PeepCode press

$9

MMS2R

Making email useful

by Mike Mondragon and Luke Francl

CONTENTS

MMS2R: Making Email Useful

©2008 Mike Mondragon and Luke Francl

Introduction

CHAPTER 1

Almost everyone has a mobile phone these days. Sending and receiving text messages with these phones has become a convenient means of communication for millions of people around the world. Have you ever asked your significant other to pick up an extra gallon of milk after they have already left for the grocery store? If you are a good texter, sending a message will be much quicker than making a call. Text messaging combines the advantage of asynchronous messaging (unlike a phone call, a text message can be dealt with at the recipient's convenience) with near-universal availability.

This book will explain how to send and receive SMS, MMS, and email messages with your Ruby application. Typically this will be using Ruby on Rails (http://rubyonrails.org) but the code and libraries presented here can be adapted for use in other web frameworks such as Merb (http://merbivore.com), Camping (http://camping.rubyforge.org) or a generic application written without the help of a framework.

Before jumping in, it will be helpful to review at a high level the protocols involved and how the cellular industry allows third-party applications to interact with their networks.

Regional Lingo

People in the United States and Canada tend to use *cell phone* to refer to the mobile phone. In other English-speaking countries, *mobile phone* is more common.

SMS & MMS protocols

SMS is short for Short Message Service (http://en.wikipedia.org/wiki/Short_message_service). SMS actually refers to the protocol that is used to transmit the text that comprises the messages. It is convenient to refer the text of the messages as SMS and this book will intermingle technical and non-technical references to SMS. SMS messages are limited to 160 7-bit characters, though some carriers support messages encoded with 16-bit Unicode characters with a maximum of 70 characters.

MMS is short for Multimedia Message Service (http://en.wikipedia.org/wiki/Multimedia_Messaging_Service). We will intermingle the use of MMS (the protocol) and the content that it comprises. As the name suggests, an MMS can contain text, images, audio, video and other types of content. MMS is more ubiquitous in that the container that is used to transport the media is the common multipart MIME email format. MMS is transmitted over the HTTP protocol.

For simplicity, we will often use the term MMS to refer to both kinds of messages.

SMS: THE WORLD'S MOST EXPENSIVE DATA DISTRIBUTION CHANNEL

SMS has a lot of advantages, but the cost on a per-byte basis is astounding.

On the T-Mobile network in the USA, sending an SMS message costs a subscriber $0.15. The maximum size of a message is 140 bytes, for a cost of 0.1071 cents per byte which is $1.097/kilobyte or over $1,120/megabyte!

Even worse, the receiver also has to pay. On T-Mobile, receiving also costs $0.15, which effectively doubles the price.

This is much higher than the rates charged for other data transferred over the same network. *Consumerist* calculated that SMS messages were marked up over 7000% (http://consumerist.com/consumer/cellphones/why-are-text-messages-marked-up-4876-247518.php) when compared to data transfer rates on the Verizon network. Another study concluded that SMS rates were four times as expensive as downloading data from the Hubble Space Telescope (http://www.physorg.com/news129793047.html).

SMS is a very powerful and convenient medium controlled by the mobile carriers. If you want to play on their network, you'll have to pay their prices. Fortunately, bulk senders can get discounts. And for very low volume use, you can use email-to-SMS gateways.

Cellular/Mobile Networks

Cellular phones operate within cellular networks. These networks are private and consumers gain access to them by purchasing a cellular plan that grants access. Some call these private networks *walled gardens*, meaning that access is restricted to a cultivated interior.

A benefit of having a private network is that the cellular carriers have absolute control over what kinds of content traverse it. The consumer benefits from this privacy by being (generally) protected from SPAM or unwanted content. Usually, unsolicited MMS or voice calls never reach their handset.

To ensure the quality of their networks, cellular carriers place higher barriers on third party applications, such as your Rails application, in gaining access to the network. Applications are subjected to strict terms of service as well as significant fees to make use of the network.

Gateways

In order for an application to gain direct access to a cellular network typical monthly fees on the order of $10,000 US per month are charged. An extensive audit of the application is also performed by the carriers before the application is deployed.

A solution that has evolved for smaller applications such as what we are developing is to use a gateway service. The gateway charges your application a smaller fee that is in proportion to its use of the network. Gateways charge fees in the range of 2¢ to 8¢ per message depending on how many messages are purchased in bulk. The gateway will have its own terms of service for your application and will ensure that your application is abiding by the carrier's terms of service.

Receiving

MMS can be retrieved from the gateway service directly if that is a part of the service plan you choose. Alternately, MMS can be sent to a regular email address. Your application then retrieves the MMS as email and ingests its content based on the logic you create in your application.

An extra benefit of this setup is that you can also receive regular email even if it didn't originate on a mobile phone.

Sending

All of the MMS gateway providers provide an interface to their service. A common interface is to take a message bundled as a XML document from a HTTP POST. The gateway you choose will likely be based on its cost and its ease of use from a programming standpoint.

Specific gateways for sending and receiving MMS will be discussed later in the book.

SMS to Email Gateways

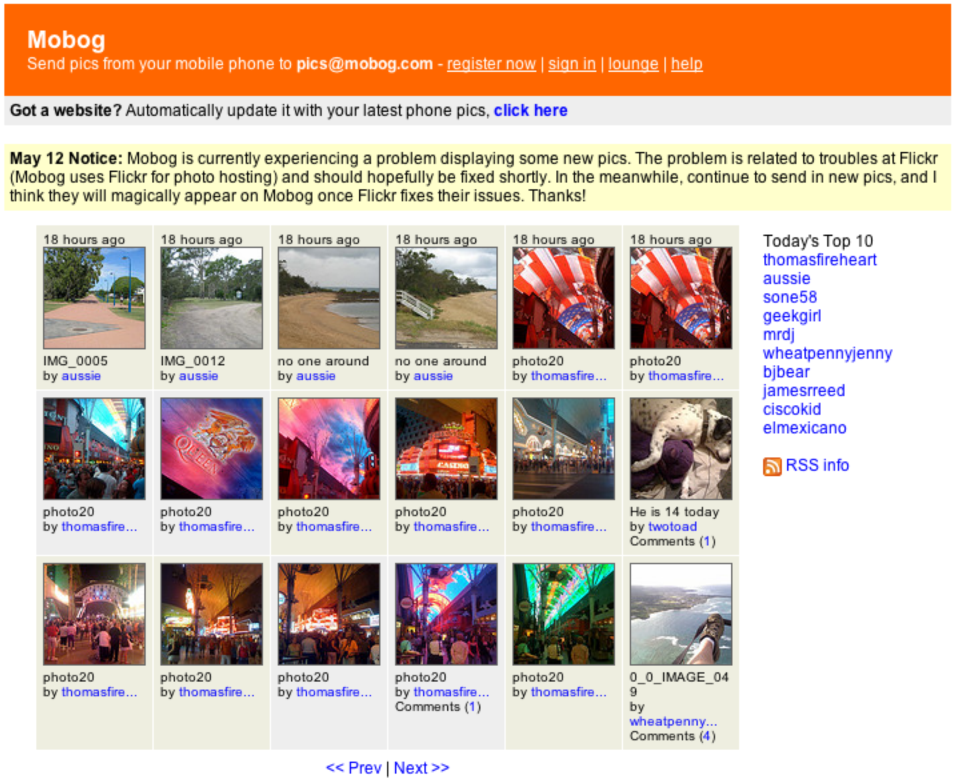
Some providers offer a way to connect mobile phones on their network to email via a special email address. Usually this takes the form of `phonenumber@carrier.com`. These gateways are not very reliable, but may suffice for low-volume applications.

Many carriers also allow their subscribers to send SMS messages to an email address, which can allow your application to accept mobile input without an expensive gateway. But again, this is not universal and is only suited for low volumes.

A Brief History

CHAPTER 2

Likely the first and largest micro blog based on MMS content is called mobog.com (http://mobog.com) and was created by Philip Kaplan http://en.wikipedia.org/wiki/Philip_Kaplan in 2003. Although Why the Lucky Stiff coined the term *tumblelog* in reference to the Anarchaia (http://anarchaia.org) blog in 2005, the tumblelog's origins go back at least to Kaplan's mobog.



Mobog
Send pics from your mobile phone to **pics@mobog.com** - register now | sign in | lounge | help

Got a website? Automatically update it with your latest phone pics, **click here**

May 12 Notice: Mobog is currently experiencing a problem displaying some new pics. The problem is related to troubles at Flickr (Mobog uses Flickr for photo hosting) and should hopefully be fixed shortly. In the meanwhile, continue to send in new pics, and I think they will magically appear on Mobog once Flickr fixes their issues. Thanks!

