The Swiftmailer Book

generated on July 24, 2012

The Swiftmailer Book

This work is licensed under the "Attribution-Share Alike 3.0 Unported" license (*http://creativecommons.org/licenses/by-sa/3.0/*).

You are free **to share** (to copy, distribute and transmit the work), and **to remix** (to adapt the work) under the following conditions:

- **Attribution**: You must attribute the work in the manner specified by the author or licensor (but not in any way that suggests that they endorse you or your use of the work).
- **Share Alike**: If you alter, transform, or build upon this work, you may distribute the resulting work only under the same, similar or a compatible license. For any reuse or distribution, you must make clear to others the license terms of this work.

The information in this book is distributed on an "as is" basis, without warranty. Although every precaution has been taken in the preparation of this work, neither the author(s) nor SensioLabs shall have any liability to any person or entity with respect to any loss or damage caused or alleged to be caused directly or indirectly by the information contained in this work.

Contents at a Glance

Chapter 1
Introduction

Swift Mailer is a component-based library for sending e-mails from PHP applications.

Organization of this Book

This book has been written so that those who need information quickly are able to find what they need, and those who wish to learn more advanced topics can read deeper into each chapter.

The book begins with an overview of Swift Mailer, discussing what's included in the package and preparing you for the remainder of the book.

It is possible to read this user guide just like any other book (from beginning to end). Each chapter begins with a discussion of the contents it contains, followed by a short code sample designed to give you a head start. As you get further into a chapter you will learn more about Swift Mailer's capabilities, but often you will be able to head directly to the topic you wish to learn about.

Throughout this book you will be presented with code samples, which most people should find ample to implement Swift Mailer appropriately in their own projects. We will also use diagrams where appropriate, and where we believe readers may find it helpful we will discuss some related theory, including reference to certain documents you are able to find online.

Code Samples

Code samples presented in this book will be displayed on a different colored background in a monospaced font. Samples are not to be taken as copy & paste code snippets.

Code examples are used through the book to clarify what is written in text. They will sometimes be usable as-is, but they should always be taken as outline/pseudo code only.

A code sample will look like this:

Listing 1-1 Listing 1-2
```
1 class AClass
2 {
```

```
3     ...
4 }
5
6 //A Comment
7 $obj = new AClass($arg1, $arg2, ... );
8
9 /* A note about another way of doing something
10 $obj = AClass::newInstance($arg1, $arg2, ... );
11
12 */
```

The presence of 3 dots `...` in a code sample indicates that we have left out a chunk of the code for brevity, they are not actually part of the code.

We will often place multi-line comments `/* ... */` in the code so that we can show alternative ways of achieving the same result.

You should read the code examples given and try to understand them. They are kept concise so that you are not overwhelmed with information.

History of Swift Mailer

Swift Mailer began back in 2005 as a one-class project for sending mail over SMTP. It has since grown into the flexible component-based library that is in development today.

Chris Corbyn first posted Swift Mailer on a web forum asking for comments from other developers. It was never intended as a fully supported open source project, but members of the forum began to adopt it and make use of it.

Very quickly feature requests were coming for the ability to add attachments and use SMTP authentication, along with a number of other "obvious" missing features. Considering the only alternative was PHPMailer it seemed like a good time to bring some fresh tools to the table. Chris began working towards a more component based, PHP5-like approach unlike the existing single-class, legacy PHP4 approach taken by PHPMailer.

Members of the forum offered a lot of advice and critique on the code as he worked through this project and released versions 2 and 3 of the library in 2005 and 2006, which by then had been broken down into smaller classes offering more flexibility and supporting plugins. To this day the Swift Mailer team still receive a lot of feature requests from users both on the forum and in by email.

Until 2008 Chris was the sole developer of Swift Mailer, but entering 2009 he gained the support of two experienced developers well-known to him: Paul Annesley and Christopher Thompson. This has been an extremely welcome change.

As of September 2009, Chris handed over the maintenance of Swift Mailer to Fabien Potencier.

Now 2009 and in its fourth major version Swift Mailer is more object-oriented and flexible than ever, both from a usability standpoint and from a development standpoint.

By no means is Swift Mailer ready to call "finished". There are still many features that can be added to the library along with the constant refactoring that happens behind the scenes.

It's a Library!

Swift Mailer is not an application - it's a library.

To most experienced developers this is probably an obvious point to make, but it's certainly worth mentioning. Many people often contact us having gotten the completely wrong end of the stick in terms of what Swift Mailer is actually for.

It's not an application. It does not have a graphical user interface. It cannot be opened in your web browser directly.

It's a library (or a framework if you like). It provides a whole lot of classes that do some very complicated things, so that you don't have to. You "use" Swift Mailer within an application so that your application can have the ability to send emails.

The component-based structure of the library means that you are free to implement it in a number of different ways and that you can pick and choose what you want to use.

An application on the other hand (such as a blog or a forum) is already "put together" in a particular way, (usually) provides a graphical user interface and most likely doesn't offer a great deal of integration with your own application.

Embrace the structure of the library and use the components it offers to your advantage. Learning what the components do, rather than blindly copying and pasting existing code will put you in a great position to build a powerful application!

Chapter 2
Library Overview

Most features (and more) of your every day mail client software are provided by Swift Mailer, using object-oriented PHP code as the interface.

In this chapter we will take a short tour of the various components, which put together form the Swift Mailer library as a whole. You will learn key terminology used throughout the rest of this book and you will gain a little understanding of the classes you will work with as you integrate Swift Mailer into your application.

This chapter is intended to prepare you for the information contained in the subsequent chapters of this book. You may choose to skip this chapter if you are fairly technically minded, though it is likely to save you some time in the long run if you at least read between the lines here.

System Requirements

The basic requirements to operate Swift Mailer are extremely minimal and easily achieved. Historically, Swift Mailer has supported both PHP 4 and PHP 5 by following a parallel development workflow. Now in it's fourth major version, and Swift Mailer operates on servers running PHP 5.2 or higher.

The library aims to work with as many PHP 5 projects as possible:

- PHP 5.2 or higher, with the SPL extension (standard)
- Limited network access to connect to remote SMTP servers
- 8 MB or more memory limit (Swift Mailer uses around 2 MB)

Component Breakdown

Swift Mailer is made up of many classes. Each of these classes can be grouped into a general "component" group which describes the task it is designed to perform.

We'll take a brief look at the components which form Swift Mailer in this section of the book.

The Mailer

The mailer class, `Swift_Mailer` is the central class in the library where all of the other components meet one another. `Swift_Mailer` acts as a sort of message dispatcher, communicating with the underlying Transport to deliver your Message to all intended recipients.

If you were to dig around in the source code for Swift Mailer you'd notice that `Swift_Mailer` itself is pretty bare. It delegates to other objects for most tasks and in theory, if you knew the internals of Swift Mailer well you could by-pass this class entirely. We wouldn't advise doing such a thing however -- there are reasons this class exists:

- for consistency, regardless of the Transport used
- to provide abstraction from the internals in the event internal API changes are made
- to provide convenience wrappers around aspects of the internal API

An instance of `Swift_Mailer` is created by the developer before sending any Messages.

Transports

Transports are the classes in Swift Mailer that are responsible for communicating with a service in order to deliver a Message. There are several types of Transport in Swift Mailer, all of which implement the Swift_Transport interface and offer underlying start(), stop() and send() methods.

Typically you will not need to know how a Transport works under-the-surface, you will only need to know how to create an instance of one, and which one to use for your environment.

Class	**Features**	**Pros/cons**
`Swift_SmtpTransport`	Sends messages over SMTP; Supports Authentication; Supports Encryption	Very portable; Pleasingly predictable results; Provides good feedback
`Swift_SendmailTransport`	Communicates with a locally installed `sendmail` executable (Linux/UNIX)	Quick time-to-run; Provides less-accurate feedback than SMTP; Requires `sendmail` installation
`Swift_MailTransport`	Uses PHP's built-in `mail()` function	Very portable; Potentially unpredictable results; Provides extremely weak feedback
`Swift_LoadBalancedTransport`	Cycles through a collection of the other Transports to manage load-reduction	Provides graceful fallback if one Transport fails (e.g. an SMTP server is down); Keeps the load on remote services down by spreading the work
`Swift_FailoverTransport`	Works in conjunction with a collection of the other Transports to provide high-availability	Provides graceful fallback if one Transport fails (e.g. an SMTP server is down)

MIME Entities

Everything that forms part of a Message is called a MIME Entity. All MIME entities in Swift Mailer share a common set of features. There are various types of MIME entity that serve different purposes such as Attachments and MIME parts.

An e-mail message is made up of several relatively simple entities that are combined in different ways to achieve different results. All of these entities have the same fundamental outline but serve a different purpose. The Message itself can be defined as a MIME entity, an Attachment is a MIME entity, all MIME parts are MIME entities -- and so on!

The basic units of each MIME entity -- be it the Message itself, or an Attachment -- are its Headers and its body:

```
Other-Header: Another value

The body content itself
```

Listing 2-1 *Listing 2-2*

The Headers of a MIME entity, and its body must conform to some strict standards defined by various RFC documents. Swift Mailer ensures that these specifications are followed by using various types of object, including Encoders and different Header types to generate the entity.

Each MIME component implements the base `Swift_Mime_MimeEntity` interface, which offers methods for retrieving Headers, adding new Headers, changing the Encoder, updating the body and so on!

All MIME entities have one Header in common -- the Content-Type Header, updated with the entity's `setContentType()` method.

Encoders

Encoders are used to transform the content of Messages generated in Swift Mailer into a format that is safe to send across the internet and that conforms to RFC specifications.

Generally speaking you will not need to interact with the Encoders in Swift Mailer -- the correct settings will be handled by the library itself. However they are probably worth a brief mention in the event that you do want to play with them.

Both the Headers and the body of all MIME entities (including the Message itself) use Encoders to ensure the data they contain can be sent over the internet without becoming corrupted or misinterpreted.

There are two types of Encoder: Base64 and Quoted-Printable.

Plugins

Plugins exist to extend, or modify the behaviour of Swift Mailer. They respond to Events that are fired within the Transports during sending.

There are a number of Plugins provided as part of the base Swift Mailer package and they all follow a common interface to respond to Events fired within the library. Interfaces are provided to "listen" to each type of Event fired and to act as desired when a listened-to Event occurs.

Although several plugins are provided with Swift Mailer out-of-the-box, the Events system has been specifically designed to make it easy for experienced object-oriented developers to write their own plugins in order to achieve goals that may not be possible with the base library.

Chapter 3
Installing the Library

Installing Swift Mailer is trivial. Usually it's just a case of uploading the extracted source files to your web server.

Installing from PEAR

If you want to install Swift Mailer globally on your machine, the easiest installation method is using the PEAR channel.

To install the Swift Mailer PEAR package:

- Run the command `pear channel-discover pear.swiftmailer.org`.
- Then, run the command `pear install swift/swift`.

Installing from a Package

Most users will download a package from the Swift Mailer website and install Swift Mailer using this.

If you downloaded Swift Mailer as a `.tar.gz` or `.zip` file installation is as simple as extracting the archive and uploading it to your web server.

Extracting the Library

You extract the archive by using your favorite unarchiving tool such as `tar` or 7-Zip.

You will need to have access to a program that can open uncompress the archive. On Windows computers, 7-Zip will work. On Mac and Linux systems you can use `tar` on the command line.

To extract your downloaded package:

- Use the "extract" facility of your archiving software.

The source code will be placed into a directory with the same name as the archive (e.g. Swift-4.0.0-b1).

The following example shows the process on Mac OS X and Linux systems using the `tar` command.

