

Action Mailer Basics

Ruby on Rails

Action Mailer Basics

[Action Mailer Basics](#)

[1 Introduction](#)

[2 Sending Emails](#)

[2.1 Walkthrough to Generating a Mailer](#)

[2.1.1 Create the Mailer](#)

[2.1.2 Edit the Mailer](#)

[2.1.3 Create a Mailer View](#)

[2.1.4 Calling the Mailer](#)

[2.2 Auto encoding header values](#)

[2.3 Complete List of Action Mailer Methods](#)

[2.3.1 Adding Attachments](#)

[2.3.2 Making Inline Attachments](#)

[2.3.3 Sending Email To Multiple Recipients](#)

[2.3.4 Sending Email With Name](#)

[2.4 Mailer Views](#)

[2.5 Action Mailer Layouts](#)

[2.6 Generating URLs in Action Mailer Views](#)

[2.6.1 generating URLs with url_for](#)

[2.6.2 generating URLs with named routes](#)

[2.7 Sending Multipart Emails](#)

[2.8 Sending Emails with Dynamic Delivery Options](#)

[2.9 Sending Emails without Template Rendering](#)

[3 Receiving Emails](#)

[4 Action Mailer Callbacks](#)

[5 Using Action Mailer Helpers](#)

[6 Action Mailer Configuration](#)

[6.1 Example Action Mailer Configuration](#)

[6.2 Action Mailer Configuration for Gmail](#)

[7 Mailer Testing](#)

[8 Intercepting Emails](#)

[Feedback](#)

Action Mailer Basics

This guide provides you with all you need to get started in sending and receiving emails from and to your application, and many internals of Action Mailer. It also covers how to test your mailers. After reading this guide, you will know:

- How to send and receive email within a Rails application.
- How to generate and edit an Action Mailer class and mailer view.
- How to configure Action Mailer for your environment.
- How to test your Action Mailer classes.

1 Introduction

Action Mailer allows you to send emails from your application using mailer classes and views. Mailers work very similarly to controllers. They inherit from `ActionMailer::Base` and live in `app/mailers`, and they have associated views that appear in `app/views`.

2 Sending Emails

This section will provide a step-by-step guide to creating a mailer and its views.

2.1 Walkthrough to Generating a Mailer

2.1.1 Create the Mailer

```
$ bin/rails generate mailer UserMailer
create  app/mailers/user_mailer.rb
create  app/mailers/application_mailer.rb
invoke  erb
create  app/views/user_mailer
create  app/views/layouts/mailer.text.erb
create  app/views/layouts/mailer.html.erb
invoke  test_unit
create  test/mailers/user_mailer_test.rb
create  test/mailers/previews/user_mailer_preview.rb
```

```
# app/mailers/application_mailer.rb
class ApplicationMailer < ActionMailer::Base
  default "from@example.com"
  layout 'mailer'
end
```

```
# app/mailers/user_mailer.rb
class UserMailer < ApplicationMailer
end
```

As you can see, you can generate mailers just like you use other generators with Rails. Mailers are conceptually similar to controllers, and so we get a mailer, a directory for views, and a test.

If you didn't want to use a generator, you could create your own file inside of `app/mailers`, just make sure that it inherits from `ActionMailer::Base`:

```
class MyMailer < ActionMailer::Base
end
```

2.1.2 Edit the Mailer

Mailers are very similar to Rails controllers. They also have methods called "actions" and use views to structure the content. Where a controller generates content like HTML to send back to the client, a Mailer creates a message to be delivered via email.

`app/mailers/user_mailer.rb` contains an empty mailer:

```
class UserMailer < ApplicationMailer
```

```
end
```

Let's add a method called `welcome_email`, that will send an email to the user's registered email address:

```
class UserMailer < ApplicationMailer
  default from: 'notifications@example.com'

  def welcome_email(user)
    @user = user
    @url = 'http://example.com/login'
    mail(to: @user.email, subject: 'Welcome to My Awesome Site')
  end
end
```

Here is a quick explanation of the items presented in the preceding method. For a full list of all available options, please have a look further down at the Complete List of Action Mailer user-settable attributes section.