18 hours ago IMG_0005 by aussie	18 hours ago IMG_0012 by aussie	18 hours ago no one around by aussie	18 hours ago no one around by aussie	18 hours ago photo20 by thomasfire...	18 hours ago photo20 by thomasfire...
photo20 by thomasfire...	photo20 by thomasfire...	photo20 by thomasfire...	photo20 by thomasfire...	photo20 by thomasfire...	He is 14 today by twotoad Comments (1)
photo20 by thomasfire...	photo20 by thomasfire...	photo20 by thomasfire...	photo20 by thomasfire... Comments (1)	photo20 by thomasfire...	0_0_IMAGE_049 by wheatpenny... Comments (4)

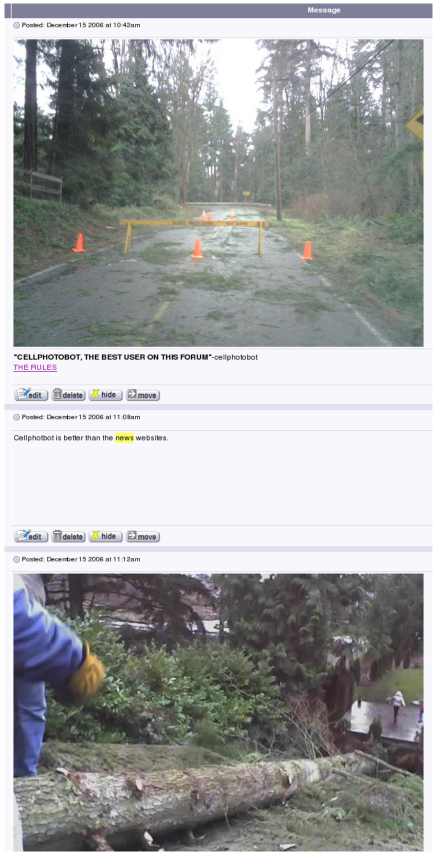
Today's Top 10
thomasfireheart
aussie
sone58
geekgirl
mrdj
wheatpennyjenny
bjbear
jamesrreed
ciscokid
elmexicano

RSS info

<< Prev | Next >>

FIG. A Mobog

Mike



Posted: December 15 2006 at 10:42am

"CELLPHOTOBOT, THE BEST USER ON THIS FORUM"-cellphotobot
THE RULES

Posted: December 15 2006 at 11:08am

Cellphotbot is better than the news websites.

Posted: December 15 2006 at 11:12am

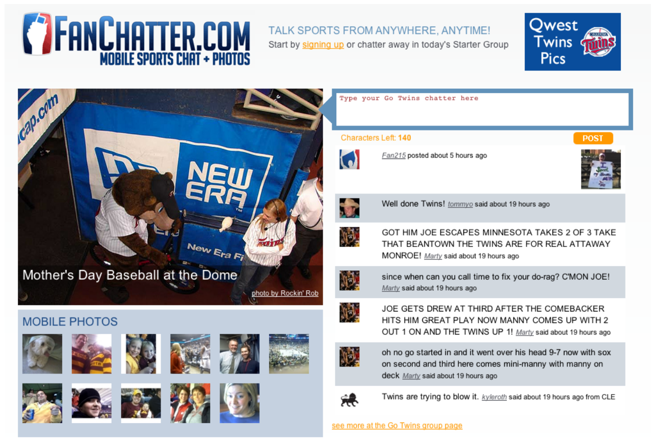
I have been interested in small devices as mobile data input since the late 90's when the Palm with JVM made its debut at JavaOne and Bill Joy was pushing Jini Technology (roaming devices) for Sun. Around 2000 I wrote a small Perl script to pull the subject from my email and send it via SMS to my cell phone. I admired Philip's micro blog when it launched and the notion of it has stuck with me to this day.

In 2005 I wrote a Perl script to process MMS, pull out an image, and post that to a private skate-board forum in Seattle. The logic that I discovered for the Perl script was transferred to Ruby in 2006 as the MMS2R Gem. I receive real feedback on MMS2R's perfor-mance from a work queue imple-mented as a Rails Acts As State Machine with `attachment_fu` that now handles the picture posting. My friends and I call this work queue *cellphotobot* and I use it to give me production feedback about MMS2R's performance for MMS that are being produced in the wild.

In December 2006 a severe ice storm hit Seattle crippling traffic for 3 or 4 days. On the first morning, pictures of crashed cars and fallen trees started to show up on our forum. One of our members posted that: "Cellphotobot is better than the news websites."

Luke

I first experimented with mobile development for a citizen journalism project for my employer in early 2007. We were tasked with accepting mobile submissions of photos and videos. I knew that cell phones could send photos via email due to my experience with Flickr, and so I discovered MMS2R and started adding more carrier support.


FANCHATTER.COM
MOBILE SPORTS CHAT + PHOTOS
TALK SPORTS FROM ANYWHERE, ANYTIME!
Start by signing up or chatter away in today's Starter Group
Qwest Twins Pics
Mother's Day Baseball at the Dome
photo by Rockin' Rob
MOBILE PHOTOS
Type your Go Twins chatter here
Characters Left: 140
POST

Fan215 posted about 5 hours ago

Well done Twins! tommyo said about 19 hours ago

GOT HIM JOE ESCAPES MINNESOTA TAKES 2 OF 3 TAKE THAT BEANTOWN THE TWINS ARE FOR REAL ATTAWAY MONROE! Marty said about 19 hours ago

since when can you call time to fix your do-rag? C'MON JOE! Marty said about 19 hours ago

JOE GETS DREW AT THIRD AFTER THE COMEBACKER HITS HIM GREAT PLAY NOW MANNY COMES UP WITH 2 OUT 1 ON AND THE TWINS UP 1! Marty said about 19 hours ago

oh no go started in and it went over his head 9-7 now with sox on second and third here comes mini-manny with manny on deck Marty said about 19 hours ago

Twins are trying to blow it. kyleroth said about 19 hours ago from CLE

see more at the Go Twins group page

FIG. B **FanChatter**

We took that experience forward when developing FanChatter (http://fanchatter.com), which allows sports fans to post messages and photos from their phones, email, or the web. This led us to develop a suite of stadium tools (http://fanchatter.com/stadium) for sports teams, which allows fans at the stadium to send in photos and have them appear on the jumbotron during the game. One of the most interesting things we've noticed is the clever photos that people send in.

FIG. C **I pinch you!**

You can see the stadium tools in action with the Minnesota Twins (http://twins.fanchatter.com).

Geoffrey

Real People - Real Stories
Come for the stories. Stay and share your own.

Get a free place to blog, post photos, share stories and connect with others. It's the easiest to use site on the planet and it's fun.

Before MMS2R was widely available, I implemented an email system for PNN (http://pnn.com), a family journalism site. Any user could choose a custom email address and send photos or blog posts to a specific page in their newspaper.

This made it easy for people to send content from their phones without the need for a full SMS shortcode, which are often too expensive and awkward for small startups to pay for.

We used a Ruby-based script that ran frequently and posted new emails to a controller on our Rails application. The Rails application processed the email and matched it up with a user's newspaper.

The hardest problems were parsing the email, and dealing with massive amounts of SPAM. If you can, you should allocate a utility server to do the email receiving and parsing so your web servers can be dedicated to serving web requests.

What Can You Do With Email?

Email is one of the most easily overlooked ways of getting information into or out of a web application. It is one of the first things people learn to do when they start out on the Internet.

Email also comes with expectations. People who receive an email expect to be able to hit *Reply* and rarely pay attention to the fact that they have just sent an email to a `do-not-reply-to` address that will disappear into the ether.

If you can give people the ability to email content directly to their blog, photo gallery, or personal journal, they will be much happier than if they were restricted to a web interface only.

Here are a few ideas and examples in the wild:

- BillMonk (https://www.billmonk.com) is an IOU service for friends. You can send an SMS describing a bill and BillMonk will parse it and add it to your tab.
- FamSpam (http://famspam.com) is a family mailing list. Although it

  has a beautiful web interface, it is also possible to interact with it completely through email. Family members can send photos from their phone and it will be carbon copied to all the other family members. Once received, they can simply reply to continue the conversation, without needing to visit the website.

- Sandy (http://www.iwantsandy.com) is a personal assistant, powered by email. A powerful text parsing engine extracts times, phone numbers, and tags from plain text sent via email or SMS.

Email is also useful as part of a larger application, even if email isn't the main focus. You may want to integrate MMS with an existing application such as:

- A customer support system could look for keywords and send an immediate response with answers to frequently asked questions.
- A bug-tracking system could accept bug reports via email and log them to a database for further investigation.
- A document management system could receive documents as email attachments and file them for backup or publishing.

Processing MMS

CHAPTER 3

In order to understand the benefits of MMS2R, we first need to look at the TMail library for working with email from Ruby.

Then, we'll look at what the MMS2R library can do for you.

