DO NOT READ THIS FILE ON GITHUB, GUIDES ARE PUBLISHED ON https://guides.rubyonrails.org.

Action Mailer Basics

This guide provides you with all you need to get started in sending emails from 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 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.

What is Action Mailer?

Action Mailer allows you to send emails from your application using mailer classes and views.

Mailers are similar to controllers

They inherit from `ActionMailer::Base` and live in `app/mailers`. Mailers also work very similarly to controllers. Some examples of similarities are enumerated below. Mailers have:

- Actions, and also, associated views that appear in `app/views`.
- Instance variables that are accessible in views.
- The ability to utilise layouts and partials.
- The ability to access a params hash.

Sending Emails

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

Walkthrough to Generating a Mailer

Create the Mailer

```
$ bin/rails generate mailer User
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: "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.

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
```

Edit the Mailer

Mailers have methods called "actions" and they use views to structure their 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 = params[: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.

- The `default` method sets default values for all emails sent from this mailer. In this case, we use it to set the `:from` header value for all messages in this class. This can be overridden on a per-email basis.
- The `mail` method creates the actual email message. We use it to specify the values of headers like `:to` and `:subject` per email.

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.

Calling the Mailer

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

Setting this up is simple.

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

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

Now that we have a user model to play with, we will edit the `app/controllers/users_controller.rb` file, 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.with(user: @user).welcome_email` right after the user is successfully saved.

We will enqueue the email to be sent by using `deliver_later`, which is backed by Active Job. That way, the controller action can continue without waiting for the send to complete.

```
class UsersController < ApplicationController
  # ...

  # POST /users or /users.json
  def create
    @user = User.new(user_params)

    respond_to do |format|
      if @user.save
        # Tell the UserMailer to send a welcome email after save
        UserMailer.with(user: @user).welcome_email.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 via the `:async` adapter. So, you can use `deliver_later` to send emails asynchronously. Active Job's default adapter runs jobs with an in-process thread pool. It's well-suited for the development/test environments, since it doesn't require any external infrastructure, but it's a poor fit for production since it drops pending jobs on restart. If you need a persistent backend, you will need to use an Active Job adapter that has a persistent 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.with(user: user).weekly_summary.deliver_now
    end
  end
end
```

Any key-value pair passed to `with` just becomes the `params` for the mailer action. So `with(user: @user, account: @user.account)` makes `params[:user]` and `params[:account]` available in the mailer action. Just like controllers have params.

The method `welcome_email` returns an `ActionMailer::MessageDelivery` object which can then be told to `deliver_now` or `deliver_later` to send itself out. The `ActionMailer::MessageDelivery` object is 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.

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.

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` - Creates the actual email itself. You can pass in headers as a hash to the `mail` method as a parameter. `mail` will create an email — either plain text or multipart — depending on what email templates you have defined.

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/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.

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'
%>
```

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 < ApplicationMailer
  default to: -> { 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.

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. You can use `email_address_with_name` for that:

```ruby
def welcome_email
  @user = params[:user]
  mail(
    to: email_address_with_name(@user.email, @user.name),
    subject: 'Welcome to My Awesome Site'
  )
end
```

The same technique works to specify a sender name:

```ruby
class UserMailer < ApplicationMailer
  default from: email_address_with_name('notification@example.com', 'Example Company
Notifications')
end
```

If the name is a blank string, it returns just the address.

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:

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

  def welcome_email
    @user = params[: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:

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

  def welcome_email
    @user = params[: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 plain: 'Render text' }
    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.

If you would like to render a template located outside of the default `app/views/mailer_name/` directory, you can apply the `prepend_view_path`, like so:

```
class UserMailer < ApplicationMailer
  prepend_view_path "custom/path/to/mailer/view"

  # This will try to load "custom/path/to/mailer/view/welcome_email" template
  def welcome_email
    # ...
  end
end
```

You can also consider using the `append_view_path` method.

Caching mailer view

You can perform fragment caching in mailer views like in application views using the `cache` method.

```
<% cache do %>
  <%= @company.name %>
<% end %>
```

And to use this feature, you need to configure your application with this:

```
config.action_mailer.perform_caching = true
```

Fragment caching is also supported in multipart emails. Read more about caching in the Rails caching guide.

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.

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
    mail(to: params[:user].email) do |format|
      format.html { render layout: 'my_layout' }
      format.text
    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.

Previewing Emails

Action Mailer previews provide a way to see how emails look by visiting a special URL that renders them. In the above example, the preview class for `UserMailer` should be named `UserMailerPreview` and located in `test/mailers/previews/user_mailer_preview.rb`. To see the preview of `welcome_email`, implement a method that has the same name and call `UserMailer.welcome_email`:

```
class UserMailerPreview < ActionMailer::Preview
  def welcome_email
    UserMailer.with(user: User.first).welcome_email
  end
end
```

Then the preview will be available in http://localhost:3000/rails/mailers/user_mailer/welcome_email.

If you change something in `app/views/user_mailer/welcome_email.html.erb` or the mailer itself, it'll automatically reload and render it so you can visually see the new style instantly. A list of previews are also available in http://localhost:3000/rails/mailers.

By default, these preview classes live in `test/mailers/previews`. This can be configured using the `preview_path` option. For example, if you want to change it to `lib/mailer_previews`, you can configure it in `config/application.rb`:

```
config.action_mailer.preview_path = "#{Rails.root}/lib/mailer_previews"
```

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.

Generating URLs with `url_for`

`url_for` generates a full URL by default in templates.

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') %>
```

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') %>
```

NOTE: non-`GET` links require rails-ujs or jQuery UJS, and won't work in mailer templates. They will result in normal `GET` requests.

Adding images 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 `:asset_host` parameter yourself.

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

```
config.asset_host = 'http://example.com'
```

Now you can display an image inside your email.