Listing 3-1

```
$ ls
Swift-4.0.0-dev.tar.gz
$ tar xvzf Swift-4.0.0-dev.tar.gz
Swift-4.0.0-dev/
Swift-4.0.0-dev/lib/
Swift-4.0.0-dev/lib/classes/
Swift-4.0.0-dev/lib/classes/Swift/
Swift-4.0.0-dev/lib/classes/Swift/ByteStream/
Swift-4.0.0-dev/lib/classes/Swift/CharacterReader/
Swift-4.0.0-dev/lib/classes/Swift/CharacterReaderFactory/
Swift-4.0.0-dev/lib/classes/Swift/CharacterStream/
Swift-4.0.0-dev/lib/classes/Swift/Encoder/

  ... etc etc ...

Swift-4.0.0-dev/tests/unit/Swift/Transport/LoadBalancedTransportTest.php
Swift-4.0.0-dev/tests/unit/Swift/Transport/SendmailTransportTest.php
Swift-4.0.0-dev/tests/unit/Swift/Transport/StreamBufferTest.php
$ cd Swift-4.0.0-dev
$ ls
CHANGES             LICENSE.GPL     LICENSE.LGPL    README          VERSION
examples        lib             test-suite      tests
$
```

Installing from Git

It's possible to download and install Swift Mailer directly from github.com if you want to keep up-to-date with ease.

Swift Mailer's source code is kept in a git repository at github.com so you can get the source directly from the repository.

You do not need to have git installed to use Swift Mailer from github. If you don't have git installed, go to *github*[1] and click the "Download" button.

Cloning the Repository

The repository can be cloned from git://github.com/swiftmailer/swiftmailer.git using the `git clone` command.

You will need to have `git` installed before you can use the `git clone` command.

To clone the repository:

- Open your favorite terminal environment (command line).
- Move to the directory you want to clone to.
- Run the command `git clone git://github.com/swiftmailer/swiftmailer.git swiftmailer`.

The source code will be downloaded into a directory called "swiftmailer".

The example shows the process on a UNIX-like system such as Linux, BSD or Mac OS X.

Listing 3-2

```
$ cd source_code/
$ git clone git://github.com/swiftmailer/swiftmailer.git swiftmailer
Initialized empty Git repository in /Users/chris/source_code/swiftmailer/.git/
```

1. `http://github.com/swiftmailer/swiftmailer`

```
remote: Counting objects: 6815, done.
remote: Compressing objects: 100% (2761/2761), done.
remote: Total 6815 (delta 3641), reused 6326 (delta 3286)
Receiving objects: 100% (6815/6815), 4.35 MiB | 162 KiB/s, done.
Resolving deltas: 100% (3641/3641), done.
Checking out files: 100% (1847/1847), done.
$ cd swiftmailer/
$ ls
CHANGES             LICENSE.LGPL    README.git      VERSION         docs
lib             test-suite      util
LICENSE.GPL README          TODO            build.xml       examples        notes
tests
$
```

Uploading to your Host

You only need to upload the "lib/" directory to your web host for production use. All other files and directories are support files not needed in production.

You will need FTP, `rsync` or similar software installed in order to upload the "lib/" directory to your web host.

To upload Swift Mailer:

- Open your FTP program, or a command line if you prefer rsync/scp.
- Upload the "lib/" directory to your hosting account.

The files needed to use Swift Mailer should now be accessible to PHP on your host.

The following example shows show you can upload the files using `rsync` on Linux or OS X.

You do not need to place the files inside your web root. They only need to be in a place where your PHP scripts can "include" them.

Listing 3-3

```
$ rsync -rvz lib d11wtq@swiftmailer.org:swiftmailer
building file list ... done
created directory swiftmailer
lib/
lib/mime_types.php
lib/preferences.php
lib/swift_required.php
lib/classes/
lib/classes/Swift/
lib/classes/Swift/Attachment.php
lib/classes/Swift/CharacterReader.php
  ... etc etc ...
lib/dependency_maps/
lib/dependency_maps/cache_deps.php
lib/dependency_maps/mime_deps.php
lib/dependency_maps/transport_deps.php

sent 151692 bytes  received 2974 bytes  5836.45 bytes/sec
total size is 401405  speedup is 2.60
$
```

Troubleshooting

Swift Mailer does not work when used with function overloading as implemented by `mbstring` (`mbstring.func_overload` set to `2`). A workaround is to temporarily change the internal encoding to `ASCII` when sending an email:

Listing 3-4

```
if (function_exists('mb_internal_encoding') && ((int) ini_get('mbstring.func_overload'))
& 2)
{
  $mbEncoding = mb_internal_encoding();
  mb_internal_encoding('ASCII');
}

// Create your message and send it with Swift Mailer

if (isset($mbEncoding))
{
  mb_internal_encoding($mbEncoding);
}
```

Listing 3-5

Chapter 4
Getting Help

There are a number of ways you can get help when using Swift Mailer, depending upon the nature of your problem. For bug reports and feature requests create a new ticket in Github. For general advice ask on the Google Group (swiftmailer).

Submitting Bugs & Feature Requests

Bugs and feature requests should be posted on Github.

If you post a bug or request a feature in the forum, or on the Google Group you will most likely be asked to create a ticket in *Github*[1] since it is the simply not feasible to manage such requests from a number of a different sources.

When you go to Github you will be asked to create a username and password before you can create a ticket. This is free and takes very little time.

When you create your ticket, do not assign it to any milestones. A developer will assess your ticket and re-assign it as needed.

If your ticket is reporting a bug present in the current version, which was not present in the previous version please include the tag "regression" in your ticket.

Github will update you when work is performed on your ticket.

Ask on the Google Group

You can seek advice at Google Groups, within the "swiftmailer" *group*[2].

You can post messages to this group if you want help, or there's something you wish to discuss with the developers and with other users.

This is probably the fastest way to get help since it is primarily email-based for most users, though bug reports should not be posted here since they may not be resolved.

---

1. `https://github.com/swiftmailer/swiftmailer/issues`
2. `http://groups.google.com/group/swiftmailer`

Chapter 5
Including Swift Mailer (Autoloading)

If you are using Composer, Swift Mailer will be automatically autoloaded.

If not, you can use the built-in autoloader by requiring the `swift_required.php` file:

Listing 5-1

```
require_once '/path/to/swift-mailer/lib/swift_required.php';
```

Listing 5-2

```
/* rest of code goes here */
```

If you want to override the default Swift Mailer configuration, call the `init()` method on the `Swift` class and pass it a valid PHP callable (a PHP function name, a PHP 5.3 anonymous function, ...):

Listing 5-3

```
require_once '/path/to/swift-mailer/lib/swift_required.php';
```

Listing 5-4

```
function swiftmailer_configurator() {
    // configure Swift Mailer

    Swift_DependencyContainer::getInstance()->...
    Swift_Preferences::getInstance()->...
}

Swift::init('swiftmailer_configurator');

/* rest of code goes here */
```

The advantage of using the `init()` method is that your code will be executed only if you use Swift Mailer in your script.

While Swift Mailer's autoloader is designed to play nicely with other autoloaders, sometimes you may have a need to avoid using Swift Mailer's autoloader and use your own instead. Include the `swift_init.php` instead of the `swift_required.php` if you need to do this. The very minimum

include is the `swift_init.php` file since Swift Mailer will not work without the dependency injection this file sets up:

Listing 5-5 Listing 5-6

```
require_once '/path/to/swift-mailer/lib/swift_init.php';

/* rest of code goes here */
```

For PHP versions starting with 5.3 it is recommended using the native quoted printable encoder. It uses PHP's native `quoted_printable_encode()`-function to achieve much better performance:

Listing 5-7 Listing 5-8

```
Swift::init(function () {
    Swift_DependencyContainer::getInstance()
        ->register('mime.qpcontentencoder')
        ->asAliasOf('mime.nativeqpcontentencoder');
});
```

Chapter 6
Creating Messages

Creating messages in Swift Mailer is done by making use of the various MIME entities provided with the library. Complex messages can be quickly created with very little effort.

Quick Reference for Creating a Message

You can think of creating a Message as being similar to the steps you perform when you click the Compose button in your mail client. You give it a subject, specify some recipients, add any attachments and write your message.

To create a Message:

- Call the `newInstance()` method of `Swift_Message`.
- Set your sender address (`From:`) with `setFrom()` or `setSender()`.
- Set a subject line with `setSubject()`.
- Set recipients with `setTo()`, `setCc()` and/or `setBcc()`.
- Set a body with `setBody()`.
- Add attachments with `attach()`.

```
1  require_once 'lib/swift_required.php';
2
3  // Create the message
4  $message = Swift_Message::newInstance()
5
6    // Give the message a subject
7    ->setSubject('Your subject')
8
9    // Set the From address with an associative array
10   ->setFrom(array('john@doe.com' => 'John Doe'))
11
12   // Set the To addresses with an associative array
13   ->setTo(array('receiver@domain.org', 'other@domain.org' => 'A name'))
14
```

Listing 6-1

Listing 6-2

```
15    // Give it a body
16    ->setBody('Here is the message itself')
17
18    // And optionally an alternative body
19    ->addPart('<q>Here is the message itself</q>', 'text/html')
20
21    // Optionally add any attachments
22    ->attach(Swift_Attachment::fromPath('my-document.pdf'))
23    ;
```

Message Basics

A message is a container for anything you want to send to somebody else. There are several basic aspects of a message that you should know.

An e-mail message is made up of several relatively simple entities that are combined in different ways to achieve different results. All of these entities have the same fundamental outline but serve a different purpose. The Message itself can be defined as a MIME entity, an Attachment is a MIME entity, all MIME parts are MIME entities -- and so on!

The basic units of each MIME entity -- be it the Message itself, or an Attachment -- are its Headers and its body:

Listing 6-3 Listing 6-4

```
1  Header-Name: A header value
2  Other-Header: Another value
3
4  The body content itself
```

The Headers of a MIME entity, and its body must conform to some strict standards defined by various RFC documents. Swift Mailer ensures that these specifications are followed by using various types of object, including Encoders and different Header types to generate the entity.

The Structure of a Message

Of all of the MIME entities, a message -- `Swift_Message` is the largest and most complex. It has many properties that can be updated and it can contain other MIME entities -- attachments for example -- nested inside it.

A Message has a lot of different Headers which are there to present information about the message to the recipients' mail client. Most of these headers will be familiar to the majority of users, but we'll list the basic ones. Although it's possible to work directly with the Headers of a Message (or other MIME entity), the standard Headers have accessor methods provided to abstract away the complex details for you. For example, although the Date on a message is written with a strict format, you only need to pass a UNIX timestamp to `setDate()`.

Header	Description	Accessors
`Message-ID`	Identifies this message with a unique ID, usually containing the domain name and time generated	`getId()` / `setId()`
`Return-Path`	Specifies where bounces should go (Swift Mailer reads this for other uses)	`getReturnPath()` / `setReturnPath()`

Header	Description	Accessors
`From`	Specifies the address of the person who the message is from. This can be multiple addresses if multiple people wrote the message.	`getFrom()` / `setFrom()`
`Sender`	Specifies the address of the person who physically sent the message (higher precedence than `From:`)	`getSender()` / `setSender()`
`To`	Specifies the addresses of the intended recipients	`getTo()` / `setTo()`
`Cc`	Specifies the addresses of recipients who will be copied in on the message	`getCc()` / `setCc()`
`Bcc`	Specifies the addresses of recipients who the message will be blind-copied to. Other recipients will not be aware of these copies.	`getBcc()` / `setBcc()`
`Reply-To`	Specifies the address where replies are sent to	`getReplyTo()` / `setReplyTo()`
`Subject`	Specifies the subject line that is displayed in the recipients' mail client	`getSubject()` / `setSubject()`
`Date`	Specifies the date at which the message was sent	`getDate()` / `setDate()`
`Content-Type`	Specifies the format of the message (usually text/plain or text/html)	`getContentType()` / `setContentType()`
`Content-Transfer-Encoding`	Specifies the encoding scheme in the message	`getEncoder()` / `setEncoder()`

Working with a Message Object

Although there are a lot of available methods on a message object, you only need to make use of a small subset of them. Usually you'll use `setSubject()`, `setTo()` and `setFrom()` before setting the body of your message with `setBody()`.

