This repository | Search   Pull requests Issues Marketplace Gist

splashinn / **Cooper-Aerial-PM-API** Private   Unwatch 2 Star 0 Fork 0

Code Issues 0 Pull requests 0 Projects 0 Wiki Insights

Cooper Aerial API for Project Manager using Devise Auth Token

67 commits **6** branches **0** releases **1** contributor

Your recently pushed branches:

costProposalUser (less than a minute ago)   Compare & pull request

Branch: **master** New pull request   Create new file Upload files Find file Clone or download

splashinn cleaned up gemfile   Latest commit `f98cf8f` 29 days ago

app	uncommented line to check for authentication before logging in	29 days ago
bin	initial commit - setup Rails API w/ authentication	2 months ago
config	added model config for users - rails_admin.rb	a month ago
db	added in test user for authorization	a month ago
lib/tasks	initial commit - setup Rails API w/ authentication	2 months ago
log	initial commit - setup Rails API w/ authentication	2 months ago
public	initial commit - setup Rails API w/ authentication	2 months ago
spec	added cost proposal model and controller, updated routes file	2 months ago
tmp	initial commit - setup Rails API w/ authentication	2 months ago
.gitignore	initial commit - setup Rails API w/ authentication	2 months ago
.rspec	initial commit - setup Rails API w/ authentication	2 months ago
Gemfile	cleaned up gemfile	29 days ago
Gemfile.lock	added rails admin gem	a month ago
README.md	start of day - update ReadMe with endpoints	a month ago
Rakefile	initial commit - setup Rails API w/ authentication	2 months ago
config.ru	initial commit - setup Rails API w/ authentication	2 months ago

README.md

Cooper Aerial Project Management API

- Backend for the project management client-side
- Authentication is setup, but authenticating the user is disabled for development purposes.
- API live at `https://cuddly-robot-api.herokuapp.com/`
- 2 week sprint June 5 - June 19
  - [x] Continue testing, work on the authentication part
  - [ ] v2 for API (this will be the version put into production)
- Pagination is set to 15 results per page

- Admin dashboard at `/admin` . Using Administrate for admin dashboard
  - [x] Add authentication to admin section. Using Clearance gem for this.
- [x] User roles w/ CanCanCan
  - [x] Have roles defined in `models/ability.rb` . Need to authorized in controllers
- [x] Add user_id to opportunities for tracking
- Rails Admin dashboard at `/pm_admin` . Using rails_admin gem for this.
- *There are currently two different admin dashboards for the purpose of testing one against another to see which fits our needs best.*

Endpoints

`/v1/companies`

`/v1/offices`

`/v1/contacts`

`/v1/opportunities`

`/v1/film_specs`

`/v1/film_quotes`

`/v1/cost_proposals`

`/v1/users`

`/admin` (Administrate dashboard)

`/pm_admin` (RailsAdmin dashboard)

How to Install

I have included a step by step guide in this README on how I set this up. But if you just want to get up and running, here are the steps:

1. Clone this repository
2. Inside your database.yml rename the database to whatever you want, otherwise it will just be named `ca_pm_api_development` , `ca_pm_api_test` , `ca_pm_api_production` for each environment respectively.
3. Rename the top level module inside `application.rb` to whatever you want otherwise it will be named `CooperApi`
4. Run `bundle install`
5. Run `bundle exec rake db:create`
6. Run `bundle exec rake db:migrate; RAILS_ENV=test bundle exec rake db:migrate`
7. Boot up the server with `rails s` and navigate to localhost:3000 to see your new rails application
8. Check out the admin dashboard at `/admin` (thanks Thoughtbot for Administrate!)

If you are interested in how this app was setup and what is included read on.

Setting up a Rails 5 API Only Application: Step by Step Guide

Introduction

This guide shows you how I setup my default Rails architecture, now updated for Rails 5. Our Rails layer will do nothing but provide an API which serves JSON, and process background jobs. Here are some of the things we will be setting up:

- Token based authentication with the `devise_token_auth` gem.
- A namespaced API
- Serialization with Active Model Serializers
- Testing with rspec and factory girl
- Useful rack middleares like rack-attack for rate limiting and throttling with rack-cors.
- Local development niceties such as mailcatcher, pry/pry-nav, and useful git commit hooks
- Background jobs with Sidekiq
- Caching with Redis

Initial Setup

Install Rails

`gem install rails`

Create a new API only project with Postgres as the selected database

`rails new project_name --api --database=postgresql`

Setup the database

```
cd project_name
rails db:setup
rails db:migrate
```

Confirm that the server is working

Boot up the server with the command `rails s` inside the project directory. Visit localhost:3000 and you should see a page that greets you with the message "Yay! You're on Rails!"