TMail

TMail is the semi-official Ruby library for processing email. It is used by Rails `ActionMailer` and many other projects. Unfortunately, for a long time TMail was unmaintained and therefore it was necessary to maintain patches for TMail in the Rails core. However, development of TMail has been picked up and is now being actively maintained on its own as a gem (http://tmail.rubyforge.org).

Creating a `TMail::Mail` instance is as simple as calling the `TMail::Mail.load` method with the path to a file containing mail protocol formatted text.

```
# Load from a file
mail = TMail::Mail.load('/path/to/mail/file')
```

You can also create new instance using a string with `TMail::Mail.parse`:

```
# Parse from a string
mail = TMail::Mail.parse("To: you@example.com")
```

In most cases, you'll be using `parse` to work with mail content received from a POP or IMAP inbox. However, the file-based method is useful when writing tests or debugging issues from a specific message or carrier.

Once you've created the instance, the TMail API is pretty easy to use:

simple_tmail.rb
```
mail.from     # ["you@example.com"]
mail.to       # ["me@example.com"]
mail.subject  # "i can has email?"
mail.body     # "Body text ftw!"
```

Both `from` and `to` return arrays. Most of the time, you'll want the first one.

For more examples, see the TMail API documentation (http://tmail.rubyforge.org/reference/index.html) and quickstart (http://tmail.rubyforge.org/quickstart/index.html).

As noted above, Rails uses TMail. For example, when a `receive` method is defined in an `ActionMailer::Base` subclass, the argument in the method's signature is a `TMail::Mail` object.

simple_mailer.rb
```
class SimpleMailer < ActionMailer::Base
  def receive(mail)
    # do something with the mail subject
    logger.info("Got a mail about: #{mail.subject}")
  end
end
```

We'll use this method later to receive email and process it with MMS2R.

The Problem

MMS2R was created because it is painful to process MMS messages from the various mobile carriers. Application developers just want to access the user-generated content such as an image from the MMS payload or plain text from the email body.

However, each carrier treats messages differently. For instance, MMS from AT&T are clean of the carrier's advertising. But MMS from T-Mobile include an HTML container with the carrier's branding. The branding is text with image elements that are also transported in the MMS payload.

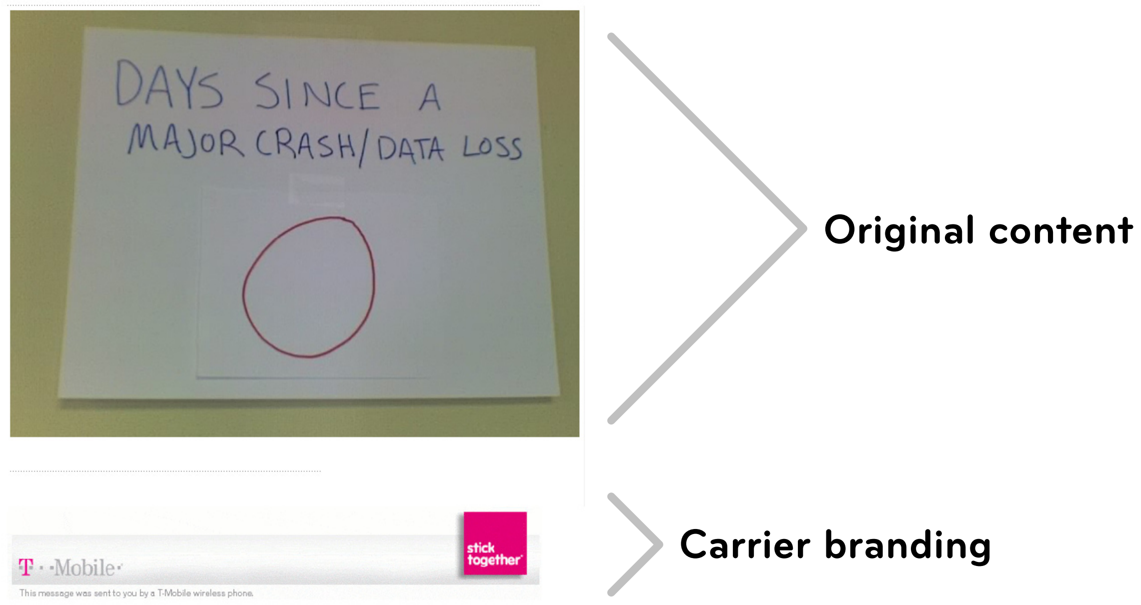
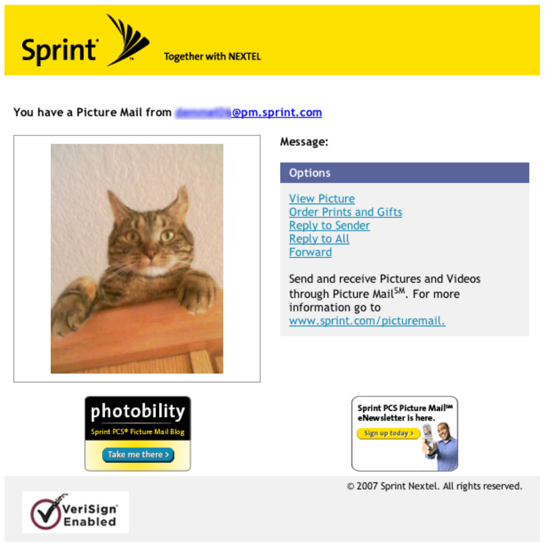

DAYS SINCE A MAJOR CRASH/DATA LOSS
Original content
T · Mobile·
stick together
This message was sent to you by a T-Mobile wireless phone.
Carrier branding

FIG. D T-Mobile branding as extra image attachments

Sprint is even worse! They add branding like T-Mobile does but the actual user-generated content resides on an external server. In both cases, a person viewing the MMS in a email reader is subjected to advertising from the sender's carrier. An application programmer needs to implement logic to find the user-generated content.



You have a Picture Mail from [redacted]@pm.sprint.com

Message:

Send and receive Pictures and Videos through Picture Mail^{SM}. For more information go to www.sprint.com/picturemail.

photobility
Sprint PCS® Picture Mail Blog
Take me there ›
Sprint PCS Picture Mail^{SM} eNewsletter is here.
Sign up today ›
VeriSign Enabled

© 2007 Sprint Nextel. All rights reserved.

FIG. E **Sprint email with MMS content**

See Luke's article on this subject at Rail Spikes (http://railspikes.com/2007/5/24/mms2r-with-rails).

MMS2R is able to process MMS from most of the mobile carriers in North America, many in Europe, and others around the world. What *process* means in terms of MMS2R is that the library has the ability to trim the non-user generated content from the MMS payload. The result for the application developer is the ability to access the remaining content that was most likely generated by the user. That might be an image, but it may also be text that is free of any extra cruft appended by the carrier, such as a footer with an advertising message. The MMS2R project welcomes samples from carriers that

SUPPORTED CARRIERS

At the time of writing, these carriers are supported in the MMS2R gem. Other carriers can be added easily.

- 1nbox.net
- bellsouth.net
- mms.3ireland.ie
- mms.alltel.com
- mms.att.net
- mms.dobson.net
- mms.luxgsm.lu
- mms.mobileiam.ma
- mms.mtn.co.za
- mms.mycricket.com
- mms.myhelio.com
- mms.netcom.no
- mms.o2online.de
- mms.three.co.uk
- mobile.indosat.net.id
- orangemms.net
- pm.sprint.com
- pxt.vodafone.net.nz
- rci.rogers.com
- sms.sasktel.com
- tmomail.net
- vzwpix.com
- waw.plspictures.com
- messaging.nextel.com
- mms.vodacom4me.co.za

it doesn't currently process so that support of them can be added to the library.

If MMS2R does not have a processing template for the MMS payload, it does not attempt to filter any of the message's content. However, the library will attempt to return the single most likely user generated content (such as an image from a camera) since that data is typically the largest part of the MMS.

MMS2R: NOT JUST FOR MMS!

MMS2R is great for stripping advertising from MMS messages, but underlying that is a great API façade for TMail's multipart email parsing.

When a new MMS2R::Media#new is created, the library takes care of pulling all the multipart elements of a message into the `media` hash. They're organized and query-able by MIME type.

So next time you need to parse multipart email, remember that MMS2R is not just for MMS messages.

TMail Issues

Even if the email is properly formatted, it's sometimes awkward to work with the TMail API. Plain text and HTML are returned together when fetching the `body`. Attachments aren't organized by content type.

The TMail website (http://tmail.rubyforge.org/quickstart/index.html) says:

> *Right now, TMail does not touch the body of the mail message, for example, it doesn't encode, decode, handle character sets etc. It only handles the message and headers and gives you access to read and write the mail body.*

MMS2R provides a more friendly API for working with email of any kind.

MMS2R Basics

Here is a simple example.