Calling methods is simple. You just call them like functions, but using the object operator "`->`" to do so. If you've created a message object and called it `$message` then you'd set a subject on it like so:

Listing 6-5

```
require_once 'lib/swift_required.php';

$message = Swift_Message::newInstance();
$message->setSubject('My subject');
```

Listing 6-6

All MIME entities (including a message) have a `toString()` method that you can call if you want to take a look at what is going to be sent. For example, if you `echo $message->toString();` you would see something like this:

Listing 6-7

```
Message-ID: <1230173678.4952f5eeb1432@swift.generated>
Date: Thu, 25 Dec 2008 13:54:38 +1100
Subject: Example subject
From: Chris Corbyn <chris@w3style.co.uk>
```

Listing 6-8

```
5  To: Receiver Name <recipient@example.org>
6  MIME-Version: 1.0
7  Content-Type: text/plain; charset=utf-8
8  Content-Transfer-Encoding: quoted-printable
9
10 Here is the message
```

We'll take a closer look at the methods you use to create your message in the following sections.

Adding Content to Your Message

Rich content can be added to messages in Swift Mailer with relative ease by calling methods such as `setSubject()`, `setBody()`, `addPart()` and `attach()`.

Setting the Subject Line

The subject line, displayed in the recipients' mail client can be set with the `setSubject()` method, or as a parameter to `Swift_Message::newInstance()`.

To set the subject of your Message:

- Call the `setSubject()` method of the Message, or specify it at the time you create the message.

Listing 6-9 Listing 6-10

```
1  // Pass it as a parameter when you create the message
2  $message = Swift_Message::newInstance('My amazing subject');
3
4  // Or set it after like this
5  $message->setSubject('My amazing subject');
```

Setting the Body Content

The body of the message -- seen when the user opens the message -- is specified by calling the `setBody()` method. If an alternative body is to be included `addPart()` can be used.

The body of a message is the main part that is read by the user. Often people want to send a message in HTML format (`text/html`), other times people want to send in plain text (`text/plain`), or sometimes people want to send both versions and allow the recipient to chose how they view the message.

As a rule of thumb, if you're going to send a HTML email, always include a plain-text equivalent of the same content so that users who prefer to read plain text can do so.

To set the body of your Message:

- Call the `setBody()` method of the Message, or specify it at the time you create the message.
- Add any alternative bodies with `addPart()`.

If the recipient's mail client offers preferences for displaying text vs. HTML then the mail client will present that part to the user where available. In other cases the mail client will display the "best" part it can - usually HTML if you've included HTML.

Listing 6-11 Listing 6-12

```php
// Pass it as a parameter when you create the message
$message = Swift_Message::newInstance('Subject here', 'My amazing body');

// Or set it after like this
$message->setBody('My <em>amazing</em> body', 'text/html');

// Add alternative parts with addPart()
$message->addPart('My amazing body in plain text', 'text/plain');
```

Attaching Files

Attachments are downloadable parts of a message and can be added by calling the `attach()` method on the message. You can add attachments that exist on disk, or you can create attachments on-the-fly.

Attachments are actually an interesting area of Swift Mailer and something that could put a lot of power at your fingertips if you grasp the concept behind the way a message is held together.

Although we refer to files sent over e-mails as "attachments" -- because they're attached to the message -- lots of other parts of the message are actually "attached" even if we don't refer to these parts as attachments.

File attachments are created by the `Swift_Attachment` class and then attached to the message via the `attach()` method on it. For all of the "every day" MIME types such as all image formats, word documents, PDFs and spreadsheets you don't need to explicitly set the content-type of the attachment, though it would do no harm to do so. For less common formats you should set the content-type -- which we'll cover in a moment.

Attaching Existing Files

Files that already exist, either on disk or at a URL can be attached to a message with just one line of code, using `Swift_Attachment::fromPath()`.

You can attach files that exist locally, or if your PHP installation has `allow_url_fopen` turned on you can attach files from other websites.

To attach an existing file:

- Create an attachment with `Swift_Attachment::fromPath()`.
- Add the attachment to the message with `attach()`.

The attachment will be presented to the recipient as a downloadable file with the same filename as the one you attached.

Listing 6-13

```php
// Create the attachment
// * Note that you can technically leave the content-type parameter out
$attachment = Swift_Attachment::fromPath('/path/to/image.jpg', 'image/jpeg');

// Attach it to the message
$message->attach($attachment);


// The two statements above could be written in one line instead
$message->attach(Swift_Attachment::fromPath('/path/to/image.jpg'));

```

Listing 6-14

```
13 // You can attach files from a URL if allow_url_fopen is on in php.ini
14 $message->attach(Swift_Attachment::fromPath('http://site.tld/logo.png'));
```

Setting the Filename

Usually you don't need to explicitly set the filename of an attachment because the name of the attached file will be used by default, but if you want to set the filename you use the `setFilename()` method of the Attachment.

To change the filename of an attachment:

- Call its `setFilename()` method.

The attachment will be attached in the normal way, but meta-data sent inside the email will rename the file to something else.

Listing 6-15 Listing 6-16

```
1 // Create the attachment and call its setFilename() method
2 $attachment = Swift_Attachment::fromPath('/path/to/image.jpg')
3   ->setFilename('cool.jpg');
4
5
6 // Because there's a fluid interface, you can do this in one statement
7 $message->attach(
8   Swift_Attachment::fromPath('/path/to/image.jpg')->setFilename('cool.jpg')
9 );
```

Attaching Dynamic Content

Files that are generated at runtime, such as PDF documents or images created via GD can be attached directly to a message without writing them out to disk. Use the standard `Swift_Attachment::newInstance()` method.

To attach dynamically created content:

- Create your content as you normally would.
- Create an attachment with `Swift_Attachment::newInstance()`, specifying the source data of your content along with a name and the content-type.
- Add the attachment to the message with `attach()`.

The attachment will be presented to the recipient as a downloadable file with the filename and content-type you specify.

If you would usually write the file to disk anyway you should just attach it with `Swift_Attachment::fromPath()` since this will use less memory:

Listing 6-17 Listing 6-18

```
1 // Create your file contents in the normal way, but don't write them to disk
2 $data = create_my_pdf_data();
3
4 // Create the attachment with your data
5 $attachment = Swift_Attachment::newInstance($data, 'my-file.pdf', 'application/
```

```
pdf');

// Attach it to the message
$message->attach($attachment);


// You can alternatively use method chaining to build the attachment
$attachment = Swift_Attachment::newInstance()
  ->setFilename('my-file.pdf')
  ->setContentType('application/pdf')
  ->setBody($data)
  ;
```

Changing the Disposition

Attachments just appear as files that can be saved to the Desktop if desired. You can make attachment appear inline where possible by using the `setDisposition()` method of an attachment.

To make an attachment appear inline:

- Call its `setDisposition()` method.

The attachment will be displayed within the email viewing window if the mail client knows how to display it.

If you try to create an inline attachment for a non-displayable file type such as a ZIP file, the mail client should just present the attachment as normal:

Listing 6-19

```
// Create the attachment and call its setDisposition() method
$attachment = Swift_Attachment::fromPath('/path/to/image.jpg')
  ->setDisposition('inline');


// Because there's a fluid interface, you can do this in one statement
$message->attach(
  Swift_Attachment::fromPath('/path/to/image.jpg')->setDisposition('inline')
);
```

Listing 6-20

Embedding Inline Media Files

Often people want to include an image or other content inline with a HTML message. It's easy to do this with HTML linking to remote resources, but this approach is usually blocked by mail clients. Swift Mailer allows you to embed your media directly into the message.

Mail clients usually block downloads from remote resources because this technique was often abused as a mean of tracking who opened an email. If you're sending a HTML email and you want to include an image in the message another approach you can take is to embed the image directly.

Swift Mailer makes embedding files into messages extremely streamlined. You embed a file by calling the `embed()` method of the message, which returns a value you can use in a `src` or `href` attribute in your HTML.

Just like with attachments, it's possible to embed dynamically generated content without having an existing file available.

The embedded files are sent in the email as a special type of attachment that has a unique ID used to reference them within your HTML attributes. On mail clients that do not support embedded files they may appear as attachments.

Although this is commonly done for images, in theory it will work for any displayable (or playable) media type. Support for other media types (such as video) is dependent on the mail client however.

Embedding Existing Files

Files that already exist, either on disk or at a URL can be embedded in a message with just one line of code, using `Swift_EmbeddedFile::fromPath()`.

You can embed files that exist locally, or if your PHP installation has `allow_url_fopen` turned on you can embed files from other websites.

To embed an existing file:

- Create a message object with `Swift_Message::newInstance()`.
- Set the body as HTML, and embed a file at the correct point in the message with `embed()`.

The file will be displayed with the message inline with the HTML wherever its ID is used as a `src` attribute.

`Swift_Image` and `Swift_EmbeddedFile` are just aliases of one another. `Swift_Image` exists for semantic purposes.

You can embed files in two stages if you prefer. Just capture the return value of `embed()` in a variable and use that as the `src` attribute.

Listing 6-21 6-22

```
// Create the message
$message = Swift_Message::newInstance('My subject');

// Set the body
$message->setBody(
'<html>' .
' <head></head>' .
' <body>' .
'  Here is an image <img src="' . // Embed the file
    $message->embed(Swift_Image::fromPath('image.png')) .
   '" alt="Image" />' .
'  Rest of message' .
' </body>' .
'</html>',
  'text/html' // Mark the content-type as HTML
);

// You can embed files from a URL if allow_url_fopen is on in php.ini
$message->setBody(
'<html>' .
' <head></head>' .
' <body>' .
'  Here is an image <img src="' .
    $message->embed(Swift_Image::fromPath('http://site.tld/logo.png')) .
```

```
    '" alt="Image" />' .
'  Rest of message' .
' </body>' .
'</html>',
  'text/html'
);


// If placing the embed() code inline becomes cumbersome
// it's easy to do this in two steps
$cid = $message->embed(Swift_Image::fromPath('image.png'));

$message->setBody(
'<html>' .
' <head></head>' .
' <body>' .
'  Here is an image <img src="' . $cid . '" alt="Image" />' .
'  Rest of message' .
' </body>' .
'</html>',
  'text/html' // Mark the content-type as HTML
);
```

Embedding Dynamic Content

Images that are generated at runtime, such as images created via GD can be embedded directly to a message without writing them out to disk. Use the standard `Swift_Image::newInstance()` method.

To embed dynamically created content:

- Create a message object with `Swift_Message::newInstance()`.
- Set the body as HTML, and embed a file at the correct point in the message with `embed()`. You will need to specify a filename and a content-type.

The file will be displayed with the message inline with the HTML wherever its ID is used as a `src` attribute.

`Swift_Image` and `Swift_EmbeddedFile` are just aliases of one another. `Swift_Image` exists for semantic purposes.

You can embed files in two stages if you prefer. Just capture the return value of `embed()` in a variable and use that as the `src` attribute.

Listing 6-23

```
//Create your file contents in the normal way, but don't write them to disk
$img_data = create_my_image_data();

//Create the message
$message = Swift_Message::newInstance('My subject');

```

Listing 6-24

```php
//Set the body
$message->setBody(
'<html>' .
' <head></head>' .
' <body>' .
'  Here is an image <img src="' . // Embed the file
    $message->embed(Swift_Image::newInstance($img_data, 'image.jpg', 'image/
jpeg')) .
   '" alt="Image" />' .
'  Rest of message' .
' </body>' .
'</html>',
  'text/html' // Mark the content-type as HTML
);


// If placing the embed() code inline becomes cumbersome
// it's easy to do this in two steps
$cid = $message->embed(Swift_Image::newInstance($img_data, 'image.jpg', 'image/
jpeg'));

$message->setBody(
'<html>' .
' <head></head>' .
' <body>' .
'  Here is an image <img src="' . $cid . '" alt="Image" />' .
'  Rest of message' .
' </body>' .
'</html>',
  'text/html' // Mark the content-type as HTML
);
```

Adding Recipients to Your Message

Recipients are specified within the message itself via `setTo()`, `setCc()` and `setBcc()`. Swift Mailer reads these recipients from the message when it gets sent so that it knows where to send the message to.