- `default Hash` - This is a hash of default values for any email you send from this mailer. In this case we are setting the `:from` header to a value for all messages in this class. This can be overridden on a per-email basis.
- `mail` - The actual email message, we are passing the `:to` and `:subject` headers in.

Just like controllers, any instance variables we define in the method become available for use in the views.

2.1.3 Create a Mailer View

Create a file called `welcome_email.html.erb` in `app/views/user_mailer/`. This will be the template used for the email, formatted in HTML:

```
<!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>
```

Let's also make a text part for this email. Not all clients prefer HTML emails, and so sending both is

best practice. To do this, create a file called `welcome_email.text.erb` in `app/views/user_mailer/`:

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

```
You have successfully signed up to example.com,
your username is: <%= @user.login %>.
```

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

```
Thanks for joining and have a great day!
```

When you call the `mail` method now, Action Mailer will detect the two templates (text and HTML) and automatically generate a multipart/alternative email.

2.1.4 Calling the Mailer

Mailers are really just another way to render a view. Instead of rendering a view and sending out the HTTP protocol, they are just sending it out through the email protocols instead. Due to this, it makes sense to just have your controller tell the Mailer to send an email when a user is successfully created.

Setting this up is painfully simple.

First, let's create a simple `User` scaffold:

```
$ bin/rails generate scaffold user name email login
$ bin/rake db:migrate
```

Now that we have a user model to play with, we will just edit the `app/controllers/users_controller.rb` make it instruct the `UserMailer` to deliver an email to the newly created user by editing the `create` action and inserting a call to `UserMailer.welcome_email` right after the user is successfully saved.

Action Mailer is nicely integrated with Active Job so you can send emails outside of the request-response cycle, so the user doesn't have to wait on it:

```
class UsersController < ApplicationController
  # POST /users
  # POST /users.json
  def create
    @user = User.new(params[:user])

    respond_to do |format|
      if @user.save
        # Tell the UserMailer to send a welcome email after save
        UserMailer.welcome_email(@user).deliver_later

        format.html { redirect_to(@user, notice: 'User was successfully created.' }
        format.json { render json: @user, status: :created, location: @user }
      else

```

```

format.html { render action: 'new' }
format.json { render json: @user.errors, status: :unprocessable_entity }
end
end
end
end

```

Note: Active Job's default behavior is to execute jobs `:inline`. So, you can use `deliver_later` now to send emails, and when you later decide to start sending them from a background job, you'll only need to set up Active Job to use a queueing backend (Sidekiq, Resque, etc).

If you want to send emails right away (from a cronjob for example) just call `deliver_now`:

```

class SendWeeklySummary
  def run
    User.find_each do |user|
      UserMailer.weekly_summary(user).deliver_now
    end
  end
end

```

The method `welcome_email` returns a `ActionMailer::MessageDelivery` object which can then just be told `deliver_now` or `deliver_later` to send itself out. The `ActionMailer::MessageDelivery` object is just a wrapper around a `Mail::Message`. If you want to inspect, alter or do anything else with the `Mail::Message` object you can access it with the `message` method on the `ActionMailer::MessageDelivery` object.

2.2 Auto encoding header values

Action Mailer handles the auto encoding of multibyte characters inside of headers and bodies.

For more complex examples such as defining alternate character sets or self-encoding text first, please refer to the [Mail](#) library.

2.3 Complete List of Action Mailer Methods

There are just three methods that you need to send pretty much any email message:

- `headers` - Specifies any header on the email you want. You can pass a hash of header field names and value pairs, or you can call `headers[:field_name] = 'value'`.
- `attachments` - Allows you to add attachments to your email. For example, `attachments['file-name.jpg'] = File.read('file-name.jpg')`.
- `mail` - Sends the actual email itself. You can pass in headers as a hash to the mail method as a parameter, mail will then create an email, either plain text, or multipart, depending on what email templates you have defined.