Add rspec, factory girl, and pry to test and development

Lets go ahead and a add a couple gems that will be very useful for testing and development. I prefer to use rspec for testing purposes, but some of you might wan't to stick with minitest, if that's the case you can just ignore the `rspec-rails` gem. If not add `rspec-rails` and `factory_girl_rails` . FactoryGirl will let us easily create and mock test objects. The gem `pry-rails` provides a very handy interactive console when using rails c, `pry-byebug` gives you powerful break points and `pry-stack_expolorer` rounds out the package with a very solid stack explorer.

Open Gemfile, add the following. You can get rid of whatever else was in the :development, :test group

```
group :development, :test do
  gem 'rspec-rails', '3.1.0'
  gem 'factory_girl_rails'
  gem 'pry-rails'
  gem 'pry-byebug'
  gem 'pry-stack_explorer'
end
```

Install the gems by running `bundle install` .

Finish the rspec installation by running the command `rails generate rspec:install` .

You can go ahead and remove the test directory since we will be using /spec

`rm -rf test`

Now would be a good time to put everything under version control

```
git init
git add --all
git commit -m 'Initial Commit'
```

Set up your git remotes if you would like. I trust you know how to do this.

Setting up token based authentication

Add `devise_token_auth` and `omniauth` gem to the Gemfile:

```
gem 'devise_token_auth'
gem 'omniauth'
```

Run `bundle install` to install the gems.

Generating the user model with devise concerns and routes

`rails g devise_token_auth:install User auth`

The devise generator accepts two arguments the user class (in our case User) which is the name of the class to use for user authentication. The second argument is the mount path (in our case auth) which is path at which to mount the authentication routes.

The generator will create an initializer at `config/initializers/devise_token_auth.rb`, a User model at `app/models/user.rb`, a concern will be included in `app/controllers/application_controller.rb`, routes defined in `config/routes.rb`, and a migration to create the `users` table. You may want to tweak the migration to add columns to your liking. See the github page for more information.

Run `rails db:migrate` to add the users table as defined by the devise migrations.

Now run `rails routes` to see the authentication routes that have been setup.

Mailer Configuration

Since devise uses mailers for account registration, deletion, password changing, etcetera, we need to configure our development environment to send mail. The first thing is we need to add the following to `config/initializers/devise.rb`

```
Devise.setup do |config|
  # Using rails-api, tell devise to not use ActionDispatch::Flash
  # middleware b/c rails-api does not include it.
  config.navigational_formats = [:json]
end
```

The comment explains it all. We wan't devise to ignore the `ActionDispatch::Flash` middleware since it is not bundled in the Rails API mode middleware stack.

Next Add the following to `config/environments/development.rb`

```
config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = { address: 'localhost', port: 1025 }
config.action_mailer.default_url_options = { :host => 'localhost', port: '3000' }
```

Token based authentication in action

Let's see some token authentication in action. Boot up your rails server with the `rails s` command.

Examining the output of `rails routes` we can see that in order to regiser a new user we have to submit a POST request to the `/auth` routes. Let's do that using `curl`:

```
curl --data
"email=test@test.com&password=testpassword1&password_confirmation=testpassword1&confirm_success_url=http://l
ocalhost:3000" http://localhost:3000/auth
```

You should see a response that looks like the following:

```
{"status":"success","data":
{"id":1,"provider":"email","uid":"test@test.com","name":null,"nickname":null,"image":null,"email":"test@test
.com","created_at":"2016-07-23T23:16:10.064Z","updated_at":"2016-07-23T23:16:10.064Z"}}
```