Message recipients are one of three types:

- `To:` recipients -- the primary recipients (required)
- `Cc:` recipients -- receive a copy of the message (optional)
- `Bcc:` recipients -- hidden from other recipients (optional)

Each type can contain one, or several addresses. It's possible to list only the addresses of the recipients, or you can personalize the address by providing the real name of the recipient.

Syntax for Addresses

If you only wish to refer to a single email address (for example your `From:` address) then you can just use a string.

```
$message->setFrom('some@address.tld');
```

Listing 6-25

Listing 6-26

If you want to include a name then you must use an associative array.

```
$message->setFrom(array('some@address.tld' => 'The Name'));
```

Listing 6-27

Listing 6-28

If you want to include multiple addresses then you must use an array.

```
$message->setTo(array('some@address.tld', 'other@address.tld'));
```

Listing 6-29

Listing 6-30

You can mix personalized (addresses with a name) and non-personalized addresses in the same list by mixing the use of associative and non-associative array syntax.

```
$message->setTo(array(
  'recipient-with-name@example.org' => 'Recipient Name One',
  'no-name@example.org', // Note that this is not a key-value pair
  'named-recipient@example.org' => 'Recipient Name Two'
));
```

Listing 6-31

Listing 6-32

Setting `To:` Recipients

`To:` recipients are required in a message and are set with the `setTo()` or `addTo()` methods of the message.

To set `To:` recipients, create the message object using either `new Swift_Message( ... )` or `Swift_Message::newInstance( ... )`, then call the `setTo()` method with a complete array of addresses, or use the `addTo()` method to iteratively add recipients.

The `setTo()` method accepts input in various formats as described earlier in this chapter. The `addTo()` method takes either one or two parameters. The first being the email address and the second optional parameter being the name of the recipient.

`To:` recipients are visible in the message headers and will be seen by the other recipients.

Multiple calls to `setTo()` will not add new recipients -- each call overrides the previous calls. If you want to iteratively add recipients, use the `addTo()` method.

```
// Using setTo() to set all recipients in one go
$message->setTo(array(
  'person1@example.org',
```

Listing 6-33

Listing 6-34

```
  'person2@otherdomain.org' => 'Person 2 Name',
  'person3@example.org',
  'person4@example.org',
  'person5@example.org' => 'Person 5 Name'
));

// Using addTo() to add recipients iteratively
$message->addTo('person1@example.org');
$message->addTo('person2@example.org', 'Person 2 Name');
```

Setting `Cc:` Recipients

`Cc:` recipients are set with the `setCc()` or `addCc()` methods of the message.

To set `Cc:` recipients, create the message object using either `new Swift_Message( ... )` or `Swift_Message::newInstance( ... )`, then call the `setCc()` method with a complete array of addresses, or use the `addCc()` method to iteratively add recipients.

The `setCc()` method accepts input in various formats as described earlier in this chapter. The `addCc()` method takes either one or two parameters. The first being the email address and the second optional parameter being the name of the recipient.

`Cc:` recipients are visible in the message headers and will be seen by the other recipients.

Multiple calls to `setCc()` will not add new recipients -- each call overrides the previous calls. If you want to iteratively add Cc: recipients, use the `addCc()` method.

Listing 6-35 6-36

```
// Using setCc() to set all recipients in one go
$message->setCc(array(
  'person1@example.org',
  'person2@otherdomain.org' => 'Person 2 Name',
  'person3@example.org',
  'person4@example.org',
  'person5@example.org' => 'Person 5 Name'
));

// Using addCc() to add recipients iteratively
$message->addCc('person1@example.org');
$message->addCc('person2@example.org', 'Person 2 Name');
```

Setting `Bcc:` Recipients

`Bcc:` recipients receive a copy of the message without anybody else knowing it, and are set with the `setBcc()` or `addBcc()` methods of the message.

To set `Bcc:` recipients, create the message object using either `new Swift_Message( ... )` or `Swift_Message::newInstance( ... )`, then call the `setBcc()` method with a complete array of addresses, or use the `addBcc()` method to iteratively add recipients.

The `setBcc()` method accepts input in various formats as described earlier in this chapter. The `addBcc()` method takes either one or two parameters. The first being the email address and the second optional parameter being the name of the recipient.

Only the individual `Bcc:` recipient will see their address in the message headers. Other recipients (including other `Bcc:` recipients) will not see the address.

Multiple calls to `setBcc()` will not add new recipients -- each call overrides the previous calls. If you want to iteratively add Bcc: recipients, use the `addBcc()` method.

Listing 6-37

```
// Using setBcc() to set all recipients in one go
$message->setBcc(array(
  'person1@example.org',
  'person2@otherdomain.org' => 'Person 2 Name',
  'person3@example.org',
  'person4@example.org',
  'person5@example.org' => 'Person 5 Name'
));

// Using addBcc() to add recipients iteratively
$message->addBcc('person1@example.org');
$message->addBcc('person2@example.org', 'Person 2 Name');
```

Listing 6-38

Specifying Sender Details

An email must include information about who sent it. Usually this is managed by the `From:` address, however there are other options.

The sender information is contained in three possible places:

- `From:` -- the address(es) of who wrote the message (required)
- `Sender:` -- the address of the single person who sent the message (optional)
- `Return-Path:` -- the address where bounces should go to (optional)

You must always include a `From:` address by using `setFrom()` on the message. Swift Mailer will use this as the default `Return-Path:` unless otherwise specified.

The `Sender:` address exists because the person who actually sent the email may not be the person who wrote the email. It has a higher precedence than the `From:` address and will be used as the `Return-Path:` unless otherwise specified.

Setting the `From:` Address

A `From:` address is required and is set with the `setFrom()` method of the message. `From:` addresses specify who actually wrote the email, and usually who sent it.

What most people probably don't realise is that you can have more than one `From:` address if more than one person wrote the email -- for example if an email was put together by a committee.

To set the `From:` address(es):

- Call the `setFrom()` method on the Message.

The `From:` address(es) are visible in the message headers and will be seen by the recipients.

If you set multiple `From:` addresses then you absolutely must set a `Sender:` address to indicate who physically sent the message.

Listing 6-39 Listing 6-40

```
// Set a single From: address
$message->setFrom('your@address.tld');

// Set a From: address including a name
$message->setFrom(array('your@address.tld' => 'Your Name'));

// Set multiple From: addresses if multiple people wrote the email
$message->setFrom(array(
  'person1@example.org' => 'Sender One',
  'person2@example.org' => 'Sender Two'
));
```

Setting the `Sender:` Address

A `Sender:` address specifies who sent the message and is set with the `setSender()` method of the message.

To set the `Sender:` address:

- Call the `setSender()` method on the Message.

The `Sender:` address is visible in the message headers and will be seen by the recipients.

This address will be used as the `Return-Path:` unless otherwise specified.

If you set multiple `From:` addresses then you absolutely must set a `Sender:` address to indicate who physically sent the message.

You must not set more than one sender address on a message because it's not possible for more than one person to send a single message.

Listing 6-41 Listing 6-42

```
$message->setSender('your@address.tld');
```

Setting the `Return-Path:` (Bounce) Address

The `Return-Path:` address specifies where bounce notifications should be sent and is set with the `setReturnPath()` method of the message.

You can only have one `Return-Path:` and it must not include a personal name.

To set the `Return-Path:` address:

- Call the `setReturnPath()` method on the Message.

Bouce notifications will be sent to this address.

```
1 $message->setReturnPath('bounces@address.tld');
```
Listing 6-43

Listing 6-44

Requesting a Read Receipt

It is possible to request a read-receipt to be sent to an address when the email is opened. To request a read receipt set the address with `setReadReceiptTo()`.

To request a read receipt:

- Set the address you want the receipt to be sent to with the `setReadReceiptTo()` method on the Message.

When the email is opened, if the mail client supports it a notification will be sent to this address.

Read receipts won't work for the majority of recipients since many mail clients auto-disable them. Those clients that will send a read receipt will make the user aware that one has been requested.

```
1 $message->setReadReceiptTo('your@address.tld');
```
Listing 6-45

Listing 6-46

Setting the Character Set

The character set of the message (and it's MIME parts) is set with the `setCharset()` method. You can also change the global default of UTF-8 by working with the `Swift_Preferences` class.

Swift Mailer will default to the UTF-8 character set unless otherwise overridden. UTF-8 will work in most instances since it includes all of the standard US keyboard characters in addition to most international characters.

It is absolutely vital however that you know what character set your message (or it's MIME parts) are written in otherwise your message may be received completely garbled.

There are two places in Swift Mailer where you can change the character set:

- In the `Swift_Preferences` class
- On each individual message and/or MIME part

To set the character set of your Message:

- Change the global UTF-8 setting by calling `Swift_Preferences::setCharset()`; or
- Call the `setCharset()` method on the message or the MIME part.

```
1 // Approach 1: Change the global setting (suggested)
2 Swift_Preferences::getInstance()->setCharset('iso-8859-2');
3
4 // Approach 2: Call the setCharset() method of the message
5 $message = Swift_Message::newInstance()
6   ->setCharset('iso-8859-2');
7
```
Listing 6-47

Listing 6-48

```
// Apprach 3: Specify the charset when setting the body
$message->setBody('My body', 'text/html', 'iso-8859-2');

// Approach 4: Specify the charset for each part added
$message->addPart('My part', 'text/plain', 'iso-8859-2');
```

Setting the Line Length

The length of lines in a message can be changed by using the `setMaxLineLength()` method on the message. It should be kept to less than 1000 characters.

Swift Mailer defaults to using 78 characters per line in a message. This is done for historical reasons and so that the message can be easily viewed in plain-text terminals.

To change the maximum length of lines in your Message:

- Call the `setMaxLineLength()` method on the Message.

Lines that are longer than the line length specified will be wrapped between words.

You should never set a maximum length longer than 1000 characters according to RFC 2822. Doing so could have unspecified side-effects such as truncating parts of your message when it is transported between SMTP servers.

Listing 6-49 Listing 6-50

```
$message->setMaxLineLength(1000);
```

Setting the Message Priority

You can change the priority of the message with `setPriority()`. Setting the priority will not change the way your email is sent -- it is purely an indicative setting for the recipient.

The priority of a message is an indication to the recipient what significance it has. Swift Mailer allows you to set the priority by calling the `setPriority` method. This method takes an integer value between 1 and 5:

- Highest
- High
- Normal
- Low
- Lowest

To set the message priority:

- Set the priority as an integer between 1 and 5 with the `setPriority()` method on the Message.

Listing 6-51 Listing 6-52

```
// Indicate "High" priority
$message->setPriority(2);
```

Chapter 7
Message Headers

Sometimes you'll want to add your own headers to a message or modify/remove headers that are already present. You work with the message's HeaderSet to do this.

Header Basics

All MIME entities in Swift Mailer -- including the message itself -- store their headers in a single object called a HeaderSet. This HeaderSet is retrieved with the `getHeaders()` method.

As mentioned in the previous chapter, everything that forms a part of a message in Swift Mailer is a MIME entity that is represented by an instance of `Swift_Mime_MimeEntity`. This includes -- most notably -- the message object itself, attachments, MIME parts and embedded images. Each of these MIME entities consists of a body and a set of headers that describe the body.

For all of the "standard" headers in these MIME entities, such as the `Content-Type`, there are named methods for working with them, such as `setContentType()` and `getContentType()`. This is because headers are a moderately complex area of the library. Each header has a slightly different required structure that it must meet in order to comply with the standards that govern email (and that are checked by spam blockers etc).