2.3.1 Adding Attachments

Action Mailer makes it very easy to add attachments.

- Pass the file name and content and Action Mailer and the [Mail gem](#) will automatically guess the `mime_type`, set the encoding and create the attachment.

```
attachments['filename.jpg'] = File.read('/path/to/filename.jpg')
```

When the `mail` method will be triggered, it will send a multipart email with an attachment, properly nested with the top level being `multipart/mixed` and the first part being a `multipart/alternative` containing the plain text and HTML email messages.

Note: Mail will automatically Base64 encode an attachment. If you want something different, encode your content and pass in the encoded content and encoding in a `Hash` to the `attachments` method.

- Pass the file name and specify headers and content and Action Mailer and Mail will use the settings you pass in.

```
encoded_content = SpecialEncode(File.read('/path/to/filename.jpg'))
attachments['filename.jpg'] = {
  mime_type: 'application/x-gzip',
  encoding: 'SpecialEncoding',
  content: encoded_content
}
```

Note: If you specify an encoding, Mail will assume that your content is already encoded and not try to Base64 encode it.

2.3.2 Making Inline Attachments

Action Mailer 3.0 makes inline attachments, which involved a lot of hacking in pre 3.0 versions, much simpler and trivial as they should be.

- First, to tell Mail to turn an attachment into an inline attachment, you just call `#inline` on the `attachments` method within your Mailer:

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

- Then in your view, you can just reference `attachments` as a hash and specify which attachment you want to show, calling `url` on it and then passing the result into the `image_tag` method:

```
<p>Hello there, this is our image</p>
```

```
<%= image_tag attachments['image.jpg'].url %>
```

- As this is a standard call to `image_tag` you can pass in an options hash after the attachment URL as you could for any other image:

```
<p>Hello there, this is our image</p>
```

```
<%= image_tag attachments['image.jpg'].url, alt: 'My Photo', class: 'photos' %>
```

2.3.3 Sending Email To Multiple Recipients

It is possible to send email to one or more recipients in one email (e.g., informing all admins of a new signup) by setting the list of emails to the `:to` key. The list of emails can be an array of email addresses or a single string with the addresses separated by commas.

```
class AdminMailer < ActionMailer::Base
  default to: Proc.new { Admin.pluck(:email) },
          from: 'notification@example.com'

  def new_registration(user)
    @user = user
    mail(subject: "New User Signup: #{@user.email}")
  end
end
```

The same format can be used to set carbon copy (Cc:) and blind carbon copy (Bcc:) recipients, by using the `:cc` and `:bcc` keys respectively.

2.3.4 Sending Email With Name

Sometimes you wish to show the name of the person instead of just their email address when they receive the email. The trick to doing that is to format the email address in the format "Full Name <email>".

```
def welcome_email(user)
  @user = user
  email_with_name = %("#{@user.name}" <#{@user.email}>)
  mail(to: email_with_name, subject: 'Welcome to My Awesome Site')
end
```

2.4 Mailer Views

Mailer views are located in the `app/views/name_of_mailer_class` directory. The specific mailer view is known to the class because its name is the same as the mailer method. In our example from above, our mailer view for the `welcome_email` method will be in `app/views/user_mailer/welcome_email.html.erb` for the HTML version and `welcome_email.text.erb` for the plain text version.

To change the default mailer view for your action you do something like:

```
class UserMailer < ApplicationMailer
  default from: 'notifications@example.com'

  def welcome_email(user)
```

```

@user = user
@url = 'http://example.com/login'
mail(to: @user.email,
      subject: 'Welcome to My Awesome Site',
      template_path: 'notifications',
      template_name: 'another')
end
end

```

In this case it will look for templates at `app/views/notifications` with name `another`. You can also specify an array of paths for `template_path`, and they will be searched in order.

