartcombo/devise/graphs/contributors>

License

MIT License. Copyright 2020-2024 Rafael França, Leonardo Tegov, Carlos Antônio da Silva. Copyright 2009-2019 Plataformatec.

The Devise logo is licensed under Creative Commons Attribution-NonCommercial-NoDerivatives 4.0 International License.