You fetch the HeaderSet from a MIME entity like so:

Listing 7-1

```
$message = Swift_Message::newInstance();

// Fetch the HeaderSet from a Message object
$headers = $message->getHeaders();

$attachment = Swift_Attachment::fromPath('document.pdf');

// Fetch the HeaderSet from an attachment object
$headers = $attachment->getHeaders();
```

Listing 7-2

The job of the HeaderSet is to contain and manage instances of Header objects. Depending upon the MIME entity the HeaderSet came from, the contents of the HeaderSet will be different, since an attachment for example has a different set of headers to those in a message.

You can find out what the HeaderSet contains with a quick loop, dumping out the names of the headers:

Listing 7-3 Listing 7-4

```
foreach ($headers->getAll() as $header) {
  printf("%s<br />\n", $header->getFieldName());
}

/*
Content-Transfer-Encoding
Content-Type
MIME-Version
Date
Message-ID
From
Subject
To
*/
```

You can also dump out the rendered HeaderSet by calling its `toString()` method:

Listing 7-5 Listing 7-6

```
echo $headers->toString();

/*
Message-ID: <1234869991.499a9ee7f1d5e@swift.generated>
Date: Tue, 17 Feb 2009 22:26:31 +1100
Subject: Awesome subject!
From: sender@example.org
To: recipient@example.org
MIME-Version: 1.0
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: quoted-printable
*/
```

Where the complexity comes in is when you want to modify an existing header. This complexity comes from the fact that each header can be of a slightly different type (such as a Date header, or a header that contains email addresses, or a header that has key-value parameters on it!). Each header in the HeaderSet is an instance of `Swift_Mime_Header`. They all have common functionality, but knowing exactly what type of header you're working with will allow you a little more control.

You can determine the type of header by comparing the return value of its `getFieldType()` method with the constants `TYPE_TEXT`, `TYPE_PARAMETERIZED`, `TYPE_DATE`, `TYPE_MAILBOX`, `TYPE_ID` and `TYPE_PATH` which are defined in `Swift_Mime_Header`.

Listing 7-7 Listing 7-8

```
foreach ($headers->getAll() as $header) {
  switch ($header->getFieldType()) {
    case Swift_Mime_Header::TYPE_TEXT: $type = 'text';
      break;
    case Swift_Mime_Header::TYPE_PARAMETERIZED: $type = 'parameterized';
      break;
    case Swift_Mime_Header::TYPE_MAILBOX: $type = 'mailbox';
```

```
        break;
      case Swift_Mime_Header::TYPE_DATE: $type = 'date';
        break;
      case Swift_Mime_Header::TYPE_ID: $type = 'ID';
        break;
      case Swift_Mime_Header::TYPE_PATH: $type = 'path';
        break;
    }
    printf("%s: is a %s header<br />\n", $header->getFieldName(), $type);
}

/*
Content-Transfer-Encoding: is a text header
Content-Type: is a parameterized header
MIME-Version: is a text header
Date: is a date header
Message-ID: is a ID header
From: is a mailbox header
Subject: is a text header
To: is a mailbox header
*/
```

Headers can be removed from the set, modified within the set, or added to the set.

The following sections show you how to work with the HeaderSet and explain the details of each implementation of `Swift_Mime_Header` that may exist within the HeaderSet.

Header Types

Because all headers are modeled on different data (dates, addresses, text!) there are different types of Header in Swift Mailer. Swift Mailer attempts to categorize all possible MIME headers into more general groups, defined by a small number of classes.

Text Headers

Text headers are the simplest type of Header. They contain textual information with no special information included within it -- for example the Subject header in a message.

There's nothing particularly interesting about a text header, though it is probably the one you'd opt to use if you need to add a custom header to a message. It represents text just like you'd think it does. If the text contains characters that are not permitted in a message header (such as new lines, or non-ascii characters) then the header takes care of encoding the text so that it can be used.

No header -- including text headers -- in Swift Mailer is vulnerable to header-injection attacks. Swift Mailer breaks any attempt at header injection by encoding the dangerous data into a non-dangerous form.

It's easy to add a new text header to a HeaderSet. You do this by calling the HeaderSet's `addTextHeader()` method.

```
$message = Swift_Message::newInstance();

$headers = $message->getHeaders();
```

Listing 7-9

Listing 7-10

```
4
5 $headers->addTextHeader('Your-Header-Name', 'the header value');
```

Changing the value of an existing text header is done by calling it's `setValue()` method.

Listing 7-11 Listing 7-12

```
1 $subject = $message->getHeaders()->get('Subject');
2
3 $subject->setValue('new subject');
```

When output via `toString()`, a text header produces something like the following:

Listing 7-13 Listing 7-14

```
1  $subject = $message->getHeaders()->get('Subject');
2
3  $subject->setValue('amazing subject line');
4
5  echo $subject->toString();
6
7  /*
8
9  Subject: amazing subject line
10
11 */
```

If the header contains any characters that are outside of the US-ASCII range however, they will be encoded. This is nothing to be concerned about since mail clients will decode them back.

Listing 7-15 Listing 7-16

```
1  $subject = $message->getHeaders()->get('Subject');
2
3  $subject->setValue('contains – dash');
4
5  echo $subject->toString();
6
7  /*
8
9  Subject: contains =?utf-8?Q?=E2=80=93?= dash
10
11 */
```

Parameterized Headers

Parameterized headers are text headers that contain key-value parameters following the textual content. The Content-Type header of a message is a parameterized header since it contains charset information after the content type.

The parameterized header type is a special type of text header. It extends the text header by allowing additional information to follow it. All of the methods from text headers are available in addition to the methods described here.

Adding a parameterized header to a HeaderSet is done by using the `addParameterizedHeader()` method which takes a text value like `addTextHeader()` but it also accepts an associative array of key-value parameters.

Listing 7-17

```
$message = Swift_Message::newInstance();

$headers = $message->getHeaders();

$headers->addParameterizedHeader(
  'Header-Name', 'header value',
  array('foo' => 'bar')
  );
```

To change the text value of the header, call it's `setValue()` method just as you do with text headers.

To change the parameters in the header, call the header's `setParameters()` method or the `setParameter()` method (note the pluralization).

Listing 7-18

```
$type = $message->getHeaders()->get('Content-Type');

// setParameters() takes an associative array
$type->setParameters(array(
  'name' => 'file.txt',
  'charset' => 'iso-8859-1'
  ));

// setParameter() takes two args for $key and $value
$type->setParameter('charset', 'iso-8859-1');
```

When output via `toString()`, a parameterized header produces something like the following:

Listing 7-19

```
$type = $message->getHeaders()->get('Content-Type');

$type->setValue('text/html');
$type->setParameter('charset', 'utf-8');

echo $type->toString();

/*

Content-Type: text/html; charset=utf-8

*/
```

If the header contains any characters that are outside of the US-ASCII range however, they will be encoded, just like they are for text headers. This is nothing to be concerned about since mail clients will decode them back. Likewise, if the parameters contain any non-ascii characters they will be encoded so that they can be transmitted safely.

Listing 7-20

Listing 7-21

Listing 7-22

Listing 7-23

Listing 7-24

```php
$attachment = Swift_Attachment::newInstance();

$disp = $attachment->getHeaders()->get('Content-Disposition');

$disp->setValue('attachment');
$disp->setParameter('filename', 'report-may.pdf');

echo $disp->toString();

/*

Content-Disposition: attachment; filename*=utf-8''report%E2%80%93may.pdf

*/
```

Date Headers

Date headers contains an RFC 2822 formatted date (i.e. what PHP's `date('r')` returns). They are used anywhere a date or time is needed to be presented as a message header.

The data on which a date header is modeled is simply a UNIX timestamp such as that returned by `time()` or `strtotime()`. The timestamp is used to create a correctly structured RFC 2822 formatted date such as `Tue, 17 Feb 2009 22:26:31 +1100`.

The obvious place this header type is used is in the `Date:` header of the message itself.

It's easy to add a new date header to a HeaderSet. You do this by calling the HeaderSet's `addDateHeader()` method.

Listing 7-25 Listing 7-26

```php
$message = Swift_Message::newInstance();

$headers = $message->getHeaders();

$headers->addDateHeader('Your-Header-Name', strtotime('3 days ago'));
```

Changing the value of an existing date header is done by calling it's `setTimestamp()` method.

Listing 7-27 Listing 7-28

```php
$date = $message->getHeaders()->get('Date');

$date->setTimestamp(time());
```

When output via `toString()`, a date header produces something like the following:

Listing 7-29 Listing 7-30

```php
$date = $message->getHeaders()->get('Date');

echo $date->toString();

/*

Date: Wed, 18 Feb 2009 13:35:02 +1100
```

```
8
9 */
```

Mailbox (e-mail address) Headers

Mailbox headers contain one or more email addresses, possibly with personalized names attached to them. The data on which they are modeled is represented by an associative array of email addresses and names.

Mailbox headers are probably the most complex header type to understand in Swift Mailer because they accept their input as an array which can take various forms, as described in the previous chapter.

All of the headers that contain e-mail addresses in a message -- with the exception of `Return-Path:` which has a stricter syntax -- use this header type. That is, `To:`, `From:` etc.

You add a new mailbox header to a HeaderSet by calling the HeaderSet's `addMailboxHeader()` method.

```
1 $message = Swift_Message::newInstance();
2
3 $headers = $message->getHeaders();
4
5 $headers->addMailboxHeader('Your-Header-Name', array(
6   'person1@example.org' => 'Person Name One',
7   'person2@example.org',
8   'person3@example.org',
9   'person4@example.org' => 'Another named person'
10  ));
```

Listing 7-32

Changing the value of an existing mailbox header is done by calling it's `setNameAddresses()` method.

```
1 $to = $message->getHeaders()->get('To');
2
3 $to->setNameAddresses(array(
4   'joe@example.org' => 'Joe Bloggs',
5   'john@example.org' => 'John Doe',
6   'no-name@example.org'
7   ));
```

Listing 7-34

If you don't wish to concern yourself with the complicated accepted input formats accepted by `setNameAddresses()` as described in the previous chapter and you only want to set one or more addresses (not names) then you can just use the `setAddresses()` method instead.

```
1 $to = $message->getHeaders()->get('To');
2
3 $to->setAddresses(array(
4   'joe@example.org',
5   'john@example.org',
6   'no-name@example.org'
7   ));
```

Listing 7-36

Both methods will accept the above input format in practice.

If all you want to do is set a single address in the header, you can use a string as the input parameter to `setAddresses()` and/or `setNameAddresses()`.

Listing 7-37 Listing 7-38

```
$to = $message->getHeaders()->get('To');

$to->setAddresses('joe-bloggs@example.org');
```

When output via `toString()`, a mailbox header produces something like the following:

Listing 7-39 Listing 7-40

```
$to = $message->getHeaders()->get('To');

$to->setNameAddresses(array(
  'person1@example.org' => 'Name of Person',
  'person2@example.org',
  'person3@example.org' => 'Another Person'
));

echo $to->toString();

/*

To: Name of Person <person1@example.org>, person2@example.org, Another Person
 <person3@example.org>

*/
```

ID Headers

ID headers contain identifiers for the entity (or the message). The most notable ID header is the Message-ID header on the message itself.

An ID that exists inside an ID header looks more-or-less less like an email address. For example, `<1234955437.499becad62ec2@example.org>`. The part to the left of the @ sign is usually unique, based on the current time and some random factor. The part on the right is usually a domain name.

Any ID passed the an ID header's `setId()` method absolutely MUST conform to this structure, otherwise you'll get an Exception thrown at you by Swift Mailer (a `Swift_RfcComplianceException`). This is to ensure that the generated email complies with relevant RFC documents and therefore is less likely to be blocked as spam.

It's easy to add a new ID header to a HeaderSet. You do this by calling the HeaderSet's `addIdHeader()` method.