If you want more flexibility you can also pass a block and render specific templates or even render inline or text without using a template file:

```

class UserMailer < ApplicationMailer
  default from: 'notifications@example.com'

  def welcome_email(user)
    @user = user
    @url = 'http://example.com/login'
    mail(to: @user.email,
          subject: 'Welcome to My Awesome Site') do |format|
      format.html { render 'another_template' }
      format.text { render text: 'Render text' }
    end
  end
end
end

```

This will render the template `'another_template.html.erb'` for the HTML part and use the rendered text for the text part. The render command is the same one used inside of Action Controller, so you can use all the same options, such as `:text`, `:inline` etc.

2.5 Action Mailer Layouts

Just like controller views, you can also have mailer layouts. The layout name needs to be the same as your mailer, such as `user_mailer.html.erb` and `user_mailer.text.erb` to be automatically recognized by your mailer as a layout.

In order to use a different file, call `layout` in your mailer:

```

class UserMailer < ApplicationMailer
  layout 'awesome' # use awesome.(html|text).erb as the layout
end

```

Just like with controller views, use `yield` to render the view inside the layout.

You can also pass in a `layout: 'layout_name'` option to the render call inside the format block to specify different layouts for different formats:

```

class UserMailer < ApplicationMailer

```

```

def welcome_email(user)
  mail(to: user.email) do |format|
    format.html { render layout: 'my_layout' }
    format.text
  end
end
end
end

```

Will render the HTML part using the `my_layout.html.erb` file and the text part with the usual `user_mailer.text.erb` file if it exists.

2.6 Generating URLs in Action Mailer Views

Unlike controllers, the mailer instance doesn't have any context about the incoming request so you'll need to provide the `:host` parameter yourself.

As the `:host` usually is consistent across the application you can configure it globally in `config/application.rb`:

```

config.action_mailer.default_url_options = { host: 'example.com' }

```

Because of this behavior you cannot use any of the `*_path` helpers inside of an email. Instead you will need to use the associated `*_url` helper. For example instead of using

```

<%= link_to 'welcome', welcome_path %>

```

You will need to use:

```

<%= link_to 'welcome', welcome_url %>

```

By using the full URL, your links will now work in your emails.

2.6.1 generating URLs with `url_for`

You need to pass the `only_path: false` option when using `url_for`. This will ensure that absolute URLs are generated because the `url_for` view helper will, by default, generate relative URLs when a `:host` option isn't explicitly provided.

```

<%= url_for(controller: 'welcome',
            action: 'greeting',
            only_path: false) %>

```

If you did not configure the `:host` option globally make sure to pass it to `url_for`.

```

<%= url_for(host: 'example.com',
            controller: 'welcome',
            action: 'greeting') %>

```

Note: When you explicitly pass the `:host` Rails will always generate absolute URLs, so there is no

need to pass `only_path: false`.

2.6.2 generating URLs with named routes

Email clients have no web context and so paths have no base URL to form complete web addresses. Thus, you should always use the `"_url"` variant of named route helpers.

If you did not configure the `:host` option globally make sure to pass it to the url helper.

```
<%= user_url(@user, host: 'example.com') %>
```

2.7 Sending Multipart Emails

Action Mailer will automatically send multipart emails if you have different templates for the same action. So, for our UserMailer example, if you have `welcome_email.text.erb` and `welcome_email.html.erb` in `app/views/user_mailer`, Action Mailer will automatically send a multipart email with the HTML and text versions setup as different parts.

The order of the parts getting inserted is determined by the `:parts_order` inside of the `ActionMailer::Base.default` method.