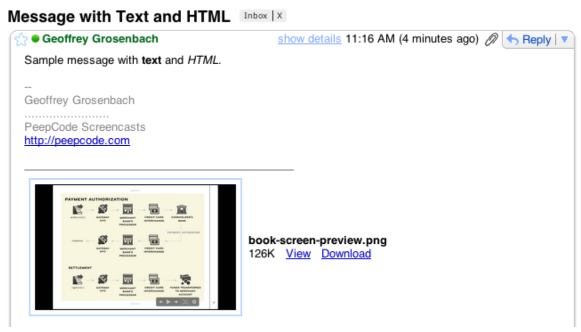
We open an MMS that contains an image from a mobile phone's container. The subject is also accessed (although many MMS messages are delivered without a subject).

simple_mms2r.rb
```
require 'rubygems'
require 'tmail'
require 'mms2r'

file = '/path/to/sample.mail'
mail = TMail::Mail.parse(IO.read(file))

mms  = MMS2R::Media.new(mail)

subject         = mms.subject
plain_text_body = mms.body

# Get a file handle that can be
# passed to attachment_fu
default_file = mms.default_media
# <File:/tmp/topfunky/mms2r/book-screen-preview.png>

# This returns the temporary path to the image
png_path = mms.media['image/png'].first
# /tmp/topfunky/mms2r/book-screen-preview.png

# Other paths to media are available by type
media_types_hash = mms.media
# {'text/plain' => ..., 'image/png' => ...}
```

API Walkthrough

Let's look at this again, step by step.

Here's a sample email in my inbox. It contains HTML, plain text, and an image attachment.


Message with Text and HTML Inbox | X

Geoffrey Grosenbach show details 11:16 AM (4 minutes ago) Reply

Sample message with **text** and *HTML*.

--
Geoffrey Grosenbach
....
PeepCode Screencasts
http://peepcode.com

book-screen-preview.png
126K View Download

FIG. F **A rich text email with an attachment**

Most email clients provide a menu item to view the raw message. Gmail calls this "Show original."

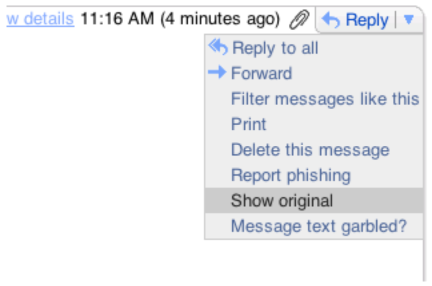
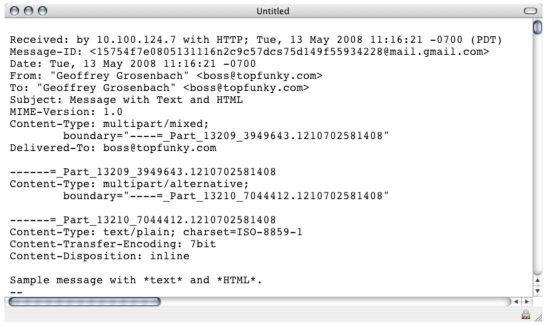


FIG. G **Gmail option to view the raw message**

For this example, I've copied the raw text and pasted it to a file on my hard drive. A live application would use a Ruby-based POP or IMAP client to fetch the message, as we will mention later.



Untitled

```
Received: by 10.100.124.7 with HTTP; Tue, 13 May 2008 11:16:21 -0700 (PDT)
Message-ID: <15754f7e0805131116n2c9c57dcs75d149f55934228@mail.gmail.com>
Date: Tue, 13 May 2008 11:16:21 -0700
From: "Geoffrey Grosenbach" <boss@topfunky.com>
To: "Geoffrey Grosenbach" <boss@topfunky.com>
Subject: Message with Text and HTML
MIME-Version: 1.0
Content-Type: multipart/mixed;
        boundary="----=_Part_13209_3949643.1210702581408"
Delivered-To: boss@topfunky.com

------=_Part_13209_3949643.1210702581408
Content-Type: multipart/alternative;
        boundary="----=_Part_13210_7044412.1210702581408"

------=_Part_13210_7044412.1210702581408
Content-Type: text/plain; charset=ISO-8859-1
Content-Transfer-Encoding: 7bit
Content-Disposition: inline

Sample message with *text* and *HTML*.
--
```

FIG. H **Raw email message**

I've started an IRB session to show how easy this is, and how MMS2R differs from TMail.

First, we'll load the necessary libraries and read the raw email from disk. At this point, it's just plain text.

```
irb_session.rb
>> require 'tmail'
>> require 'mms2r'

>> raw = File.read('code/sample.mail')
=> "Received: by 10.100.124.7 with HTTP..."
```

Next, we create a TMail object with the contents of the email.

```
>> mail = TMail::Mail.parse(raw)
=> #<TMail::Mail port=#<TMail::StringPort:id=0x9a6db6>
bodyport=nil>
```

TMail has parsed the email, but the interface is still a bit awkward. For example, the email has both plain text and HTML segments, but TMail returns them both when we ask for the body. This isn't very useful!

In addition, the attachments are presented as an array. Later, MMS2R will give us an easier way to access these.

```
irb_session.rb
>> mail.body
=> "Sample message with text and HTML.\n-- \nGeoffrey Gro
senbach\n.......................\nPeepCode Screencasts\
nhttp://peepcode.com\n\nSample message with <b>text</b> and
<i>HTML</i>.<div><br></div><div>-- <br>Geoffrey Grosenbach
<br>.......................<br>PeepCode Screencasts<br><a
href="http://peepcode.com">http://peepcode.com</a><br></
div>\n\nAttachment: book-screen-preview.png\n"

>> mail.attachments
=> [#<TMail::Attachment:0x2d9950c>]
```

Now that you've seen how TMail processes the email, let's see if MMS2R can do any better. We load the TMail object into a new MMS2R::Media object.

```
>> mms = MMS2R::Media.new(mail)
=> #<MMS2R::Media:0x2d95d6c @was_processed=true...
```

If lazy processing is needed, pass in `:process => :lazy` to `new` when a new MMS2R object is created. However, you'll need to explicitly call `process` to trigger processing at a later point within your application.

This time, it returns the plain text version when we ask for the `body`. We can also access the HTML version of this rich text email, but most web applications will be concerned only with processing the plain text version.

```
>> mms.body
=> "Sample message with text and HTML.\n-- \nGeoffrey Gro
senbach\n.......................\nPeepCode Screencasts\
nhttp://peepcode.com"
```

The **media** method returns a hash of all the parts of the email, separated by content type. This is much more useful than the blind array that TMail returned. We can look at the **keys** of the hash to see what's available in this message.

```
>> mms.media.keys
=> ["text/plain", "text/html", "image/png"]
```

The convenient **default_media** method returns a file handle to the photo attachment. If there are several photo attachments, this will return only one.

```
>> mms.default_media
=> #<File:/tmp/topfunky/mms2r/15754f7e0805131116n2c9c57dcs75
d149f55934228mailgmailcom/3/book-screen-preview.png>
```

Alternately, we can access the array of PNG attachments. Instead of a file handle, we get the file path to the image. For long-term storage, you should copy this to a local directory, store the file contents in your database, or upload the image to a file storage service such as Amazon S3.

```
>> mms.media['image/png'].first
=> "/tmp/topfunky/mms2r/15754f7e0805131116n2c9c57dcs75d149f5
5934228mailgmailcom/3/book-screen-preview.png"
```

Finally, clean up the temporary files created for the text and attachments.

```
>> mms.purge
```

And that's it! With these few methods, you can extract the most important text and images from a simple email message.

If the message had contained extra carrier-generated cruft, it would have been cleaned out by the library as well.

Adding support for additional carriers

While MMS2R supports a number of carriers in North America and Europe (and has simple heuristics to find "real" content in unsupported carriers), you may need to add support for other carriers. This is easy to do and we encourage you to submit patches so we may officially support new carriers. The existing configurations are easily modified if alternative requirements are needed for your application.

Configuration files

In most cases, new carriers can be added by creating new processing rules in the configuration files located in the `conf/` directory. Processing rules are contained in files named after the carrier's sending domain with an appended `.yml` extension. If MMS emanating from your carrier has the address pattern of `joe.user@mms.yourcarrier.com`, then its configuration file can be found in `conf/mms.yourcarrier.com.yml`.

> The location of the `conf` directory can be found by calling MMS2R::Media.conf_dir. There is also a `conf_dir` setter if you need to point MMS2R at a custom directory.

IGNORE AND TRANSFORM

The format of the configuration is a hash in the YAML file. There are two root keys in this hash: `ignore` and `transform`. Each root key refers to another hash that is keyed by MIME type. Each MIME type refers to an array that contains strings or regular expressions that are used to determine if a part of the message has been injected by the carrier. In the case of `transform`, the array itself contains arrays that are used as arguments for `String#gsub`.

GENERIC DEFAULTS FOR ALL CARRIERS

The `conf/mms2r_media.yml` file contains master defaults for ignore and transform that will be applied to any MMS that MMS2R processes.