Listing 7-41 Listing 7-42

```
$message = Swift_Message::newInstance();

$headers = $message->getHeaders();

$headers->addIdHeader('Your-Header-Name', '123456.unqiue@example.org');
```

Changing the value of an existing date header is done by calling its `setId()` method.

```
$msgId = $message->getHeaders()->get('Message-ID');

$msgId->setId(time() . '.' . uniqid('thing') . '@example.org');
```

Listing 7-44

When output via `toString()`, an ID header produces something like the following:

```
$msgId = $message->getHeaders()->get('Message-ID');

echo $msgId->toString();

/*

Message-ID: <1234955437.499becad62ec2@example.org>

*/
```

Listing 7-46

Path Headers

Path headers are like very-restricted mailbox headers. They contain a single email address with no associated name. The Return-Path header of a message is a path header.

You add a new path header to a HeaderSet by calling the HeaderSet's `addPathHeader()` method.

```
$message = Swift_Message::newInstance();

$headers = $message->getHeaders();

$headers->addPathHeader('Your-Header-Name', 'person@example.org');
```

Listing 7-48

Changing the value of an existing path header is done by calling its `setAddress()` method.

```
$return = $message->getHeaders()->get('Return-Path');

$return->setAddress('my-address@example.org');
```

Listing 7-50

When output via `toString()`, a path header produces something like the following:

```
$return = $message->getHeaders()->get('Return-Path');

$return->setAddress('person@example.org');

echo $return->toString();

/*

Return-Path: <person@example.org>
```

Listing 7-52

```
10
11 */
```

Header Operations

Working with the headers in a message involves knowing how to use the methods on the HeaderSet and on the individual Headers within the HeaderSet.

Adding new Headers

New headers can be added to the HeaderSet by using one of the provided `add..Header()` methods.

To add a header to a MIME entity (such as the message):

Get the HeaderSet from the entity by via its `getHeaders()` method.

- Add the header to the HeaderSet by calling one of the `add..Header()` methods.

The added header will appear in the message when it is sent.

Listing 7-53 7-54

```
1 // Adding a custom header to a message
2 $message = Swift_Message::newInstance();
3 $headers = $message->getHeaders();
4 $headers->addTextHeader('X-Mine', 'something here');
5
6 // Adding a custom header to an attachment
7 $attachment = Swift_Attachment::fromPath('/path/to/doc.pdf');
8 $attachment->getHeaders()->addDateHeader('X-Created-Time', time());
```

Retrieving Headers

Headers are retrieved through the HeaderSet's `get()` and `getAll()` methods.

To get a header, or several headers from a MIME entity:

- Get the HeaderSet from the entity by via its `getHeaders()` method.
- Get the header(s) from the HeaderSet by calling either `get()` or `getAll()`.

When using `get()` a single header is returned that matches the name (case insensitive) that is passed to it. When using `getAll()` with a header name, an array of headers with that name are returned. Calling `getAll()` with no arguments returns an array of all headers present in the entity.

It's valid for some headers to appear more than once in a message (e.g. the Received header). For this reason `getAll()` exists to fetch all headers with a specified name. In addition, `get()` accepts an optional numerical index, starting from zero to specify which header you want more specifically.

If you want to modify the contents of the header and you don't know for sure what type of header it is then you may need to check the type by calling its `getFieldType()` method.

```php
$headers = $message->getHeaders();

// Get the To: header
$toHeader = $headers->get('To');

// Get all headers named "X-Foo"
$fooHeaders = $headers->getAll('X-Foo');

// Get the second header named "X-Foo"
$foo = $headers->get('X-Foo', 1);

// Get all headers that are present
$all = $headers->getAll();
```

Listing 7-55

Listing 7-56

Check if a Header Exists

You can check if a named header is present in a HeaderSet by calling its `has()` method.

To check if a header exists:

- Get the HeaderSet from the entity by via its `getHeaders()` method.
- Call the HeaderSet's `has()` method specifying the header you're looking for.

If the header exists, `true` will be returned or `false` if not.

It's valid for some headers to appear more than once in a message (e.g. the Received header). For this reason `has()` accepts an optional numerical index, starting from zero to specify which header you want to check more specifically.

```php
$headers = $message->getHeaders();

// Check if the To: header exists
if ($headers->has('To')) {
  echo 'To: exists';
}

// Check if an X-Foo header exists twice (i.e. check for the 2nd one)
if ($headers->has('X-Foo', 1)) {
  echo 'Second X-Foo header exists';
}
```

Listing 7-57

Listing 7-58

Removing Headers

Removing a Header from the HeaderSet is done by calling the HeaderSet's `remove()` or `removeAll()` methods.

To remove an existing header:

- Get the HeaderSet from the entity by via its `getHeaders()` method.
- Call the HeaderSet's `remove()` or `removeAll()` methods specifying the header you want to remove.

When calling `remove()` a single header will be removed. When calling `removeAll()` all headers with the given name will be removed. If no headers exist with the given name, no errors will occur.

It's valid for some headers to appear more than once in a message (e.g. the Received header). For this reason `remove()` accepts an optional numerical index, starting from zero to specify which header you want to check more specifically. For the same reason, `removeAll()` exists to remove all headers that have the given name.

Listing 7-59 Listing 7-60

```
$headers = $message->getHeaders();

// Remove the Subject: header
$headers->remove('Subject');

// Remove all X-Foo headers
$headers->removeAll('X-Foo');

// Remove only the second X-Foo header
$headers->remove('X-Foo', 1);
```

Modifying a Header's Content

To change a Header's content you should know what type of header it is and then call it's appropriate setter method. All headers also have a `setFieldBodyModel()` method that accepts a mixed parameter and delegates to the correct setter.

To modify an existing header:

- Get the HeaderSet from the entity by via its `getHeaders()` method.
- Get the Header by using the HeaderSet's `get()`.
- Call the Header's appropriate setter method or call the header's `setFieldBodyModel()` method.

The header will be updated inside the HeaderSet and the changes will be seen when the message is sent.

Listing 7-61 Listing 7-62

```
$headers = $message->getHeaders();

// Change the Subject: header
$subj = $headers->get('Subject');
$subj->setValue('new subject here');

// Change the To: header
$to = $headers->get('To');
$to->setNameAddresses(array(
  'person@example.org' => 'Person',
  'thing@example.org'
));

// Using the setFieldBodyModel() just delegates to the correct method
// So here to calls setNameAddresses()
$to->setFieldBodyModel(array(
  'person@example.org' => 'Person',
  'thing@example.org'
));
```

Chapter 8

Sending Messages

Quick Reference for Sending a Message

Sending a message is very straightforward. You create a Transport, use it to create the Mailer, then you use the Mailer to send the message.

To send a Message:

- Create a Transport from one of the provided Transports -- `Swift_SmtpTransport`, `Swift_SendmailTransport`, `Swift_MailTransport` or one of the aggregate Transports.
- Create an instance of the `Swift_Mailer` class, using the Transport as it's constructor parameter.
- Create a Message.
- Send the message via the `send()` method on the Mailer object.

The `Swift_SmtpTransport` and `Swift_SendmailTransport` transports use `proc_*` PHP functions, which might not be available on your PHP installation. You can easily check if that the case by running the following PHP script: ``<?php echo function_exists('proc_open') ? "Yep, that will work" : "Sorry, that won't work"; ``

When using `send()` the message will be sent just like it would be sent if you used your mail client. An integer is returned which includes the number of successful recipients. If none of the recipients could be sent to then zero will be returned, which equates to a boolean `false`. If you set two `To:` recipients and three `Bcc:` recipients in the message and all of the recipients are delivered to successfully then the value 5 will be returned.

Listing 8-1 *Listing 8-2*

```php
require_once 'lib/swift_required.php';

// Create the Transport
$transport = Swift_SmtpTransport::newInstance('smtp.example.org', 25)
  ->setUsername('your username')
```

```
6     ->setPassword('your password')
7     ;
8
9   /*
10  You could alternatively use a different transport such as Sendmail or Mail:
11
12  // Sendmail
13  $transport = Swift_SendmailTransport::newInstance('/usr/sbin/sendmail -bs');
14
15  // Mail
16  $transport = Swift_MailTransport::newInstance();
17  */
18
19  // Create the Mailer using your created Transport
20  $mailer = Swift_Mailer::newInstance($transport);
21
22  // Create a message
23  $message = Swift_Message::newInstance('Wonderful Subject')
24    ->setFrom(array('john@doe.com' => 'John Doe'))
25    ->setTo(array('receiver@domain.org', 'other@domain.org' => 'A name'))
26    ->setBody('Here is the message itself')
27    ;
28
29  // Send the message
30  $result = $mailer->send($message);
```

Transport Types

A Transport is the component which actually does the sending. You need to provide a Transport object to the Mailer class and there are several possible options.

Typically you will not need to know how a Transport works under-the-surface, you will only need to know how to create an instance of one, and which one to use for your environment.

The SMTP Transport

The SMTP Transport sends messages over the (standardized) Simple Message Transfer Protocol. It can deal with encryption and authentication.

The SMTP Transport, `Swift_SmtpTransport` is without doubt the most commonly used Transport because it will work on 99% of web servers (I just made that number up, but you get the idea). All the server needs is the ability to connect to a remote (or even local) SMTP server on the correct port number (usually 25).

SMTP servers often require users to authenticate with a username and password before any mail can be sent to other domains. This is easily achieved using Swift Mailer with the SMTP Transport.

SMTP is a protocol -- in other words it's a "way" of communicating a job to be done (i.e. sending a message). The SMTP protocol is the fundamental basis on which messages are delivered all over the internet 7 days a week, 365 days a year. For this reason it's the most "direct" method of sending messages you can use and it's the one that will give you the most power and feedback (such as delivery failures) when using Swift Mailer.

Because SMTP is generally run as a remote service (i.e. you connect to it over the network/internet) it's extremely portable from server-to-server. You can easily store the SMTP server address and port number in a configuration file within your application and adjust the settings accordingly if the code is moved or if the SMTP server is changed.

Some SMTP servers -- Google for example -- use encryption for security reasons. Swift Mailer supports using both SSL and TLS encryption settings.

Using the SMTP Transport

The SMTP Transport is easy to use. Most configuration options can be set with the constructor.

To use the SMTP Transport you need to know which SMTP server your code needs to connect to. Ask your web host if you're not sure. Lots of people ask me who to connect to -- I really can't answer that since it's a setting that's extremely specific to your hosting environment.

To use the SMTP Transport:

- Call `Swift_SmtpTransport::newInstance()` with the SMTP server name and optionally with a port number (defaults to 25).
- Use the returned object to create the Mailer.

A connection to the SMTP server will be established upon the first call to `send()`.

Listing 8-3

```
require_once 'lib/swift_required.php';

// Create the Transport
$transport = Swift_SmtpTransport::newInstance('smtp.example.org', 25);

// Create the Mailer using your created Transport
$mailer = Swift_Mailer::newInstance($transport);

/*
It's also possible to use multiple method calls

$transport = Swift_SmtpTransport::newInstance()
  ->setHost('smtp.example.org')
  ->setPort(25)
  ;
*/
```

Encrypted SMTP

You can use SSL or TLS encryption with the SMTP Transport by specifying it as a parameter or with a method call.

To use encryption with the SMTP Transport:

- Pass the encryption setting as a third parameter to `Swift_SmtpTransport::newInstance()`; or
- Call the `setEncryption()` method on the Transport.

A connection to the SMTP server will be established upon the first call to `send()`. The connection will be initiated with the correct encryption settings.

For SSL or TLS encryption to work your PHP installation must have appropriate OpenSSL transports wrappers. You can check if "tls" and/or "ssl" are present in your PHP installation by using the PHP function `stream_get_transports()`

Listing 8-4

Listing 8-5

Listing 8-6

```php
require_once 'lib/swift_required.php';

// Create the Transport
$transport = Swift_SmtpTransport::newInstance('smtp.example.org', 587, 'ssl');

// Create the Mailer using your created Transport
$mailer = Swift_Mailer::newInstance($transport);

/*
It's also possible to use multiple method calls

$transport = Swift_SmtpTransport::newInstance()
  ->setHost('smtp.example.org')
  ->setPort(587)
  ->setEncryption('ssl')
  ;
*/
```

SMTP with a Username and Password

Some servers require authentication. You can provide a username and password with `setUsername()` and `setPassword()` methods.