2.8 Sending Emails with Dynamic Delivery Options

If you wish to override the default delivery options (e.g. SMTP credentials) while delivering emails, you can do this using `delivery_method_options` in the mailer action.

```
class UserMailer < ApplicationMailer
  def welcome_email(user, company)
    @user = user
    @url = user_url(@user)
    delivery_options = { user_name: company.smtp_user,
                        password: company.smtp_password,
                        address: company.smtp_host }

    mail(to: @user.email,
         subject: "Please see the Terms and Conditions attached",
         delivery_method_options: delivery_options)
  end
end
```

2.9 Sending Emails without Template Rendering

There may be cases in which you want to skip the template rendering step and supply the email body as a string. You can achieve this using the `:body` option. In such cases don't forget to add the `:content_type` option. Rails will default to `text/plain` otherwise.

```
class UserMailer < ApplicationMailer
  def welcome_email(user, email_body)
    mail(to: user.email,
         body: email_body,
         content_type: "text/html",
         subject: "Already rendered!")
  end
end
```

3 Receiving Emails

Receiving and parsing emails with Action Mailer can be a rather complex endeavor. Before your email reaches your Rails app, you would have had to configure your system to somehow forward emails to your app, which needs to be listening for that. So, to receive emails in your Rails app you'll need to:

- Implement a `receive` method in your mailer.
- Configure your email server to forward emails from the address(es) you would like your app to receive to `/path/to/app/bin/rails runner 'UserMailer.receive(STDIN.read)'`.

Once a method called `receive` is defined in any mailer, Action Mailer will parse the raw incoming email into an email object, decode it, instantiate a new mailer, and pass the email object to the mailer `receive` instance method. Here's an example:

```
class UserMailer < ApplicationMailer
  def receive(email)
    page = Page.find_by(address: email.to.first)
    page.emails.create(
      subject: email.subject,
      body: email.body
    )

    if email.has_attachments?
      email.attachments.each do |attachment|
        page.attachments.create({
          file: attachment,
          description: email.subject
        })
      end
    end
  end
end
```

4 Action Mailer Callbacks

Action Mailer allows for you to specify a `before_action`, `after_action` and `around_action`.

- Filters can be specified with a block or a symbol to a method in the mailer class similar to controllers.
- You could use a `before_action` to populate the mail object with defaults, `delivery_method_options` or insert default headers and attachments.
- You could use an `after_action` to do similar setup as a `before_action` but using instance variables set in your mailer action.

```
class UserMailer < ApplicationMailer
  after_action :set_delivery_options,
              :prevent_delivery_to_guests,
              :set_business_headers

  def feedback_message(business, user)
    @business = business
    @user = user
    mail
  end

  def campaign_message(business, user)
    @business = business
    @user = user
  end

  private

  def set_delivery_options
    # You have access to the mail instance,
    # @business and @user instance variables here
    if @business && @business.has_smtp_settings?
      mail.delivery_method.settings.merge!(@business.smtp_settings)
    end
  end

  def prevent_delivery_to_guests
    if @user && @user.guest?
      mail.perform_deliveries = false
    end
  end

  def set_business_headers
    if @business
      headers["X-SMTPAPI-CATEGORY"] = @business.code
    end
  end
end
```


- Mailer Filters abort further processing if body is set to a non-nil value.

5 Using Action Mailer Helpers

Action Mailer now just inherits from `AbstractController`, so you have access to the same generic helpers as you do in Action Controller.

6 Action Mailer Configuration

The following configuration options are best made in one of the environment files (environment.rb, production.rb, etc...)

address - Allows you to use a remote mail server. Just change it from its default "localhost" setting.

Configuration Description

logger Generates information on the mailing run if available. Can be set to nil for no logging. Compatible with both Ruby's own Logger and Log4r loggers.

smtp_settings Allows detailed configuration for :smtp delivery method:

:port - On the off chance that your mail server doesn't run on port 25, you can change it.

:domain - If you need to specify a HELO domain, you can do it here.

:user_name - If your mail server requires authentication, set the username in this setting.

:password - If your mail server requires authentication, set the password in this setting.