If you examine your server logs, you should see that the action was processed and an email response was generated:

```
 Started POST "/auth" for ::1 at 2016-07-23 19:16:09 -0400
Processing by DeviseTokenAuth::RegistrationsController#create as */*
  Parameters: {"email"=>"test@test.com", "password"=>"[FILTERED]", "password_confirmation"=>"[FILTERED]", '
-- snip --
Devise::Mailer#confirmation_instructions: processed outbound mail in 170.2ms
Sent mail to test@test.com (8.4ms)
Date: Sat, 23 Jul 2016 19:16:10 -0400
From: please-change-me-at-config-initializers-devise@example.com
Reply-To: please-change-me-at-config-initializers-devise@example.com
To: test@test.com
Message-ID:
Subject: Confirmation instructions
Mime-Version: 1.0
Content-Type: text/html;
 charset=UTF-8
Content-Transfer-Encoding: 7bit
client-config: default
redirect-url: http://localhost:3000

<p>Welcome test@test.com!</p>

<p>You can confirm your account email through the link below: </p>

<p><a href="http://localhost:3000/auth/confirmation?config=default&confirmation_token=c9UbzsguMJ3BZkKZYC1A&

Completed 200 OK in 666ms (Views: 0.6ms | ActiveRecord: 35.4ms)
```

Go ahead and copy the link into your broswer. Then, boot up a rails console using the `rails c` command, and run `User.first`. This should return the new test user you just created.

Setting up mailcatcher

Reading the server logs to see what emails were rendered is not a great dev experience. Let's set up mailcatcher so that we can intercept emails delivered locally.

Install mailcatcher with the command `gem install mailcatcher`. Start mailcatcher by running `mailcatcher`. This will boot it up as a background daemon. Go to http://localhost:1080/ to see the mailcatcher UI.

Now lets try registering a new user.

```
curl --data
"email=test2@test.com&password=testpassword1&password_confirmation=testpassword1&confirm_success_url=http://
localhost:3000" http://localhost:3000/auth
```

This time we should be able to see the email in the mailcatcher UI.

Serialization with ActiveModelSerializers

Our API will be responsible for rendering JSON responses, and that's about it. In order to build these JSON responses, we will use a gem called ActiveModelSerializers.

Install ActiveModelSerializers

Add `gem 'active_model_serializers', '~> 0.10.0'` to the Gemfile and run `bundle install`.

Add a config file in the location `config/initializers/active_model_serializer.rb` with the code:

```
ActiveModelSerializers.config.adapter = :json_api
```

Generating a user serializer

Run the command `rails g serializer user`. In the generated user serializer add the attributes :name, and :email so that the final code looks like this:

```
class UserSerializer < ActiveModel::Serializer
  attributes :id, :name, :email
end
```

Setup an endpoint for serving user objects

Now that we have a user serializer, let's actually setup some endpoints that will serve us some user objects. This will be the begining of our API.

Let's start by setting up the routes. We will scope these api routes to a module 'api' and then provide a v1 namespace for all our routes. Here is what we will add to the routes file:

```
scope module: 'api' do
  namespace :v1 do
    resources :users, only: [:index, :show]
  end
end
```

Run `rails routes` to see the routes that have been set up. You should see something like this:

```
v1_users GET      /v1/users(.:format)                    api/v1/users#index
v1_user GET      /v1/users/:id(.:format)                api/v1/users#show
```

Now let's add an api controller from which all our other controllers in the API will inherit. The file should live in `app/controllers/api/v1/api_controller.rb`.

```
module Api::V1
  class ApiController < ApplicationController
    # before_action :authenticate_user!
  end
end
```

Next add a users controller, with two actions index and show: `app/controllers/api/v1/users_controller.rb`

```
module Api::V1
  class UsersController < ApiController

    # GET /v1/users
    def index
      render json: User.all
    end

    # GET /v1/users/{id}
    def show
      render json: User.find(params[:id])
    end

  end
end
```

Now lets confirm that our controllers, routes, and serialization is working as expected:

Run `curl http://localhost:3000/v1/users` , which will be routed to our index action, and you should see a response that looks something like this:

```
{"data":[{"id":"1","type":"users","attributes":{"name":null,"email":"test@test.com"}},{"id":"2","type":"use
```

Run `curl http://localhost:3000/v1/users/1` , which will be routed to our show action, and you should see a response like this:

```
{"data":{"id":"1","type":"users","attributes":{"name":null,"email":"test@test.com"}}}
```

Great! We now have a bare minimum working API that serves some JSON. We can see that the attributes being returned are only the ones we specified in the serializer.