To use a username and password with the SMTP Transport:

- Create the Transport with `Swift_SmtpTransport::newInstance()`.
- Call the `setUsername()` and `setPassword()` methods on the Transport.

Your username and password will be used to authenticate upon first connect when `send()` are first used on the Mailer.

If authentication fails, an Exception of type `Swift_TransportException` will be thrown.

If you need to know early whether or not authentication has failed and an Exception is going to be thrown, call the `start()` method on the created Transport.

Listing 8-7 Listing 8-8

```php
require_once 'lib/swift_required.php';

// Create the Transport the call setUsername() and setPassword()
$transport = Swift_SmtpTransport::newInstance('smtp.example.org', 25)
  ->setUsername('username')
  ->setPassword('password')
  ;

// Create the Mailer using your created Transport
$mailer = Swift_Mailer::newInstance($transport);
```

The Sendmail Transport

The Sendmail Transport sends messages by communicating with a locally installed MTA -- such as `sendmail`.

The Sendmail Transport, `Swift_SendmailTransport` does not directly connect to any remote services. It is designed for Linux servers that have `sendmail` installed. The Transport starts a local `sendmail` process and sends messages to it. Usually the `sendmail` process will respond quickly as it spools your messages to disk before sending them.

The Transport is named the Sendmail Transport for historical reasons (`sendmail` was the "standard" UNIX tool for sending e-mail for years). It will send messages using other transfer agents such as Exim or Postfix despite its name, provided they have the relevant sendmail wrappers so that they can be started with the correct command-line flags.

It's a common misconception that because the Sendmail Transport returns a result very quickly it must therefore deliver messages to recipients quickly -- this is not true. It's not slow by any means, but it's certainly not faster than SMTP when it comes to getting messages to the intended recipients. This is because sendmail itself sends the messages over SMTP once they have been quickly spooled to disk.

The Sendmail Transport has the potential to be just as smart of the SMTP Transport when it comes to notifying Swift Mailer about which recipients were rejected, but in reality the majority of locally installed `sendmail` instances are not configured well enough to provide any useful feedback. As such Swift Mailer may report successful deliveries where they did in fact fail before they even left your server.

You can run the Sendmail Transport in two different modes specified by command line flags:

- "`-bs`" runs in SMTP mode so theoretically it will act like the SMTP Transport
- "`-t`" runs in piped mode with no feedback, but theoretically faster, though not advised

You can think of the Sendmail Transport as a sort of asynchronous SMTP Transport -- though if you have problems with delivery failures you should try using the SMTP Transport instead. Swift Mailer isn't doing the work here, it's simply passing the work to somebody else (i.e. `sendmail`).

Using the Sendmail Transport

To use the Sendmail Transport you simply need to call `Swift_SendmailTransport::newInstance()` with the command as a parameter.

To use the Sendmail Transport you need to know where `sendmail` or another MTA exists on the server. Swift Mailer uses a default value of `/usr/sbin/sendmail`, which should work on most systems.

You specify the entire command as a parameter (i.e. including the command line flags). Swift Mailer supports operational modes of "`-bs`" (default) and "`-t`".

If you run sendmail in "`-t`" mode you will get no feedback as to whether or not sending has succeeded. Use "`-bs`" unless you have a reason not to.

To use the Sendmail Transport:

- Call `Swift_SendmailTransport::newInstance()` with the command, including the correct command line flags. The default is to use `/usr/sbin/sendmail -bs` if this is not specified.
- Use the returned object to create the Mailer.

A sendmail process will be started upon the first call to `send()`. If the process cannot be started successfully an Exception of type `Swift_TransportException` will be thrown.

Listing 8-9

```
require_once 'lib/swift_required.php';

// Create the Transport
$transport = Swift_SendmailTransport::newInstance('/usr/sbin/exim -bs');

```

Listing 8-10

```
6 // Create the Mailer using your created Transport
7 $mailer = Swift_Mailer::newInstance($transport);
```

The Mail Transport

The Mail Transport sends messages by delegating to PHP's internal `mail()` function.

In my experience -- and others' -- the `mail()` function is not particularly predictable, or helpful.

Quite notably, the `mail()` function behaves entirely differently between Linux and Windows servers. On linux it uses `sendmail`, but on Windows it uses SMTP.

In order for the `mail()` function to even work at all `php.ini` needs to be configured correctly, specifying the location of sendmail or of an SMTP server.

The problem with `mail()` is that it "tries" to simplify things to the point that it actually makes things more complex due to poor interface design. The developers of Swift Mailer have gone to a lot of effort to make the Mail Transport work with a reasonable degree of consistency.

Serious drawbacks when using this Transport are:

- Unpredictable message headers
- Lack of feedback regarding delivery failures
- Lack of support for several plugins that require real-time delivery feedback

It's a last resort, and we say that with a passion!

Using the Mail Transport

To use the Mail Transport you simply need to call `Swift_MailTransport::newInstance()`. It's unlikely you'll need to configure the Transport.

To use the Mail Transport:

- Call `Swift_MailTransport::newInstance()`.
- Use the returned object to create the Mailer.

Messages will be sent using the `mail()` function.

The `mail()` function can take a `$additional_parameters` parameter. Swift Mailer sets this to "`-f%s`" by default, where the "%s" is substituted with the address of the sender (via a `sprintf()`) at send time. You may override this default by passing an argument to `newInstance()`.

Listing 8-11 Listing 8-12

```
1 require_once 'lib/swift_required.php';
2
3 // Create the Transport
4 $transport = Swift_MailTransport::newInstance();
5
6 // Create the Mailer using your created Transport
7 $mailer = Swift_Mailer::newInstance($transport);
```

Available Methods for Sending Messages

The Mailer class offers two methods for sending Messages -- `send()`. Each behaves in a slightly different way.

When a message is sent in Swift Mailer, the Mailer class communicates with whichever Transport class you have chosen to use.

Each recipient in the message should either be accepted or rejected by the Transport. For example, if the domain name on the email address is not reachable the SMTP Transport may reject the address because it cannot process it. Whichever method you use -- `send()` -- Swift Mailer will return an integer indicating the number of accepted recipients.

It's possible to find out which recipients were rejected -- we'll cover that later in this chapter.

Using the `send()` Method

The `send()` method of the `Swift_Mailer` class sends a message using exactly the same logic as your Desktop mail client would use. Just pass it a Message and get a result.

To send a Message with `send()`:

- Create a Transport from one of the provided Transports -- `Swift_SmtpTransport`, `Swift_SendmailTransport`, `Swift_MailTransport` or one of the aggregate Transports.
- Create an instance of the `Swift_Mailer` class, using the Transport as it's constructor parameter.
- Create a Message.
- Send the message via the `send()` method on the Mailer object.

The message will be sent just like it would be sent if you used your mail client. An integer is returned which includes the number of successful recipients. If none of the recipients could be sent to then zero will be returned, which equates to a boolean `false`. If you set two `To:` recipients and three `Bcc:` recipients in the message and all of the recipients are delivered to successfully then the value 5 will be returned.

Listing 8-13

```php
require_once 'lib/swift_required.php';

// Create the Transport
$transport = Swift_SmtpTransport::newInstance('localhost', 25);

// Create the Mailer using your created Transport
$mailer = Swift_Mailer::newInstance($transport);

// Create a message
$message = Swift_Message::newInstance('Wonderful Subject')
  ->setFrom(array('john@doe.com' => 'John Doe'))
  ->setTo(array('receiver@domain.org', 'other@domain.org' => 'A name'))
  ->setBody('Here is the message itself')
  ;

// Send the message
$numSent = $mailer->send($message);

printf("Sent %d messages\n", $numSent);

/* Note that often that only the boolean equivalent of the
   return value is of concern (zero indicates FALSE)

if ($mailer->send($message))
```

Listing 8-14

```
25 {
26   echo "Sent\n";
27 }
28 else
29 {
30   echo "Failed\n";
31 }
32
33 */
```

Sending Emails in Batch

If you want to send a separate message to each recipient so that only their own address shows up in the `To:` field, follow the following recipe:

- Create a Transport from one of the provided Transports -- `Swift_SmtpTransport`, `Swift_SendmailTransport`, `Swift_MailTransport` or one of the aggregate Transports.
- Create an instance of the `Swift_Mailer` class, using the Transport as it's constructor parameter.
- Create a Message.
- Iterate over the recipients and send message via the `send()` method on the Mailer object.

Each recipient of the messages receives a different copy with only their own email address on the `To:` field.

In the following example, two emails are sent. One to each of `receiver@domain.org` and `other@domain.org`. These recipients will not be aware of each other.

Listing 8-15 8-16

```
1  require_once 'lib/swift_required.php';
2
3  // Create the Transport
4  $transport = Swift_SmtpTransport::newInstance('localhost', 25);
5
6  // Create the Mailer using your created Transport
7  $mailer = Swift_Mailer::newInstance($transport);
8
9  // Create a message
10 $message = Swift_Message::newInstance('Wonderful Subject')
11   ->setFrom(array('john@doe.com' => 'John Doe'))
12   ->setBody('Here is the message itself')
13   ;
14
15 // Send the message
16 $failedRecipients = array();
17 $numSent = 0;
18 $to = array('receiver@domain.org', 'other@domain.org' => 'A name');
19
20 foreach ($to as $address => $name)
21 {
22   if (is_int($address)) {
23     $message->setTo($name);
24   } else {
25     $message->setTo(array($address => $name));
26   }
```

```
27
28    $numSent += $mailer->send($message, $failedRecipients);
29  }
30
31  printf("Sent %d messages\n", $numSent);
```

Finding out Rejected Addresses

It's possible to get a list of addresses that were rejected by the Transport by using a by-reference parameter to `send()`.

As Swift Mailer attempts to send the message to each address given to it, if a recipient is rejected it will be added to the array. You can pass an existing array, otherwise one will be created by-reference.

Collecting the list of recipients that were rejected can be useful in circumstances where you need to "prune" a mailing list for example when some addresses cannot be delivered to.

Getting Failures By-reference

Collecting delivery failures by-reference with the `send()` method is as simple as passing a variable name to the method call.

To get failed recipients by-reference:

- Pass a by-reference variable name to the `send()` method of the Mailer class.

If the Transport rejects any of the recipients, the culprit addresses will be added to the array provided by-reference.

If the variable name does not yet exist, it will be initialized as an empty array and then failures will be added to that array. If the variable already exists it will be type-cast to an array and failures will be added to it.

Listing 8-17

```
1  $mailer = Swift_Mailer::newInstance( ... );
2
3  $message = Swift_Message::newInstance( ... )
4    ->setFrom( ... )
5    ->setTo(array(
6      'receiver@bad-domain.org' => 'Receiver Name',
7      'other@domain.org' => 'A name',
8      'other-receiver@bad-domain.org' => 'Other Name'
9    ))
10   ->setBody( ... )
11   ;
12
13 // Pass a variable name to the send() method
14 if (!$mailer->send($message, $failures))
15 {
16   echo "Failures:";
17   print_r($failures);
18 }
19
```

Listing 8-18

```
20 /*
21 Failures:
22 Array (
23   0 => receiver@bad-domain.org,
24   1 => other-receiver@bad-domain.org
25 )
26 */
```

Chapter 9
Plugins

Plugins are provided with Swift Mailer and can be used to extend the behavior of the library in ways that simple class inheritance would be more complex.

AntiFlood Plugin

Many SMTP servers have limits on the number of messages that may be sent during any single SMTP connection. The AntiFlood plugin provides a way to stay within this limit while still managing a large number of emails.

A typical limit for a single connection is 100 emails. If the server you connect to imposes such a limit, it expects you to disconnect after that number of emails has been sent. You could manage this manually within a loop, but the AntiFlood plugin provides the necessary wrapper code so that you don't need to worry about this logic.

