RoR - Action Mailer

Overview

Mailers are similar to controllers in layout, having their own actions, views, instance variables, access to params hash, and ability to use before\_actions, layouts, and partials.

Setup

The application mailer can be used to set a default email address to send from using:

default from: 'admin@example.com'

layout 'mailer'

The host for Action Mailer is set up in the application configuration for each environment, development, test, and production:

Host locations:

config.action\_mailer.default\_url\_options = { host: 'example.com' }

config.action\_mailer.asset\_host = 'https://example.com'

Delivery method: config.action\_mailer.delivery\_method

* :smtp - configured in smtp\_settings
* :sendmail - sendmail\_settings
* :file - saved to file in file\_settings
* :test - saved to ActionMailer::Base.deliveries array

SMTP settings: config.action\_mailer.smtp\_settings

* address: - remote mailer address, default localhost
* port: - port for remote mailer, default 25
* domain: - HELO domain
* user\_name: - mail authentication
* password: - mail authentication
* authentication: - Type of authentication :plain/:login/:cram\_md5
* enable\_starttle\_auto: - Detects STARTTLS, default true
* openssl\_verify\_mode: - TLS certificate verification setting ('none' or 'peer')
* :ssl/:tls - Enables the SMTP connection to use SMTPS

Sendmail settings: config.action\_mailer.sendmail\_settings

* location: location of sendmail executable, defualt /usr/sbin/sendmail
* arguments: commandline arugments for sendmail

Delivery errors: config.action\_mailer.raise\_delivery\_errors

* set to true to raise error for immediate delivery mail, default true

Perform deliveries: config.action\_mailer.perform\_deliveries

* boolean - default true

Level of logging: config.action\_mailer.logger

* nil - no logging

Creating a New Mailer

Generate a new mailer and associated views and tests using the generate command:

rails g mailer PasswordMailer

# app/mailers/password\_mailer.rb

class UserMailer < ApplicationMailer

end

An example action would be a welcome email on signup:

def welcome\_email

@user = params[:user]

@url = 'http://example.com/login'

mail(to: @user.email, subject: 'Welcome to My Awesome Site')

end

Creating the Views

The welcome\_email action would then use the template view for the mailer named 'welcome\_email.hmtl.erb', which would be set up like:

<!DOCTYPE html>

<html>

<head>

<meta content='text/html; charset=UTF-8' http-equiv='Content-Type' />

</head>

<body>

<h1>Welcome to example.com, <%= @user.name %></h1>

<p>

You have successfully signed up to example.com,

your username is: <%= @user.login %>.<br>

</p>

<p>

To login to the site, just follow this link: <%= @url %>.

</p>

<p>Thanks for joining and have a great day!</p>

</body>

</html>

Since not clients want html emails, it is best practice to also create a text based version named: 'welcome\_email.text.erb'. Action mailer will then automatically combine the two emails for users to select from.

Different template views can be used by defining them in the template path header of mail:

mail(to: @user.email,

subject: 'Welcome to My Awesome Site',

template\_path: 'notifications',

template\_name: 'another')

Calling the Mailer

Mailers can be called in controllers using their class name, passing variables, then the action (email type) which is required to be sent.

# Tell the UserMailer to send a welcome email after save

UserMailer.with(user: @user).welcome\_email.deliver\_later

The deliver later option allows emails to be sent asynchronously and outside the request repsonse cycle. The job will be placed under the mailers queue, for production setups a Active Job adapter which has a persistent backend (Sidekiq, Resque) will be required to stop restarts deleting all unsent mail. Alternatively, if email is in a cronjob the '.deliver\_now' option can be used.

A specific time or delay can be given for the async delivery using the following syntax:

.deliver\_later(wait: 1.hour)

.deliver\_later(wait\_until: 5.days.from\_now)

Variables are passed through as a key value pair, and can be accessed in the mailer as params. E.g. if Mailer(user: user) is passed, it can be called in the mailer using params[:user].

Headers and Attachments

There are three methods which can build most emails:

* headers[:field\_name] = value - sets header value in email
* attachments['file-name.jpg'] = File.read('file-name.jpg') - adds attachment
* mail - send the email, headers can be passed as a hash

A typical header such as email subject can be add in the mail params:

mail(to: @user.email, subject: "New User Signup: #{@user.email}")

Inline attachments such as images can be added using the inline method, then called in a view using the image tag:

def welcome

attachments.inline['image.jpg'] = File.read('/path/to/image.jpg')

end

<%= image\_tag attachments['image.jpg'].url %>

Images can also be added remotely, and downloaded by the clients MUA using the image\_tag:

<%= image\_tag 'image.jpg' %>

Emails With a Name

Giving the email a name to mask the email address can be performed by using the following format:

"<name>" <<email-address>>

Therefore place it in the two header using:

email\_with\_name = %("#{@user.name}" <#{@user.email}>)

mail(to: email\_with\_name, subject: 'Welcome to My Awesome Site')

Links

Links can be added normally or by using the helper link\_to, however \_path has to be replaced with \_url:

<%= link\_to 'welcome', welcome\_url %>

<%= url\_for(host: 'example.com',

controller: 'welcome',

action: 'greeting') %>

Testing Mailers

The purpose of testing mailers is to:

* make sure emails are being processed and sent - unit test mailer
* the email content is correct - unit test mailer/view
* the right emails are sent at the right time - functional test controller

First the environment must be set up to test:

config.action\_mailer.delivery\_method = :test

RSpec mailer specs are a thin wrapper for ActionMailer::TestCase, providing all behaviour and assertions that it provides along with RSpecs own expectations:

require "rails\_helper"

RSpec.describe Notifications, :type => :mailer do

describe "notify" do

let(:mail) { Notifications.signup }

it "renders the headers" do

expect(mail.subject).to eq("Signup")

expect(mail.to).to eq(["to@example.org"])

expect(mail.from).to eq(["from@example.com"])

end

it "renders the body" do

expect(mail.body.encoded).to match("Hi")

end

end

end

Functional test example for User model, based in user model unit tests:

it "sends invite email" do

expect { user.send\_invite }.to change { ActionMailer::Base.deliveries.count }.by(1)

end

Unit test example for mailer, based in mailer unit tests:

describe Notifier do

describe 'instructions' do

let(:user) { mock\_model User, name: 'Lucas', email: 'lucas@email.com' }

let(:mail) { Notifier.instructions(user) }

it 'renders the subject' do

expect(mail.subject).to eql('Instructions')

end

it 'renders the receiver email' do

expect(mail.to).to eql([user.email])

end

it 'renders the sender email' do

expect(mail.from).to eql(['noreply@company.com'])

end

it 'assigns @name' do

expect(mail.body.encoded).to match(user.name)

end

it 'assigns @confirmation\_url' do

expect(mail.body.encoded).to match("http://aplication\_url/#{user.id}/confirmation")

end

end

end

To reduce dependencies in unit tests Action Mailer can also be doubled in RSpec using instance doubles:

user = User.first

message\_delivery = instance\_double(ActionMailer::MessageDelivery)

expect(UsersMailer).to receive(:welcome\_email).with(user.id).and\_return(message\_delivery)

expect(message\_delivery).to receive(:deliver\_later)

mail = UsersMailer.welcome\_email(user.email)

mail.deliver\_later