Authenticating our API

We setup devise, but we are not actually authenticating anywhere in our API! Let's do that now.

Devise makes it easy for us. All we have to do is add the following `before_action` to `inapp/controllers/api/v1/api_controller.rb` :

```
before_action :authenticate_user!
```

Since we are poking around in our base api controller, and since we will exclusively be serving JSON to our clients, we can add the following to api controller:

```
respond_to :json
```

Great! Now if we try to hit the users controller with the same curl commands we used above we should get an authentication error. Let's test it out. Run `curl http://localhost:3000/v1/users/1` , and you should get the response

```
{"errors":["Authorized users only."]}
```

Using tokens for authentication

When using token based authentication, we have to generate a token for every single request. Tokens are invalidated after each request to the API. During each request, a new token is generated. The access-token header that should be used in the next request is returned in the access-token header of the response to the previous request. For more information see the devise_token_auth documentation.

When we run `rails routes` we should see a devise route that looks like this:

```
user_session POST    /auth/sign_in(.:format)                devise_token_auth/sessions#create
```

This is our login route. It return the initial auth token for when a user signs in. Let's test that by sending a post request to the endpoint with the appropriate payload:

```
curl -XPOST -v -H 'Content-Type: application/json' localhost:3000/auth/sign_in -d '{"email": "test@test.com
```

We should get a response that looks like this:

```
* Connected to localhost (::1) port 3000 (#0)
> POST /auth/sign_in HTTP/1.1
-- snip --
< Content-Type: application/json; charset=utf-8
< access-token: Avj8j2wQ4JAlFUDuPbS3fQ
< token-type: Bearer
< client: r4Pn4MLXvpCFTkwSc0HD7w
< expiry: 1470579487
< uid: test@test.com
-- snip --
* Connection #0 to host localhost left intact
{"data":{"id":1,"email":"the.kyle.maune@gmail.com","provider":"email","uid":"test@test.com","name":null,"ni
```

As you can see, we get the access-token, client, and uid in the headers of the response. The JSON response provides us with some attributes on the user object. Since this is a default devise route, it does not go through the user serializer we setup.

Now we can use the validate_token route to confirm that our token is, well, valid:

```
curl -XGET -v -H 'Content-Type: application/json' -H 'access-token: Avj8j2wQ4JAlFUDuPbS3fQ' -H 'client: r4P
```

The success response:

```
* Connected to localhost (::1) port 3000 (#0)
> GET /auth/validate_token HTTP/1.1
-- snip --
< Content-Type: application/json; charset=utf-8
< access-token: 2J6ygFVQrYHGy6aSH25D_g
< token-type: Bearer
< client: r4Pn4MLXvpCFTkwSc0HD7w
< expiry: 1470579646
< uid: test@test.com
-- snip --
{"success":true,"data":{"id":1,"provider":"email","uid":"test@test.com","name":null,"nickname":null,"image"
```

You can see the new access was generated and sent back in the headers of the previous response. So let's hit the API with the token from the last request!

```
curl -XGET -v -H 'Content-Type: application/json' -H 'access-token: 2J6ygFVQrYHGy6aSH25D_g' -H 'client: r4P
```

And the sucessful response should look as follows:

```
{"data":[{"id":"2","type":"users","attributes":{"name":null,"email":"test@test.com"}},{"id":"1","type":"use
```

Fantastic. At this point we have a bare minimum rails 5 api. Our routes are namespaced, users are authenticated with token based auth, and can sign up with email. We serialize our responses using `active_model_serializers`. This is most of what we need to get up and running.

Setting Up Specs

We sent some time setting up rspec and some useful debugging tools like pry-nav and factory girl, but we haven't actually written any tests yet. Why don't we go ahead and set up our first spec.

First we need to set up factory girl. At the top of our `spec/spec_helper.rb` we need to explicitly require it as follows: `require 'factory_girl_rails'` . Then to gain access to the FactoryGirl DSL inside our specs we need to add the following line `config.include FactoryGirl::Syntax::Methods` within our config block. Then inside of our `.rspec` file we need to add the following line `--require rails_helper` . We can go ahead and get rid of the line `--require spec_helper` , since the `rails_helper` includes the `spec_helper` . If we don't do this RSpec Cannot find the Controllers and throws an Uninitialized Constant error. See here for more information.