:authentication - If your mail server requires authentication, you need to specify the authentication type here. This is a symbol and one of :plain, :login, :cram_md5.

:enable_starttls_auto - Set this to false if there is a problem with your server certificate that you cannot resolve.

|| sendmail_settings | Allows you to override options for the :sendmail delivery method.

:location - The location of the sendmail executable. Defaults to /usr/sbin/sendmail.

:arguments - The command line arguments to be passed to sendmail. Defaults to -i -t.

|| raise_delivery_errors | Whether or not errors should be raised if the email fails to be delivered. This only works if the external email server is configured for immediate delivery. || delivery_method | Defines a delivery method. Possible values are:

:smtp (default), can be configured by using config.action_mailer.smtp_settings.

:sendmail, can be configured by using config.action_mailer.sendmail_settings.

:file: save emails to files; can be configured by using config.action_mailer.file_settings.

:test: save emails to ActionMailer::Base.deliveries array. See API docs for more info. || perform_deliveries | Determines whether deliveries are actually carried out when the deliver method is invoked on the Mail message. By default they are, but this can be turned off to help functional

testing. || deliveries | Keeps an array of all the emails sent out through the Action Mailer with delivery_method :test. Most useful for unit and functional testing. || default_options | Allows you to set default values for the mail method options (:from, :reply_to, etc.). |

For a complete writeup of possible configurations see the [Configuring Action Mailer](#) in our Configuring Rails Applications guide.

6.1 Example Action Mailer Configuration

An example would be adding the following to your appropriate config/environments/\$RAILS_ENV.rb file:

```
config.action_mailer.delivery_method = :sendmail
# Defaults to:
# config.action_mailer.sendmail_settings = {
#   location: '/usr/sbin/sendmail',
#   arguments: '-i -t'
# }
config.action_mailer.perform_deliveries = true
config.action_mailer.raise_delivery_errors = true
config.action_mailer.default_options = {from: 'no-reply@example.com'}
```

6.2 Action Mailer Configuration for Gmail

As Action Mailer now uses the [Mail gem](#), this becomes as simple as adding to your config/environments/\$RAILS_ENV.rb file:

```
config.action_mailer.delivery_method = :smtp
config.action_mailer.smtp_settings = {
  address:           'smtp.gmail.com',
  port:              587,
  domain:             'example.com',
  user_name:          '<username>',
  password:           '<password>',
  authentication:     'plain',
  enable_starttls_auto: true }
```

7 Mailer Testing

You can find detailed instructions on how to test your mailers in the [testing guide](#).

8 Intercepting Emails

There are situations where you need to edit an email before it's delivered. Fortunately Action Mailer provides hooks to intercept every email. You can register an interceptor to make modifications to mail messages right before they are handed to the delivery agents.

```
class SandboxEmailInterceptor
  def self.delivering_email(message)
    message.to = ['sandbox@example.com']
  end
end
```

Before the interceptor can do its job you need to register it with the Action Mailer framework. You can do this in an initializer file `config/initializers/sandbox_email_interceptor.rb`

```
ActionMailer::Base.register_interceptor(SandboxEmailInterceptor) if Rails.env.staging?
```

Note: The example above uses a custom environment called "staging" for a production like server but for testing purposes. You can read [Creating Rails environments](#) for more information about custom Rails environments.

Feedback

You're encouraged to help improve the quality of this guide.

Please contribute if you see any typos or factual errors. To get started, you can read our [documentation contributions](#) section.

You may also find incomplete content, or stuff that is not up to date. Please do add any missing documentation for master. Make sure to check [Edge Guides](#) first to verify if the issues are already fixed or not on the master branch. Check the [Ruby on Rails Guides Guidelines](#) for style and conventions.

If for whatever reason you spot something to fix but cannot patch it yourself, please [open an issue](#).

And last but not least, any kind of discussion regarding Ruby on Rails documentation is very welcome in the [rubyonrails-docs mailing list](#).