Dizzy Making business beautiful.

Cheatsheet * ActionMailer

ActionMailer::Base Configuration

ActionMailer is configured by acessing configuration methods at the class level, for example, `ActionMailer::Base.template_root = "/my/templates"`. These methods allow you to define the overall settings to be used by your application whenever it invokes ActionMailer. Define these settings in your `config/environment.rb` file using `config.action_mailer.method_name_here`. If you require different settings for each of your Rails' environments, define settings separately via `config/environments`.

smtp_settings = {hash}

- `:address` - the address of the SMTP server you will be using to send email. Defaults to `localhost`
- `:port` - the port number of the SMTP server you will be using. Defaults to `25`
- `: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 variable
- `:password` - if your mail server requires authentication, set the password in this variable
- `: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`

sendmail_settings = {hash}

- `:location` - the location of the sendmail executable, defaults to `/usr/sbin/sendmail`
- `:arguments` - the command line arguments for sendmail

raise_delivery_errors = true or false

Whether or not errors should be raised if the email fails to be delivered.

delivery_method = :smtp, :sendmail or :test

Defines a delivery method, defaults to `:smtp`

perform_deliveries = true or false

Determines whether `deliver_*` methods are actually carried out. By default they are, but this can be turned off to help functional testing.

template_root = "/path"

The root from which template references will be made

logger

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

default_charset = "string"

the default charset used for the body and to encode the subject. Defaults to `UTF-8`. You can also pick a different charset from inside a mailer method by setting `charset`

default_mime_version = "string"

The default mime version used for the message. Defaults to `1.0`. You can also pick a different value from inside a mailer method by setting `mime_version`

default_implicit_parts_order = [array]

When an email is built implicitly, this variable controls how the parts are ordered. Defaults to `["text/html", "text/enriched", "text/plain"]`. Items that appear first in the array have higher priority in the receiving mail client and appear last in the mime encoded message. You can also pick a different value from inside a mailer method by setting `implicit_parts_order`

default_content_type = "string"

The default content type used for the main part of the message. Defaults to `text/plain`. You can also pick a different value from inside a mailer method by setting `content_type`

Delivering mail

Once a mailer action and template are defined, you can `deliver` your message or `create` and save it for delivery later by calling the mailer class and prefixing your chosen class method with `deliver_` or `create_`

Send mail

```
Notifier.deliver_signup_notification(customer)
```

Create mail

```
mail = Notifier.create_signup_notification(customer)
Notifier.deliver(mail)
```

You can pass the mailer model any variables you need to use in the generation of the email. In the example above we have passed it a variable named `customer` which could be an instance of an ActiveRecord `Customer` model. We can then access our customer's details in the mailer model.

Views & Templates

Like ActionController, each mailer class has a corresponding view directory in which each method of the class looks for a template with its own name. For example...

Mailer model	*Class method*	*Corresponding template*
`Notifier`	`signup_notification`	`app/views/notifier/signup_notification.erb`
`Notifier`	`despatch_alert`	`app/views/notifier/despatch_alert.erb`
`MailingList`	`welcome_message`	`app/views/mailing_list/welcome_message.erb`

URLs

If your view includes URLs from the application, you need to use `url_for` in the mailer class method instead of in the view template. You can pass the result to the view via the `body` method. Unlike controllers from ActionPack, the mailer instance doesn't have any context about the incoming request.

```
  body :home_page => url_for(:host => "dizzy.co.uk", :controller =>
"welcome", :action => "index")
```

Mailer Model

To use ActionMailer, you need to create a mailer model. Emails are defined by creating methods within the mailer model which are then used to set variables to be used in the mail template, to change options on the mail, or to add attachments.

Mailer model generator

```
    ruby script/generate mailer NameOfMailer method1
method2 method3
```

Example mailer model

```
class OrderMailer < ActionMailer::Base

  def confirm(order,sent_at = Time.now)
    subject    "Subject line goes here"
    body       :order => order
    recipients ["bill@microsoft.com", "steve@apple.
com"]
    from       "david@dizzy.co.uk"
    sent_on    sent_at
  end
end
```

Mailer Configuration Methods

recipients = [array] or "string"	A string containing the email of address of the recipient, or an array of strings containing email addresses of multiple recipients. Will use the email's `To:` header.
sent_on = Time object	A `Time` object which will be used to set the `Date:` header of the email. If not specified, then the current time and date will be used.
subject = "string"	The subject line to be used to set the email's `Subject:` header.
from = [array] or "string"	A string containing the email address to appear on the `From:` line of the email being created, or a array of strings containing multiple email addresses in the same format as `recipients`.
body = {hash}	The `body` method sets instance variables to be available in the view template. For example, to make the variables `order` and `name` accessible as `@order` and `@name` respectively in your view template, use... `body :order => order, :name => name`

attachment = {hash} or block	Enables you to add attachments to your email message.
	`attachment :content_type => "image/jpeg", :body => File.read("an-image.jpg")`
	`attachment "application/pdf" do |a| a.body = generate_your_pdf_here() end`

bcc = [array] or "string"	Blind carbon copy recipients in the same format as `recipients`
cc = [array] or "string"	Carbon copy recipients in the same format as `recipients`
content_type = "string"	Set the content type of the message. Defaults to `text/plain`
headers = {hash}	A hash containing name/value pairs to be converted into abitrary header lines. For example... `headers "X-Mail-Count" => 107370`
mime_version = "string"	The mime version for the message. Defaults to `1.0`
charset = "string"	The charset for the body and to encode the subject. Defaults to `UTF-8`
implicit_parts_order = [array]	When an email is built implicitly, this variable controls how the parts are ordered. Defaults to `["text/html", "text/enriched", "text/plain"]`. Items that appear first in the array have higher priority in the receiving mail client and appear last in the mime encoded message.

Multipart messages

There are two ways to send multipart email messages, *explicity* by manually defining each part, and *implicitly* by letting ActionMailer do the donkey work.

Explicitly

You can explicitly define multipart messages using the `part` method...

```
  part "text/plain" do |p|
    p.body = render_message("signup-as-plain", :account
=> recipient)
    p.transfer_encoding = "base64"
  end
  part :content_type => "text/html",
    :body => render_message("signup-as-html", :account
=> recipient)
```

Implicitly

ActionMailer will automatically detect and use multipart templates, where each template is named after the name of the method, followed by the content type. Each such detected template will be added as a separate part to the message.For example:

- `signup_notification.text.plain.erb`
- `signup_notification.text.html.erb`
- `signup_notification.text.xml.builder`

Each would be rendered and added as a separate part to the message with the corresponding content type. The same `body` hash is passed to each template.