Regardless of limits imposed by the server, it's usually a good idea to be conservative with the resources of the SMTP server. Sending will become sluggish if the server is being over-used so using the AntiFlood plugin will not be a bad idea even if no limits exist.

The AntiFlood plugin's logic is basically to disconnect and the immediately re-connect with the SMTP server every X number of emails sent, where X is a number you specify to the plugin.

You can also specify a time period in seconds that Swift Mailer should pause for between the disconnect/re-connect process. It's a good idea to pause for a short time (say 30 seconds every 100 emails) simply to give the SMTP server a chance to process its queue and recover some resources.

Using the AntiFlood Plugin

The AntiFlood Plugin -- like all plugins -- is added with the Mailer class' `registerPlugin()` method. It takes two constructor parameters: the number of emails to pause after, and optionally the number of seconds to pause for.

To use the AntiFlood plugin:

- Create an instance of the Mailer using any Transport you choose.
- Create an instance of the `Swift_Plugins_AntiFloodPlugin` class, passing in one or two constructor parameters.

- Register the plugin using the Mailer's `registerPlugin()` method.
- Continue using Swift Mailer to send messages as normal.

When Swift Mailer sends messages it will count the number of messages that have been sent since the last re-connect. Once the number hits your specified threshold it will disconnect and re-connect, optionally pausing for a specified amount of time.

Listing 9-1 Listing 9-2

```
require_once 'lib/swift_required.php';

// Create the Mailer using any Transport
$mailer = Swift_Mailer::newInstance(
  Swift_SmtpTransport::newInstance('smtp.example.org', 25)
);

// Use AntiFlood to re-connect after 100 emails
$mailer->registerPlugin(new Swift_Plugins_AntiFloodPlugin(100));

// And specify a time in seconds to pause for (30 secs)
$mailer->registerPlugin(new Swift_Plugins_AntiFloodPlugin(100, 30));

// Continue sending as normal
for ($lotsOfRecipients as $recipient) {
  ...

  $mailer->send( ... );
}
```

Throttler Plugin

If your SMTP server has restrictions in place to limit the rate at which you send emails, then your code will need to be aware of this rate-limiting. The Throttler plugin makes Swift Mailer run at a rate-limited speed.

Many shared hosts don't open their SMTP servers as a free-for-all. Usually they have policies in place (probably to discourage spammers) that only allow you to send a fixed number of emails per-hour/day.

The Throttler plugin supports two modes of rate-limiting and with each, you will need to do that math to figure out the values you want. The plugin can limit based on the number of emails per minute, or the number of bytes-transferred per-minute.

Using the Throttler Plugin

The Throttler Plugin -- like all plugins -- is added with the Mailer class' `registerPlugin()` method. It has two required constructor parameters that tell it how to do its rate-limiting.

To use the Throttler plugin:

- Create an instance of the Mailer using any Transport you choose.
- Create an instance of the `Swift_Plugins_ThrottlerPlugin` class, passing the number of emails, or bytes you wish to limit by, along with the mode you're using.
- Register the plugin using the Mailer's `registerPlugin()` method.
- Continue using Swift Mailer to send messages as normal.

When Swift Mailer sends messages it will keep track of the rate at which sending messages is occurring. If it realises that sending is happening too fast, it will cause your program to `sleep()` for enough time to average out the rate.

```
require_once 'lib/swift_required.php';

// Create the Mailer using any Transport
$mailer = Swift_Mailer::newInstance(
  Swift_SmtpTransport::newInstance('smtp.example.org', 25)
);

// Rate limit to 100 emails per-minute
$mailer->registerPlugin(new Swift_Plugins_ThrottlerPlugin(
  100, Swift_Plugins_ThrottlerPlugin::MESSAGES_PER_MINUTE
));

// Rate limit to 10MB per-minute
$mailer->registerPlugin(new Swift_Plugins_ThrottlerPlugin(
  1024 * 1024 * 10, Swift_Plugins_ThrottlerPlugin::BYTES_PER_MINUTE
));

// Continue sending as normal
for ($lotsOfRecipients as $recipient) {
  ...

  $mailer->send( ... );
}
```

Listing 9-3

Listing 9-4

Logger Plugin

The Logger plugins helps with debugging during the process of sending. It can help to identify why an SMTP server is rejecting addresses, or any other hard-to-find problems that may arise.

The Logger plugin comes in two parts. There's the plugin itself, along with one of a number of possible Loggers that you may choose to use. For example, the logger may output messages directly in realtime, or it may capture messages in an array.

One other notable feature is the way in which the Logger plugin changes Exception messages. If Exceptions are being thrown but the error message does not provide conclusive information as to the source of the problem (such as an ambiguous SMTP error) the Logger plugin includes the entire SMTP transcript in the error message so that debugging becomes a simpler task.

There are a few available Loggers included with Swift Mailer, but writing your own implementation is incredibly simple and is achieved by creating a short class that implements the `Swift_Plugins_Logger` interface.

- `Swift_Plugins_Loggers_ArrayLogger`: Keeps a collection of log messages inside an array. The array content can be cleared or dumped out to the screen.
- `Swift_Plugins_Loggers_EchoLogger`: Prints output to the screen in realtime. Handy for very rudimentary debug output.

Using the Logger Plugin

The Logger Plugin -- like all plugins -- is added with the Mailer class' `registerPlugin()` method. It accepts an instance of `Swift_Plugins_Logger` in its constructor.

To use the Logger plugin:

- Create an instance of the Mailer using any Transport you choose.
- Create an instance of the a Logger implementation of `Swift_Plugins_Logger`.

- Create an instance of the `Swift_Plugins_LoggerPlugin` class, passing the created Logger instance to its constructor.
- Register the plugin using the Mailer's `registerPlugin()` method.
- Continue using Swift Mailer to send messages as normal.
- Dump the contents of the log with the logger's `dump()` method.

When Swift Mailer sends messages it will keep a log of all the interactions with the underlying Transport being used. Depending upon the Logger that has been used the behaviour will differ, but all implementations offer a way to get the contents of the log.

Listing 9-5 Listing 9-6

```
require_once 'lib/swift_required.php';

// Create the Mailer using any Transport
$mailer = Swift_Mailer::newInstance(
 Swift_SmtpTransport::newInstance('smtp.example.org', 25)
);

// To use the ArrayLogger
$logger = new Swift_Plugins_Loggers_ArrayLogger();
$mailer->registerPlugin(new Swift_Plugins_LoggerPlugin($logger));

// Or to use the Echo Logger
$logger = new Swift_Plugins_Loggers_EchoLogger();
$mailer->registerPlugin(new Swift_Plugins_LoggerPlugin($logger));

// Continue sending as normal
for ($lotsOfRecipients as $recipient) {
 ...

 $mailer->send( ... );
}

// Dump the log contents
// NOTE: The EchoLogger dumps in realtime so dump() does nothing for it
echo $logger->dump();
```

Decorator Plugin

Often there's a need to send the same message to multiple recipients, but with tiny variations such as the recipient's name being used inside the message body. The Decorator plugin aims to provide a solution for allowing these small differences.

The decorator plugin works by intercepting the sending process of Swift Mailer, reading the email address in the To: field and then looking up a set of replacements for a template.

While the use of this plugin is simple, it is probably the most commonly misunderstood plugin due to the way in which it works. The typical mistake users make is to try registering the plugin multiple times (once for each recipient) -- inside a loop for example. This is incorrect.

The Decorator plugin should be registered just once, but containing the list of all recipients prior to sending. It will use this list of recipients to find the required replacements during sending.

Using the Decorator Plugin

To use the Decorator plugin, simply create an associative array of replacements based on email addresses and then use the mailer's `registerPlugin()` method to add the plugin.

First create an associative array of replacements based on the email addresses you'll be sending the message to.

The replacements array becomes a 2-dimensional array whose keys are the email addresses and whose values are an associative array of replacements for that email address. The curly braces used in this example can be any type of syntax you choose, provided they match the placeholders in your email template.

Listing 9-7

```
$replacements = array();
foreach ($users as $user) {
  $replacements[$user['email']] = array(
    '{username}'=>$user['username'],
    '{password}'=>$user['password']
  );
}
```

Listing 9-8

Now create an instance of the Decorator plugin using this array of replacements and then register it with the Mailer. Do this only once!

Listing 9-9

```
$decorator = new Swift_Plugins_DecoratorPlugin($replacements);

$mailer->registerPlugin($decorator);
```

Listing 9-10

When you create your message, replace elements in the body (and/or the subject line) with your placeholders.

Listing 9-11

```
$message = Swift_Message::newInstance()
  ->setSubject('Important notice for {username}')
  ->setBody(
    "Hello {username}, we have reset your password to {password}\n" .
    "Please log in and change it at your earliest convenience."
  )
  ;

foreach ($users as $user) {
  $message->addTo($user['email']);
}
```

Listing 9-12

When you send this message to each of your recipients listed in your `$replacements` array they will receive a message customized for just themselves. For example, the message used above when received may appear like this to one user:

Listing 9-13

```
Subject: Important notice for smilingsunshine2009

Hello smilingsunshine2009, we have reset your password to rainyDays
Please log in and change it at your earliest convenience.
```

Listing 9-14

While another use may receive the message as:

Listing 9-15 Listing 9-16

```
Subject: Important notice for billy-bo-bob

Hello billy-bo-bob, we have reset your password to dancingOctopus
Please log in and change it at your earliest convenience.
```

While the decorator plugin provides a means to solve this problem, there are various ways you could tackle this problem without the need for a plugin. We're trying to come up with a better way ourselves and while we have several (obvious) ideas we don't quite have the perfect solution to go ahead and implement it. Watch this space.

Providing Your Own Replacements Lookup for the Decorator

Filling an array with replacements may not be the best solution for providing replacement information to the decorator. If you have a more elegant algorithm that performs replacement lookups on-the-fly you may provide your own implementation.

Providing your own replacements lookup implementation for the Decorator is simply a matter of passing an instance of `Swift_Plugins_Decorator_Replacements` to the decorator plugin's constructor, rather than passing in an array.

The Replacements interface is very simple to implement since it has just one method: `getReplacementsFor($address)`.

Imagine you want to look up replacements from a database on-the-fly, you might provide an implementation that does this. You need to create a small class.

Listing 9-17 Listing 9-18

```
class DbReplacements implements Swift_Plugins_Decorator_Replacements {
  public function getReplacementsFor($address) {
    $sql = sprintf(
      "SELECT * FROM user WHERE email = '%s'",
      mysql_real_escape_string($address)
    );

    $result = mysql_query($sql);

    if ($row = mysql_fetch_assoc($result)) {
      return array(
        '{username}'=>$row['username'],
        '{password}'=>$row['password']
      );
    }
  }
}
```

Now all you need to do is pass an instance of your class into the Decorator plugin's constructor instead of passing an array.

Listing 9-19 Listing 9-20

```
$decorator = new Swift_Plugins_DecoratorPlugin(new DbReplacements());

$mailer->registerPlugin($decorator);
```

For each message sent, the plugin will call your class' `getReplacementsFor()` method to find the array of replacements it needs.

If your lookup algorithm is case sensitive, you should transform the `$address` argument as appropriate -- for example by passing it through `strtolower()`.

Chapter 10
Using Swift Mailer for Japanese Emails

To send emails in Japanese, you need to tweak the default configuration.

After requiring the Swift Mailer autoloader (by including the `swift_required.php` file), call the `Swift::init()` method with the following code:

Listing 10-1 Listing 10-2

```
require_once '/path/to/swift-mailer/lib/swift_required.php';

Swift::init(function () {
    Swift_DependencyContainer::getInstance()
        ->register('mime.qpheaderencoder')
        ->asAliasOf('mime.base64headerencoder');

    Swift_Preferences::getInstance()->setCharset('iso-2022-jp');
});

/* rest of code goes here */
```

That's all!