Now lets set up a very simple test for the users controller in the location `spec/controllers/api/v1/users_controller_spec.rb` .

```
require 'spec_helper'

describe Api::V1::UsersController do
  describe "GET #show" do
    before(:each) do
      @user = FactoryGirl.create :user
      auth_headers = @user.create_new_auth_token
      request.headers.merge!(auth_headers)
      get :show, id: @user.id
    end

    it 'responds with 200 status code' do
      expect(response.code).to eq('200')
    end
  end
end
```

As you can see in the before block we hadd to generate an auth token for our user, and merge it into the request headers. See here for more information. Since you will likely be doing this in all your controller tests, it might be useful to abstract this to a convienience method inside `spec_helper` , for now I will leave it.

When I tried running the test by executing `rspec spec/controllers/api/v1/users_controller_spec.rb` I got the following error:

```
Devise::MissingWarden:
       Devise could not find the `Warden::Proxy` instance on your request environment.
       Make sure that your application is loading Devise and Warden as expected and that the `Warden::Manag
       If you are seeing this on one of your tests, ensure that your tests are either executing the Rails m
```

What a useful error message! All I had to do was add the devise test helpers to `rails_helper.rb` inside the config block: `config.include Devise::Test::ControllerHelpers, type: :controller` .

Now running the test should pass!

Let's flesh this test out by adding another expectation for checking the response body:

```
it "returns the serialized user attributes" do
  expect(JSON.parse(response.body)['data']['attributes']).to eq({"name"=>"John Doe", "email"=>"test@test.cc
end
```

And now let's add some tests for the index action:

```
describe Api::V1::UsersController do

  -- snip --

  describe 'GET #index' do
    before(:each) do
      @user = FactoryGirl.create :user
      auth_headers = @user.create_new_auth_token
      request.headers.merge!(auth_headers)
      get :index, id: @user.id
    end

    it 'responds with 200 status code' do
      expect(response.code).to eq('200')
    end

    it 'returns the serialized user attributes' do
      expect(JSON.parse(response.body)['data'].length).to eq(1)
      expect(JSON.parse(response.body)['data'].first['attributes']).to eq({'name'=>'John Doe', 'email'=>'te
    end
  end
end
```

Rack Middlewares

Use Rack CORS to allows cross origin resource sharing

If your API will be public, you will need to enable Cross-Origin Resource Sharing. A resource makes a cross-origin HTTP request when it requests a resource from a different domain than the one which the first resource itself serves.

Add `gem 'rack-cors'` to the Gemfile and run `bundle install`

Then add this to `application.rb` :

```
  config.middleware.use Rack::Cors do
    allow do
      origins '*'
      resource '*',
        :headers => :any,
        :expose  => ['access-token', 'expiry', 'token-type', 'uid', 'client'],
        :methods => [:get, :post, :options, :delete, :put]
    end
  end
```

WARNING: This will whitelist requests from any domain. Make sure to whitelist only the needed domains.

Rate limiting/throttling, blacklisting/whitelisting of IP's with rack-attack

`rack-attack` is a middleware that allows us rate limit, throttle, white list, and black list IP's. Trust me, you want this.

Add `gem 'rack-attack'` to the Gemfile and then run `bundle install` . Inside of `config/application.rb` add `config.middleware.use Rack::Attack` .

Next, create a new initializer `config/initializers/rack_attack.rb` . I plucked the basic config from this super helpful guide

```
  class Rack::Attack

    # `Rack::Attack` is configured to use the `Rails.cache` value by default,
    # but you can override that by setting the `Rack::Attack.cache.store` value
    Rack::Attack.cache.store = ActiveSupport::Cache::MemoryStore.new

    # Allow all local traffic
    whitelist('allow-localhost') do |req|
      '127.0.0.1' == req.ip || '::1' == req.ip
    end

    # Allow an IP address to make 5 requests every 5 seconds
    throttle('req/ip', limit: 5, period: 5) do |req|
      req.ip
    end

    # Send the following response to throttled clients
    self.throttled_response = ->(env) {
      retry_after = (env['rack.attack.match_data'] || {})[:period]
      [
        429,
        {'Content-Type' => 'application/json', 'Retry-After' => retry_after.to_s},
        [{error: "Throttle limit reached. Retry later."}.to_json]
      ]
    }
  end
```

Depending on your needs, the 5 requests every 5 seconds may be a little too strict. For instance, if you are building a web client that makes several async requests, you may want to loosen the limit a bit.