```
<%= image_tag 'image.jpg' %>
```

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.

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 = params[:user]
    @url  = user_url(@user)
    delivery_options = { user_name: params[:company].smtp_user,
                         password: params[:company].smtp_password,
                         address: params[:company].smtp_host }
    mail(to: @user.email,
         subject: "Please see the Terms and Conditions attached",
         delivery_method_options: delivery_options)
  end
end
```

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
    mail(to: params[:user].email,
         body: params[:email_body],
         content_type: "text/html",
         subject: "Already rendered!")
  end
end
```

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 set instance variables, populate the mail object with defaults, or insert default headers and attachments.

```
class InvitationsMailer < ApplicationMailer
  before_action :set_inviter_and_invitee
  before_action { @account = params[:inviter].account }

  default to:       -> { @invitee.email_address },
          from:     -> { common_address(@inviter) },
          reply_to: -> { @inviter.email_address_with_name }

  def account_invitation
    mail subject: "#{@inviter.name} invited you to their Basecamp (#
{@account.name})"
  end

  def project_invitation
    @project    = params[:project]
    @summarizer = ProjectInvitationSummarizer.new(@project.bucket)

    mail subject: "#{@inviter.name.familiar} added you to a project in Basecamp (#
{@account.name})"
  end

  private

  def set_inviter_and_invitee
    @inviter = params[:inviter]
    @invitee = params[:invitee]
  end
end
```

- You could use an `after_action` to do similar setup as a `before_action` but using instance variables set in your mailer action.
- Using an `after_action` callback also enables you to override delivery method settings by updating `mail.delivery_method.settings`.

```
class UserMailer < ApplicationMailer
  before_action { @business, @user = params[:business], params[:user] }

  after_action :set_delivery_options,
               :prevent_delivery_to_guests,
               :set_business_headers

  def feedback_message
  end

  def campaign_message
```

```
  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.

Using Action Mailer Helpers

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

There are also some Action Mailer-specific helper methods available in `ActionMailer::MailHelper`. For example, these allow accessing the mailer instance from your view with `mailer`, and accessing the message as `message`:

```
<%= stylesheet_link_tag mailer.name.underscore %>
<h1><%= message.subject %></h1>
```

Action Mailer Configuration

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

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:

	• `:address` - Allows you to use a remote mail server. Just change it from its default `"localhost"` setting. • `: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` (will send the password in the clear), `:login` (will send password Base64 encoded) or `:cram_md5` (combines a Challenge/Response mechanism to exchange information and a cryptographic Message Digest 5 algorithm to hash important information) • `:enable_starttls` - Use STARTTLS when connecting to your SMTP server and fail if unsupported. Defaults to `false`. • `:enable_starttls_auto` - Detects if STARTTLS is enabled in your SMTP server and starts to use it. Defaults to `true`. • `:openssl_verify_mode` - When using TLS, you can set how OpenSSL checks the certificate. This is really useful if you need to validate a self-signed and/or a wildcard certificate. You can use the name of an OpenSSL verify constant ('none' or 'peer') or directly the constant (`OpenSSL::SSL::VERIFY_NONE` or `OpenSSL::SSL::VERIFY_PEER`). • `:ssl/:tls` - Enables the SMTP connection to use SMTP/TLS (SMTPS: SMTP over direct TLS connection) • `:open_timeout` - Number of seconds to wait while attempting to open a connection. • `:read_timeout` - Number of seconds to wait until timing-out a read(2) call.
`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`.
`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. If this value is `false`, `deliveries` array will not be populated even if `delivery_method` is `:test`.
`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.
`delivery_job`	The job class used with `deliver_later`. Defaults to `ActionMailer::MailDeliveryJob`.
`deliver_later_queue_name`	The name of the queue used with `deliver_later`.
`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.

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'
# }
config.action_mailer.perform_deliveries = true
config.action_mailer.raise_delivery_errors = true
config.action_mailer.default_options = {from: 'no-reply@example.com'}
```

Action Mailer Configuration for Gmail

Action Mailer uses the Mail gem and accepts similar configuration. Add this to your `config/environments/$RAILS_ENV.rb` file to send via Gmail:

```
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,
```

```
  open_timeout:          5,
  read_timeout:          5 }
```

NOTE: On July 15, 2014, Google increased its security measures to block attempts from apps it deems less secure. You can change your Gmail settings here to allow the attempts. If your Gmail account has 2-factor authentication enabled, then you will need to set an app password and use that instead of your regular password.

Mailer Testing

You can find detailed instructions on how to test your mailers in the testing guide.

Intercepting and Observing Emails

Action Mailer provides hooks into the Mail observer and interceptor methods. These allow you to register classes that are called during the mail delivery life cycle of every email sent.

Intercepting Emails

Interceptors allow you to make modifications to emails before they are handed off to the delivery agents. An interceptor class must implement the `::delivering_email(message)` method which will be called before the email is sent.

```
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 using the `interceptors` config option. You can do this in an initializer file like `config/initializers/mail_interceptors.rb`:

```
Rails.application.configure do
  if Rails.env.staging?
    config.action_mailer.interceptors = %w[SandboxEmailInterceptor]
  end
end
```

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.

Observing Emails

Observers give you access to the email message after it has been sent. An observer class must implement the `:delivered_email(message)` method, which will be called after the email is sent.

```
class EmailDeliveryObserver
  def self.delivered_email(message)
    EmailDelivery.log(message)
  end
end
```

Similar to interceptors, you must register observers using the `observers` config option. You can do this in an initializer file like `config/initializers/mail_observers.rb`:

```
Rails.application.configure do
  config.action_mailer.observers = %w[EmailDeliveryObserver]
end
```