This default configuration will ignore subjects with the text "no subject." It will ignore parts of the message that declare how many attachments are included. It will strip out iPhone-related signatures while keeping the rest of the message (Blackberry signatures will also be stripped...see the gem for the full list).

```
---
ignore:
 text/plain:
 - /^\(no subject\)$/i
 multipart/mixed:
 - "/^Attachment: /i"
transform:
  text/plain:
  - - /\A(.*?)Sent from my iPhone$/im
    - "\1"
```

The Helio carrier's configuration is also illustrative. Its `ignore` section contains filenames of gifs that should be ignored. It also has a regular expression that matches HTML that should be ignored. Finally it has a `transform` section that is used to extract any real text that the user typed in their message as Helio intermingles that text with their own branding.

```
---
```

```
ignore:
  image/gif:
  - top.gif
  - bottom.gif
  - middle_img.gif
  text/html:
   - /<html>\s+<head>\s+\<meta http-equiv=\"Content-Type\"
content=\"text\/html; charset=iso-8859-1\">\s+<title>MMS
Email<\/title>/im
transform:
  text/plain:
  - - /Note:\s{1,2}.*?\s+message:\s{1,2}(.+)$/m
    - "\1"
```

ALIASES

Additionally, cellular carriers tend to be promiscuous in adding domains for which they provide service. Imagine that Orange begins to provide mobile service for Vatican City and MMS from these users has the pattern `a.cardinal@orange.va`. To process messages from this source using the Orange master template, add a YAML key/value of `orange.va: orangemms.net` to `conf/aliases.yml`.

In that example, the `conf/orangemss.net.yml` file serves as the master template for domains that are known to be serviced by Orange, such as `orangemms.net`, `orange.fr`, and `mmsemail.orange.pl`.

MMS2R Template Method Pattern

A configuration template will usually be sufficient to ignore and transform the content of messages. However, if you need to do more extensive manipulation by running Ruby code, you can do that, too.

The MMS2R::Media class uses a Template Method Pattern (http://en.wikipedia.org/wiki/Template_Pattern) as its architecture for processing MMS content. This allows easy customization of MMS2R's processing logic within your application's codebase. An example is MMS2R::Sprint::Media for the Sprint carrier. Sprint hosts its users' images and video on external content servers. To handle this special case, MMS2R has implemented a web scraping strategy to fetch that content. Your application doesn't need to concern itself with this since it happens automatically under the defined MMS2R::Media interface.

> See `lib/mms2r/media/sprint.rb` for an example.

The methods defined are:

Method	Description
`process`	The main method encompassing the template methods that divide the processing of a message. `MMS2R::Media` performs the file transformations by calling `process` greedily from within the `intitialize` method.
`ignore_media?`	Should the part of a certain type be ignored?
`process_media`	Retrieves media to a temporary file, returns path to the file.
`add_file`	Add the processed file to the `media` hash.
`transform_text`	Called by `process_media` and strips out advertising.

Your application will be most concerned with

Method	Description
`default_media`	Returns the most likely candidate for a single piece of user-generated media which is typically an image or video file. The file is decorated like a CGI.rb file so it can be passed to the `uploaded_data` method of an `attachment_fu` model.
`default_text`	Returns the most likely candidate for a single piece of user-generated text. The file is decorated like a CGI.rb file so it can be passed to the `uploaded_data` method of an `attachment_fu` model.
`media`	A hash of mime-type keys that have array values. Values in the array are string paths to a file of the keyed mime-type.
`purge`	Clean up all of the temporary files that MMS2R created. If your application accessed files through their paths in the `media` hash those files should be copied to a permanent location if your application is to utilize them at a later point.

Lets assume that you redefined some or all of the template methods defined above in a class named MMS2R::Media::YourCarrier Your class needs to register itself with MMS2R so that it is known that this class should be used to process content from the `mms.yourcarrier.com` carrier. This is done with the MMS2R class method `register`.

```
MMS2R.register(
  'mms.yourcarrier.com',
  MMS2R::Media::YourCarrier
)
```

Working with ActionMailer

CHAPTER 4

Rails is a full-stack web framework. It comes "out of the box" with everything you need to create a web application. But it also comes with a library to handle sending and receiving email: ActionMailer (http://api.rubyonrails.com/classes/ActionMailer/Base.html).

We will discuss both sending and receiving email with ActionMailer. MMS2R is usually used within ActionMailer to process incoming email.

Receiving MMS and Email

As has been noted, MMS is multi-part MIME encoded email. There are two aspects to receiving email with Rails.

First, you need an `ActionMailer::Base` subclass to process the mail. Second, you need a way to trigger Rails to run your subclass.

Creating an `ActionMailer::Base` subclass is easy. Use the Rails generator script to create a mailer:

```
ruby script/generate mailer MyMailer
```

Alternately, you can manually create a new file in your `models` directory, subclassing `ActionMailer::Base`.

```
class MyMailer < ActionMailer::Base

end
```

Regardless of how your mailer was created, it needs to have a `receive` method that takes one argument: a `TMail::Mail` instance.

```ruby
action_mailer_base_subclass_example.rb
class IncomingMailHandler < ActionMailer::Base

  def receive(email)
    logger.info("Got a mail about: #{mail.subject}")
    puts "I can has email?"
  end

end
```

Your `receive` method can do whatever you need to do with your mail.

> ActionMailer writes a copy of the entire email to the log when at the `info` level (this is the default in production mode). If your application receives a lot of email, your log may fill up quickly. You may want to increase the level to `warn`, `error`, or `fatal`.

Quick and Dirty

To trigger your `ActionMailer::Base` subclass, you need to invoke Rails. One way to do this is with a procmail rule as described on the Rails wiki (http://wiki.rubyonrails.org/rails/pages/HowToReceiveEmailsWithActionMailer).

Using a procmail rule to start a Rails process with `script/runner` is a basic starting point. Invoking `script/runner` can also be accomplished with other mail transfer agents such as QMail (http://qmail.org). We'll leave investigating other alternatives to you.

However, There are a few disadvantages to this approach.

First, you must have a mail server installed on your Rails web server (or vice versa). Depending on your hosting setup, this may not be the case. Or you may want to let someone else have the "fun" of configuring an MTA and making sure that you're not running as an open relay for spammers. Or, you may want to hook into an existing email service such as Gmail (http://gmail.com).

Second, this will create a new Rails process for each email received. A single Rails process uses 30-50 MB of RAM. If you get a lot of email, this will mean a lot of Rails processes will be spun up and then almost immediately thrown away.

Third, you can easily lose messages if you are in the middle of a database migration or if your application throws an error. It would be much better to have a solution that can retry if errors occur during any part of the process.

A better approach is to periodically fetch your mail from an IMAP or POP server with a long-running process. This addresses both of the problems above, but introduces a new one: keeping the mail fetcher running.

OTHER OPTIONS

If you can configure your MTA and are concerned about needlessly spinning up Rails processes but don't want to run a daemon, check out Craig Ambrose's solution (http://blog.craigambrose.com/past/2008/2/9/respond_toemail_or_how_to_handle) in which a Ruby script triggered by incoming email sends an HTTP POST to your (already running) Rails application.

Way back in 2006, Rails Podcast (http://podcast.rubyonrails.org/programs/1/episodes/billmonk) featured the founders of BillMonk (https://www.billmonk.com) who also use a similar setup to keep track of IOUs between friends.

Another solution is to use cron to periodically check for email. An example of this is presented below.

Daemonizing ActionMailer

Dan Weinand and Luke Francl wrote a Rails plugin called Fetcher (http://slantwisedesign.com/rdoc/fetcher) to accomplish this. It generates code and a script to daemonize an instance of your Rails application, fetch mail from a mail source (POP and IMAP references are provided), and then pass the retrieved mail into the `receive` method of your designated ActionMailer. The daemonizing module that it utilizes is based on an example that Sharon Rosner (http://snippets.dzone.com/posts/show/2265) posted at DZone.

Here is an example of installing the Fetcher plugin and generating a daemon that we'll name MailerDaemon.

```
rails myapp
cd myapp
ruby script/plugin install \
  svn://rubyforge.org/var/svn/slantwise/fetcher/trunk
ruby script/generate fetcher_daemon MailerDaemon

# create  config/mailer_daemon.yml
# create  script/mailer_daemon_fetcher
# create  /lib/daemon.rb
```

First, the fetcher's `receiver` should be modified to point to your `ActionMailer::Base` subclass. If you open `script/mailer_daemon_fetcher`, you'll see that the default points to `nil`:

```
# Add your own receiver object below
@fetcher = Fetcher.create({:receiver => nil}.merge(@config))
```

Using the example above, the receiver should be configured to use `IncomingMailHandler`:

```
  options = {
    :receiver => IncomingMailHandler
  }.merge(@config)
  @fetcher = Fetcher.create(options)
```

Configuring the fetcher

The fetcher daemon generator drops a stub configuration in `conf/mailer_daemon.yml`. The configuration is a YAML file with connection details for the development, test, and production Rails environments (similar to `database.yml`). You will need to configure the fetcher to use a POP3 or IMAP server with a username, password, and authentication method.