Cacheing with Redis

You may have noticed that we are using the `ActiveSupport::Cache::MemoryStore.new` to back rack-attack. This is no good! This particular store puts everything into memory in the same process, so while the cache might be blazingly fast, it disappears when the process dissapears, and it's cache data cannot be shared across processes.

I like to use Redis for cacheing. Some prefer Memcached. Both have their merits. But since we will be using sidekiq for job processing which requires Redis, and we may want to use action-cable, we might as well stick with Redis.

First add `gem 'redis-rails'` to your Gemfile, and then run `bundle install`. Install and start redis using instructions here.

In `development.rb` get rid of the block of code that looks like this:

```
# Enable/disable caching. By default caching is disabled.
if Rails.root.join('tmp/caching-dev.txt').exist?
  config.action_controller.perform_caching = true

  config.cache_store = :memory_store
  config.public_file_server.headers = {
    'Cache-Control' => 'public, max-age=172800'
  }
else
  config.action_controller.perform_caching = false

  config.cache_store = :null_store
end
```

And then replace with this:

```
if ENV['REDIS_URL']
  config.action_controller.perform_caching = true
  config.cache_store = :redis_store, ENV['REDIS_URL']
else
  config.action_controller.perform_caching = false
  config.cache_store = :null_store
end
```

This way we can enable cacheing locally if we wish to test it. Usually in development cacheing is turned off, but if you want to enable it all you have to do is set the environment variable `REDIS_URL` to `redis://localhost:6379`.

In `production.rb` add

```
config.cache_store = :redis_store, ENV['REDIS_URL']
```

Background Job Processing with Sidekiq

This is sort of optional, but Sideiq is my go to for background job processing in ruby. It is extremely robust and efficient.

Getting Started

Add `gem 'sidekiq'` to the Gemfile and bundle.

Create an `app/workers` directory, and try running a sample job:

```
class SampleWorker
  include Sidekiq::Worker
  def perform()
    # do something
  end
end
```

You can kick it off in your console with `SampleWorker.perform_async` which will return a job ID.

Caching

http://www.victorareba.com/tutorials/speed-your-rails-app-with-model-caching-using-redis
https://medium.com/ruby-on-rails/easy-caching-with-rails-4-heroku-redis-5fb36381628#.o1a12hbzb

Nice to Haves

Setup Rack-attack to use Default Cache

Now that we have enabled redis we can enable rack-attack to use the default cache. In `config/initializers/rack_attack.rb` remove the line `Rack::Attack.cache.store = ActiveSupport::Cache::MemoryStore.new`. By default, Rack::Attack will use whichever caching backend is configured as Rails.cache.

Set up a git commit hook to stop from committing binding.pry

I use binding.pry all over the place, and sometimes I forget to take it out before commiting. Here's a little pre-commit hook taken from the following gist to prevent that from happening.

Inside the `/.git/hooks/pre-commit` file

```
#!/bin/bash

FILES_PATTERN='\.(rb)(\..+)?$'
FORBIDDEN='binding.pry'

git diff --cached --name-only | \
    grep -E $FILES_PATTERN | \
    GREP_COLOR='4;5;37;41' xargs grep --color --with-filename -n $FORBIDDEN && \
    echo 'COMMIT REJECTED' && \
    exit 1

exit 0
```

Make it executable by running `chmod +x /.git/hooks/pre-commit`

Now if you try to commit a binding pry it will reject the commit! Awesome!

Get rid of the "Yay! You're on Rails!" splash page

I like rails a whole bunch. But I don't need the world to know that when they go to my root url. For now let's just replace that with an empty JSON response.

Inside our `routes.rb` file lets add a root:

```
root to: 'home#show'
```

And then add the following controller: `app/controllers/home_controller.rb` with

```
class HomeController < ApplicationController
  def show
    render json: {}
  end
end
```

Change the sent from email in devise mailers

You may have noticed the "please-change-me-at-config-initializers-devise@example.com" address in the from field of our devise mailers. We can change that by specifying the `mailer_sender` inside the devise config file:

```
Devise.setup do |config|
  --snip--
  config.mailer_sender = "support@myapp.com"

end
```