**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 Userbut 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 (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'):

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\_pathwill 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\_inamong 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**

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

class ApplicationController < ActionController::Base

before\_action :configure\_permitted\_parameters, if: :devise\_controller?

protected

def configure\_permitted\_parameters

devise\_parameter\_sanitizer.permit(:sign\_up, keys: [:first\_name, :last\_name, address\_attributes: [:country, :state, :city, :area, :postal\_code]])

end

end

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

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:

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:

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

1. Tell the router to use this controller:

devise\_for :users, controllers: { sessions: 'users/sessions' }

1. 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.
2. Finally, change or extend the desired controller actions.

You can completely override a controller action:

class Users::SessionsController < Devise::SessionsController

def create

# custom sign-in code

end

end

Or you can simply add new behaviour to it:

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', confirmation: 'verification', unlock: 'unblock', registration: 'register', sign\_up: 'cmon\_let\_me\_in' }

Be sure to check devise\_for [documentation](http://www.rubydoc.info/github/plataformatec/devise/master/ActionDispatch/Routing/Mapper%3Adevise_for) 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:

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:

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.

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):

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:

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:

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.

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:

sign\_in users(:bob)

sign\_in users(:bob), scope: :admin

sign\_out :user

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

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:

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:

# 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.

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](http://rubydoc.info/github/plataformatec/devise/master/Devise/Models/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>