The fetcher supports the following configuration options:

Option	Description
`type`	POP or IMAP
`server`	The IP address or domain name of the server
`port`	The port to connect to (defaults to the standard port for the type of server)
`ssl`	Set to any value to use SSL encryption
`username`	The username used to connect to the server
`password`	The password used to connect to the server

Option	Description
authentication	The authentication scheme to use (IMAP only). Supports LOGIN, CRAM-MD5, and PASSWORD (defaults to PLAIN)
use_login	Set to any value to use the LOGIN command instead of AUTHENTICATE. Some servers, like GMail, do not support AUTHENTICATE (IMAP only).
sleep_time	The number of seconds to sleep between fetches (defaults to 60 seconds)
processed_folder	The name of a folder to move mail to after it has been processed (IMAP only). If not specified, mail is deleted.
error_folder	The name a folder to move mail that causes an error during processing (IMAP only). Defaults to bogus.

A typical configuration entry looks like this:

```
development:
  type: imap
  server: imap.gmail.com
  ssl: true
  username: user@domain.com
  password: hackme
  receiver: IncomingMailHandler
  port: 993
```

```
    use_login: true
```

This example shows how to connect to GMail IMAP (note the use of `use_login`). No `processed_folder` has been specified, so messages will be deleted after processing. The default `error_folder` named *bogus* will be used for mail that causes problems.

Running the fetcher daemon

After generating and configuring the fetcher daemon, you are ready to start processing mail.

To start the daemon, run the start command with the appropriate environment:

```
RAILS_ENV=production ruby script/mailer_daemon_fetcher start
```

By default, the fetcher daemon deletes email after processing it. Don't point it at your personal mailbox!

To stop the daemon, use the stop command:

```
RAILS_ENV=production ruby script/mailer_daemon_fetcher stop
```

To automate this process when you deploy your application, you should write an after deploy task to stop and start the fetcher using Capistrano (http://capify.org) or Vlad (http://rubyhitsquad.com/Vlad_the_Deployer.html).

Since the fetcher is a long-running process, it will not pick up changes to your models while running in production mode.

Therefore, it's important to restart the fetcher after deployment.

Keeping the daemon running

To keep the fetcher running, you will need process monitoring tool such as God (http://god.rubyforge.org) or Monit (http://www.tildeslash.com/monit). When the fetcher daemon starts, it writes out a PID file in the Rails log directory. Both monitoring systems will read the PID file to know which process in the operating system is to be monitored. If the process crashes, the process monitor can restart it. In the example above, this file would be called `log/MailerDaemonFetcherDaemon.pid`.

MONIT

An example monitoring task for Monit would be as follows.

```
check process mailer_daemon_fetcher with pidfile /app/
shared/log/MailerDaemonFetcher.pid
        start program = "/usr/local/bin/ruby /app/current/
script/mailer_daemon_fetcher start production"
        stop program = "/usr/local/bin/ruby /app/current/
script/mailer_daemon_fetcher stop production"
        if 5 restarts within 5 cycles then timeout
```

Monit runs in a very limited environment, so you must use full paths to executables and directories. Double check that the commands point to the right locations.

GOD

There is a default God script for Mongrels on the God project page (http://god.rubyforge.org) . The following is the start/stop/restart stanza that would be used with the script/mailer_daemon_fetcher

daemon.god

```
FETCHER_SCRIPT = "#{RAILS_ROOT}/script/mailer_daemon_fetcher"
w.name = "fetcher-daemon"
w.interval = 30.seconds # default
w.start   = "#{FETCHER_SCRIPT} start"
w.stop    = "#{FETCHER_SCRIPT} stop"
w.restart = "#{FETCHER_SCRIPT} restart"
w.start_grace = 10.seconds
w.restart_grace = 10.seconds
w.pid_file = File.join(RAILS_ROOT, 'log',
'MailerDaemonFetcher.pid')

w.behavior(:clean_pid_file)
```

Starting up when the system boots

To make sure your fetcher daemon starts when you server reboots, you will need an `init.d` script. You may already have one for your Rails application which can be modified. If God is being used on the system, then a master init.d script would be created for God, and the God daemon would then be responsible to starting Mongrels, Fetcher daemons, etc.

Fetching email with cron

As we've seen above, a daemon ensures only one Rails process is started to process your mail, but keeping it running can be a pain.

Another solution is to use `cron`, the old standby for running background tasks. The fetcher plugin can be used here as well, making it very easy to write a cron-based email solution. The advantage of cron is that it runs once a minute, so you don't have to worry about keeping your fetcher running for a long period of time. This tends to make it more reliable. The disadvantages include the slower processing time (you can only check for email at most once a minute) and frequently creating and disposing of Rails processes.

Another disadvantage of `cron` is that it has no concept of process synchronization, so you could have multiple mail fetchers running simulatenously. Fortunately, that is easy to fix with the Lockfile (http://raa.ruby-lang.org/project/lockfile) gem.

```
sudo gem install lockfile
```

Using Lockfile, we create a guard around the processing code, and rescue and ignore the exception that occurs when another cron-spawned process tries to start up.

cron_fetcher.rb
```ruby
begin
  Lockfile.new('cron_mail_fetcher.lock', :retries => 0) do
    config = YAML.load_file("#{RAILS_ROOT}/config/mail.yml")
    config = config[RAILS_ENV].to_options

    puts "Running MailFetcher in #{RAILS_ENV} mode"
    options = { :receiver => MailReceiver }
    fetcher = Fetcher.create(options.merge(config))
    fetcher.fetch

    puts "Finished running MailFetcher in #{RAILS_ENV} mode"
  end
rescue Lockfile::MaxTriesLockError => e
  puts "Another fetcher is already running. Exiting."
end
```

Next, configure cron so this code is run with script/runner once a minute.

```
# +---------------- minute (0 - 59)
# |  +------------- hour (0 - 23)
# |  |  +---------- day of month (1 - 31)
# |  |  |  +------- month (1 - 12)
# |  |  |  |  +---- day of week (0 - 6) (Sunday=0 or 7)
# |  |  |  |  |

  *  *  *  *  *  cd /u/apps/your_app/current; RAILS_ENV=production script/runner \
                   ./script/cron_fetcher.rb 2>&1 >> /u/apps/your_app/current/log/cron_fetcher.log
```

This is stored in source control in a file called `crontab`. Upon deployment, Capistrano is used to replace the server's `crontab` entry.

cron_capistrano.rb

```ruby
after :symlink, :configure_crontab

desc "Install crontab from the repository"
task :configure_crontab, :roles => :app do
  run "crontab #{current_path}/config/crontab"
end
```

> This example assumes that you are deploying to one server. If you have a cluster of servers, you could define a utility role and restrict it to a single server.

Getting scripts running under `cron` can be frustrating because of the lack of feedback. Here are some tips:

- Fully specify all paths.
- Redirect STDOUT and STDERR to a file for easier process monitoring.
- Test your scripts on the command line, just as they will be run from cron.
- Double check that your server's time is the same as what you think it is!

- Read the server's local mail account, which sometimes logs cron-related errors.

Writing a Fetcher::Base subclass

The `Fetcher::Imap` implementation found in `vendor/plugins/fetcher/lib/fetcher/imap.rb` should work for most situations (and patches are welcome!) but if needed you can either modify or create your own implementation that adds extra features.

To create a `Fetcher::Base` subclass, you need to implement four methods.

Method	Description
`establish_connection`	Connect to the server. Called at the beginning of the fetcher cycle.
`get_messages`	Retrieve the messages from the server and pass each one to `process_message`.
`close_connection`	Close the connection. Called after processing the messages, ending the fetcher cycle.
`handle_bogus_message(message)`	Called when the receiver throws an exception processing a message. This could move the message to an error folder, or log it.

Take the following example for `Fetcher::Imap2`.

It will use GMail as its IMAP source and is able to recover when your application experiences network problems while communicating with GMail. Additionally, when `Fetcher::Imap2` encounters an error it will place the mail in an IMAP folder named *errors*. When `Fetcher::Imap2` successfully processes a mail it will place the original in a folder named *processed*.

In this manner you can use GMail to provide storage for the mail that you've processed. If there is an error, you will have a copy of the mail involved with the error as it may illustrate a bug in your application and can be processed later by moving it to the INBOX.

Here is the class definition:

```
module Fetcher
  class Imap2 < Base
```

Here is a custom `get_messages` method that moves messages to the *processed* folder if the message is successfully fetched.

fetcher_imap2.rb
```
# Retrieve messages from server
def get_messages
  uids = nil
  begin
    @connection.select('INBOX')
    uids = @connection.uid_search(['ALL'])
  rescue => err
    @logger.error err
    @logger.error "#{err.backtrace.join("\n")}"
  end
  return if uids.nil? || uids.size == 0
  messages = @connection.uid_fetch(uids, [
    'ENVELOPE', 'FLAGS', 'INTERNALDATE'
  ])
  require 'parsedate'
  messages.sort_by do |data|
    ParseDate.parsedate(data.attr['INTERNALDATE'])
  end.each do |data|
    begin
      uid = data.attr['UID']
      msg = @connection.uid_fetch(uid, ['RFC822']).first
      process_message(msg.attr['RFC822'])
    rescue => err
      @logger.error err
      @logger.error "#{err.backtrace.join("\n")}"
      handle_bogus_message(uid)
    else
      # copy message into the copy folder
      @connection.create('processed') rescue
      @connection.uid_copy([uid], "processed")
    end
    # Mark message as deleted
    @connection.uid_store(uid, "+FLAGS", [:Seen, :Deleted])
  end
end
```

Finally, we move messages to the *errors* folder if there has been an error fetching the message.

fetcher_imap2.rb

```
  # Store the message for inspection if the receiver errors
  def handle_bogus_message(uid)
    @connection.create('errors') rescue
    @connection.uid_copy([uid], "errors")
    @connection.uid_store(uid, "+FLAGS", [:Seen, :Deleted])
  end
```

In order to use this custom class, you would configure `conf/mailer_daemon.yml` to point to the custom `imap2` class. The `type` key which has a value of `imap2` corresponds to your `Fetcher::Imap2` class.

```
  production:
    type: imap2
    receiver: MailReceiver
    server: imap.gmail.com
    port: 993
    ssl: true
    username: me
```

```
password: hackme
```

Further Integration

CHAPTER 5

MMS2R works well with other libraries and plugins. Here are a few you might want to work with.

MMS2R and attachment_fu

attachment_fu (http://svn.techno-weenie.net/projects/plugins/attachment_fu/README) is a popular Rails file upload plugin written by Rick Olson (http://techno-weenie.net). Since MMS2R is merely one part of the flow of messages, images, and other attachments, we've customized MMS2R to work smoothly with attachment_fu.

When using attachment_fu, you set the form upload data to a virtual attribute called `uploaded_data`. Under the covers, CGI.rb adds some metadata from the HTTP request to the uploaded file, including MIME type and the original filename. attachment_fu uses this to process your file according to the rules you've set up.

MMS2R also adds these metadata to the files it extracts from mail messages. When they are accessed through the `default_media` and `default_text` methods on MMS2R::Media, the resulting filehandle looks just like a CGI upload object. In this regard MMS2R integrates seamlessly with ActiveRecord models that use attachment_fu.

attachment_fu_example.rb
```ruby
# An ActiveRecord model decorated for attachment_fu.
class Photo < ActiveRecord::Base
  has_attachment :content_type => :image,
                 :storage => :file_system
end

# Process a mail message
mms = MMS2R::Media.new(mail_message)
```

```
  media = mms.default_media

  # Send the attachment to the model
  Photo.create!(:uploaded_data => media) if
    media.content_type.split('/').first == 'image'
```

The end result is that you can use all the features of attachment_fu to post-process email and MMS payloads. Here are a few ideas to get you started:

- Resize images or make thumbnails.
- Upload attachments to Amazon S3.
- Store attachment metadata in the database for further processing.
- Store word processing documents and import text to your content management system.

Sending & Receiving SMS

Sending and receiving from a SMS gateway service such as Open-Market (http://www.openmarket.com) is a matter of POSTing or receiving XML from their service and parsing it in a manner that is suitable for your application. There are any number of other ways to send and receive SMS, from a proxy service solution to hosting a Linux Kannel SMS Gateway server (http://www.kannel.org) Your options are virtually unlimited, so the cost of maintenance of the service for your application should be considered before investing a lot of time and money into your solution.

Dave Myron of Contentfree (http://contentfree.com) provides this example of formating a SMS message and posting to the OpenMarket gateway as follows.

- Builder is used to format the XML that OpenMarket's gateway expects.

- account_id and account_password in the configuration Hash are your application's OpenMarket credentials.
- source_number is your application's number that the recipient will reply to.
- service is the OpenMarket service that accepts the Net::HTTP POST.

openmarket_example.rb
```ruby
def send_sms_message(message, to_number, config)

  destination_number =
    (to_number =~ /^\+/) ? to_number : '+' + to_number

  builder = Builder::XmlMarkup.new
  xml_data = builder.request({
    :version => '3.0',
    :protocol => 'wmp',
    :type => 'submit'
  }) do |r|

    r.user :agent => 'XML/SMS/1.0.0'
    r.account({
      :id => config['account_id'],
      :password => config['account_password']
    })
    r.source :ton => '0', :address => config['source_number']
    r.destination({
      :ton => '0',
      :address => destination_number
    })
    r.message :text => message

  end

  service_uri = URI.parse(config['service'])
  Net::HTTP.post_form(service_uri, { 'xml' => xml_data })

  xml_data
end
```

ActionSMS and other plugins

A Rails plugin called ActionSMS (http://open.movilforum.com/wiki/index.php/ActionSMS) exists that adds SMS sending ability to `ActionMailer`. The plugin appears to be in a beta state and its documentation is in Spanish. Also its reference gateway is the Spanish Movistar (http://www.movistar.es) service.

That said, it does provide good example of integrating SMS functionality into `ActionMailer`, which keeps application logic within Rails conventions. The code (http://action-sms.googlecode.com/svn/tags/action_sms) for `ActionSMS` is hosted on Google Code.

Dealing with spam

Likely you'll have to deal with SPAM that is targeted to the mail addresses that represents your application and eventually passes through its `ActionMailer#receive` method. A few simple solutions you might consider to combat spam are:

- SpamAssassin (http://spamassassin.apache.org) – pass incoming mail through a SpamAssassin process that you control. SpamAssassin can be customized for your application's specific needs.
- GMail (http://mail.google.com) – route your application's mail through GMail. GMail has excellent spam controls and can be trained to handle special cases for your application. Having a human inspect GMail's spam bin will ensure that the mail that your application receives is free of spam.
- CRM114 – the Controllable Regex Mutilator (http://crm114.sourceforge.net) is a type of machine learning algorithm that can have nearly 99.9 percent accuracy in spam prediction. See the CRM114 Wiki (http://crm114.sourceforge.net/wiki) for examples of implementing the algorithm in conjunction with your application. CRM114 Wikipedia

page (http://en.wikipedia.org/wiki/CRM114_%28program%29)

Testing

CHAPTER 6

There many examples of testing mail that is sent with ActionMailer. However, information about testing the reception of mail is sparse. Fortunately, it's quite easy.

Consider a simple ActionMailer that receives MMS and creates a `Picture` object with attachment_fu.

First, create the receiver.

```
ruby script/generate mailer MailReceiver
```

Next, define the behavior in the `MailReceiver#receive` method. In this example, an attachment_fu based `Picture` with a `title` is created.

mail_receiver.rb

```
require 'mms2r'

class MailReceiver < ActionMailer::Base

  def receive(mail)
    mms = MMS2R::Media.new(mail)
    title =
      mms.subject.empty? ? "default title" : mms.subject
    Picture.create!(:uploaded_data => mms.default_media,
                    :title => title)
  end

end
```

If your code writes images to disk, as this one does, you may need to clean them up afterwards in a `teardown` method (Test::Unit) or `after(:all)` block (RSpec).

Test::Unit

When we generated MailReceiver, the shell of a test was created in test/unit/mail_receiver_test.rb for us to expand upon. To test the functionality of MailReceiver, we want to check that a picture is created and that its title is the same as the original subject of the MMS.

To test this, we will need a sample MMS message. You can get one by sending an MMS from your phone to your email address, then saving the raw text to a file such as sample-mms.mail. In a real project, you'll probably need many samples from many different carriers, so a better naming scheme with subdirectories is in order.

It is also helpful to write re-usable helper code to read the sample message. This can be placed in the lib directory and will be included by tests that need it.

mail_fixture.rb
```
module MailFixture
  FIXTURES_PATH = File.dirname(__FILE__) + '/../fixtures'
  CHARSET = "utf-8"

  def read_fixture(fixture)
    IO.read("#{MailFixture::FIXTURES_PATH}/mmses/#{fixture}")
  end

end
```

Now our test becomes quite simple:

mail_receiver_test.rb
```
class MailReceiverTest < ActionMailer::TestCase

  include MailFixture

  def test_picture_created_with_title_from_mms_subject
    mms = read_fixture('sample-mms.mail')
    assert_difference 'Picture.count', 1 do
      MailReceiver.receive(mms)
    end

    pic = Picture.find(Picture.maximum(:id))

    assert_equal TMail::Mail.parse(mms).subject, pic.title
  end

end
```

You should also write a test for the default subject behavior, but this is left as an exercise for the reader.

RSpec

RSpec (http://rspec.info) is is a Behaviour Driven Development framework. It is not packaged in the standard Rails installation but it does install as a plugin and has a generator to enable using it in your project.

In our specification for the behavior of `MailReceiver`, we check that a picture is created. We also confirm that its title is the same as the original subject of the MMS. Again, `MailFixture` (see above) is used to load the sample MMS for parsing.

mail_receiver_spec.rb

```
describe 'Ingesting MMS files' do
  include MailFixture

  it 'creates a picture and extracts title from MMS' do
    mms = read_fixture('sample-mms.mail')
    lambda {
      MailReceiver.receive(mms)
    }.should change(Picture, :count).by(1)

    pic = Picture.find(Picture.maximum(:id))

    pic.title.should == TMail::Mail.parse(mms).subject
  end
```

```
end
```

Mocking Ruby Network Libraries

RSpec is great because it makes it easy to mock objects and thus verify that various methods are being invoked at different times (i.e. testing expected behavior). When we are testing our applications at the edge of where they are interacting with the network, it's not always clear that we can actually mock Ruby's network libraries.

Take the following Mocha (http://mocha.rubyforge.org) mocking example. We mock the behavior of the `Net::HTTP.post_form` method but our real code will never actually call out to the network since `Net::HTTP` is mocked.

mocha_example.rb
```
def test_openmarket_sms_post
  uri = URI.parse('http://wmp.simplewire.com/wmp')
  builder = mock()
  xml_data = mock()
  Builder::XmlMarkup.expects(:new).once.returns(builder)
  builder.expects(:request).once.returns(xml_data)
  Net::HTTP.expects(:post_form).once.with(uri, { 'xml' =>
xml_data })

  assert_nothing_raised { send_sms_message }
end
```

Advanced topics

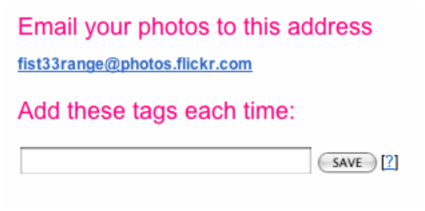
CHAPTER 7

Strategies for matching

If you are building a site that allows users to interact with it through email, you need a way to match user accounts with email addresses. There are two ways to do this.

Unique Address

One way is to follow Flickr's example and generate a unique email address for each user.


Email your photos to this address

fist33range@photos.flickr.com

Add these tags each time:

FIG. 1 A **secret address for each user**

When you receive an email addressed to a user's secret, unique address, you know it is from them.

```
  user = User.find_by_secret_address(mail.to.first.downcase)
```

To do this, you need to configure your mail server with a catch-all address. Because this dramatically increases the amount of SPAM you will receive, some mail providers are hesitant to do this.

PLUS (OR MINUS) ADDRESSING

Another option is to use *plus or minus addressing*. Many email servers will deliver mail to the `bart` user if a message is addressed to bart+special@example.com.

```
bart+0028272@example.com
bart+2928273@example.com
bart+9983780@example.com
```

So you could create a unique address with the plus qualifier and read it from a single, non-catchall inbox.

However, this may be confusing for some users. It may be appropriate for an automated system as opposed to an address that users will type into their phone or email program.

- Plus or Minus Addressing (http://en.wikipedia.org/wiki/E-mail_address#Plus_.28or_Minus.29_addressing)
- QMail extension addresses (http://www.lifewithqmail.org/lwq.html#extension-addresses)

Activation Code

The second way to match users to email addresses is to have your users prove they own an email address.

You do this by generating some unique code and having the user send in that code from their email or mobile device. By correlating the code with the email it came from, you know that the user with that code owns that device. Twitter handles this by having users SMS in a random, unique code from their phone.

Twitter's random code is made only of characters that can be typed with a single key press on a mobile phone: `a`, `d`, `h`, `j`, `m`, `p`, `t`, and `w`.

Email address claiming can work one of two ways.

- Email first. When you receive an email from an unknown user, save the `From:` address and reply with a code. The new user goes to the site and finishes registration with the code.
- Web first. A user registers on the site, and then activates their email address by sending in a code displayed on the site.

Don't try to infer email addresses

One thing you should *not* do is assume that you can match up a user's mobile email address based on their phone number and SMS carrier.

We originally tried this because many phone carriers have `your_phone_number@carrier.com` type of email `From:` addresses. But not all of them do. And some carriers have different email addresses for SMS and MMS messages sent to an email address.

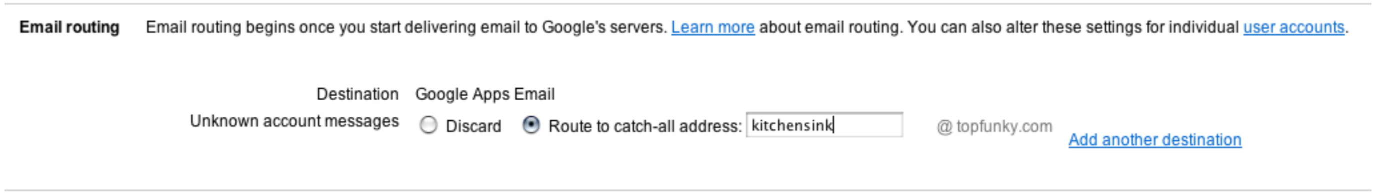
As examples, US T-Mobile SMS messages sent to an email address come from an address like `555123456789@tmomail.net`, whereas the same phone sending an MMS message to an email address comes from `1555123456789@tmomail.net`. Verizon text messages don't have numbers at all, but Verizon MMS messages do. And so on.

Many carriers use the some variation on the `phone_number@gateway.com` pattern, but by no means all. You can find an extensive list of email to SMS gateways (http://en.wikipedia.org/wiki/SMS_gateways) on Wikipedia.

Server setup for catchall email address

Lets assume that you would like to create an ActionMailer receiver for support requests. When it receives an email to *support@example.com*, a support ticket is created. Otherwise, the address of the `To:` portion of the mail received is treated as a lookup token into the support system. If the token is not found, then the mail is ignored.

First, you'll need to setup your mail server to funnel all unmatched email to a specific mailbox. This will differ based on the mail software you are using.



Email routing Email routing begins once you start delivering email to Google's servers. Learn more about email routing. You can also alter these settings for individual user accounts.

Destination Google Apps Email

Unknown account messages ○ Discard ● Route to catch-all address: kitchensink @ topfunky.com

Add another destination

FIG. J **Gmail for domains catch-all setup**

A catch-all email address will receive a huge amount of email, most of which will be SPAM.

Next, you'll need to filter incoming messages based on the `To` address. The `SupportMailer#receive` method is just a multiplexer for messages mailed to the support system.

support_mailer.rb
```ruby
def receive(mail)
  if(mail.to.first.downcase == 'support@example.com')
    ticket = SupportTicket.create_from_mail(mail)
    SupportMailer.deliver_new_ticket(ticket)
  else
    token = mail.to.first.downcase.split('@').first
    ticket = SupportTicket.find_by_token(token)
    return unless ticket
    ticket.append_reply(mail)
    SupportMailer.deliver_reply_received(mail, ticket)
    end
  end
```

The response that the support ticket was created could be

```
support_mailer.rb
def new_ticket(ticket)
  @subject    = "MyCo: Ticket [#{ticket.token}] created"
  @from       = "#{ticket.token}@support.example.com"
  @recipients = ticket.customer_mail
  @body["ticket"]  = ticket
end
```

The customer's reply was appended to the support thread. We'll assume that notification to the support personnel takes place through the ticket model. And a brief response is sent back to the customer.

```
support_mailer.rb
def reply_received(mail, ticket)
  @subject    = "MyCo: Ticket [#{ticket.token}] received"
  @from       = "#{ticket.token}@support.example.com"
  @recipients = ticket.customer_mail
  @body["ticket"]  = ticket
end
```

The code used to trigger a reply by the support personnel from a controller or model could then be

```
SupportMailer.deliver_reply_to_thread(ticket, message)
```

And the setup of replying to the customer would be

```
support_mailer.rb
def reply_to_customer(ticket, message)
  @subject    = "MyCo: Ticket [#{ticket.token}] response"
  @from       = "#{ticket.token}@support.example.com"
```

```
    @recipients = ticket.customer_mail
    @body["ticket"]  = ticket
    @body["message"]  = message
  end
```

Additionally you would probably want a generic acknowledgement to be sent immediately to the customer whenever a mail from them is received. This will happen when the support ticket is first created but it should also happen when a customer's response is appended to the support thread in `append_to_thread`. We'll leave that for you to implement perhaps as an after create or after update ActiveRecord filter.