The Swiftmailer Book

generated on May 25, 2019

The Swiftmailer Book

Contents at a Glance

Chapter 1
Introduction

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

System Requirements

Swift Mailer requires PHP 7.0 or higher (`proc_*` functions must be available).

Swift Mailer does not work when used with function overloading as implemented by `mbstring` when `mbstring.func_overload` is set to `2`.

Installation

The recommended way to install Swiftmailer is via Composer:

Listing 1-1
```
$ composer require "swiftmailer/swiftmailer:^6.0"
```

Basic Usage

Here is the simplest way to send emails with Swift Mailer:

Listing 1-2
```
require_once '/path/to/vendor/autoload.php';

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

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

// Create a message
$message = (new Swift_Message('Wonderful Subject'))
```

```
    ->setFrom(['john@doe.com' => 'John Doe'])
    ->setTo(['receiver@domain.org', 'other@domain.org' => 'A name'])
    ->setBody('Here is the message itself')
    ;

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

You can also use Sendmail as a transport:

Listing 1-3
```
// Sendmail
$transport = new Swift_SendmailTransport('/usr/sbin/sendmail -bs');
```

Getting Help

For general support, use *Stack Overflow*[1].

For bug reports and feature requests, create a new ticket in *GitHub*[2].

1. https://stackoverflow.com
2. https://github.com/swiftmailer/swiftmailer/issues

Chapter 2
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

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:

Listing 2-1
```php
// Create the message
$message = (new Swift_Message())

  // Give the message a subject
  ->setSubject('Your subject')

  // Set the From address with an associative array
  ->setFrom(['john@doe.com' => 'John Doe'])

  // Set the To addresses with an associative array (setTo/setCc/setBcc)
  ->setTo(['receiver@domain.org', 'other@domain.org' => 'A name'])

  // Give it a body
  ->setBody('Here is the message itself')

  // And optionally an alternative body
  ->addPart('<q>Here is the message itself</q>', 'text/html')

  // Optionally add any attachments
  ->attach(Swift_Attachment::fromPath('my-document.pdf'))
  ;
```

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 2-2
```
Header-Name: A header value
Other-Header: Another value

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 DateTimeInterface instance 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()`
`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()`

Header	Description	Accessors
`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()`:

Listing 2-3
```
$message = new Swift_Message();
$message->setSubject('My subject');
```

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 2-4
```
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>
To: Receiver Name <recipient@example.org>
MIME-Version: 1.0
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: quoted-printable

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 `new Swift_Message()`:

Listing 2-5
```
// Pass it as a parameter when you create the message
$message = new Swift_Message('My amazing subject');

// Or set it after like this
$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 choose 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.

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 2-6
```php
// Pass it as a parameter when you create the message
$message = new Swift_Message('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.

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.

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

Listing 2-7
```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'));

// You can attach files from a URL if allow_url_fopen is on in php.ini
$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.

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

Listing 2-8
```php
// Create the attachment and call its setFilename() method
$attachment = Swift_Attachment::fromPath('/path/to/image.jpg')
  ->setFilename('cool.jpg');

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

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 `Swift_Attachment` directly.

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

Listing 2-9
```php
// Create your file contents in the normal way, but don't write them to disk
$data = create_my_pdf_data();

// Create the attachment with your data
$attachment = new Swift_Attachment($data, 'my-file.pdf', 'application/pdf');

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


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

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.

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.

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

Listing 2-10
```php
// 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')
);
```

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.

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.

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

Listing 2-11

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

// Set the body
$message->setBody(
'<html>' .
' <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>' .
' <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'
);
```

`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 2-12

```
// 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>' .
' <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 `new Swift_Image()` method.

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

Listing 2-13

```
// 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 = new Swift_Message('My subject');

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

`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 2-14
```php
// If placing the embed() code inline becomes cumbersome
// it's easy to do this in two steps
$cid = $message->embed(new Swift_Image($img_data, 'image.jpg', 'image/jpeg'));

$message->setBody(
'<html>' .
' <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.

Make sure to add only valid email addresses as recipients. If you try to add an invalid email address with `setTo()`, `setCc()` or `setBcc()`, Swift Mailer will throw a `Swift_RfcComplianceException`.

If you add recipients automatically based on a data source that may contain invalid email addresses, you can prevent possible exceptions by validating the addresses using:

Listing 2-15
```php
use Egulias\EmailValidator\EmailValidator;
use Egulias\EmailValidator\Validation\RFCValidation;

$validator = new EmailValidator();
$validator->isValid("example@example.com", new RFCValidation()); //true
```

and only adding addresses that validate. Another way would be to wrap your `setTo()`, `setCc()` and `setBcc()` calls in a try-catch block and handle the `Swift_RfcComplianceException` in the catch block.

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:

Listing 2-16
```
$message->setFrom('some@address.tld');
```

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

Listing 2-17
```
$message->setFrom(['some@address.tld' => 'The Name']);
```

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

Listing 2-18
```
$message->setTo(['some@address.tld', 'other@address.tld']);
```

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:

Listing 2-19
```
$message->setTo([
  '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'
]);
```

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 `new Swift_Message( ... )`, 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:

Listing 2-20
```
// Using setTo() to set all recipients in one go
$message->setTo([
  'person1@example.org',
  'person2@otherdomain.org' => 'Person 2 Name',
  'person3@example.org',
  'person4@example.org',
  'person5@example.org' => 'Person 5 Name'
]);
```

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:

Listing 2-21
```
// 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 `new Swift_Message( ... )`, 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:

Listing 2-22
```php
// Using setTo() to set all recipients in one go
$message->setTo([
  'person1@example.org',
  'person2@otherdomain.org' => 'Person 2 Name',
  'person3@example.org',
  'person4@example.org',
  'person5@example.org' => 'Person 5 Name'
]);
```

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 2-23
```php
// 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 `new Swift_Message( ... )`, 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:

Listing 2-24
```php
// Using setBcc() to set all recipients in one go
$message->setBcc([
  'person1@example.org',
  'person2@otherdomain.org' => 'Person 2 Name',
  'person3@example.org',
  'person4@example.org',
  'person5@example.org' => 'Person 5 Name'
]);
```

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 2-25
```php
// Using addBcc() to add recipients iteratively
$message->addBcc('person1@example.org');
$message->addBcc('person2@example.org', 'Person 2 Name');
```

Internationalized Email Addresses

Traditionally only ASCII characters have been allowed in email addresses. With the introduction of internationalized domain names (IDNs), non-ASCII characters may appear in the domain name. By default, Swiftmailer encodes such domain names in Punycode (e.g. xn--xample-ova.invalid). This is compatible with all mail servers.

RFC 6531 introduced an SMTP extension, SMTPUTF8, that allows non-ASCII characters in email addresses on both sides of the @ sign. To send to such addresses, your outbound SMTP server must support the SMTPUTF8 extension. You should use the `Swift_AddressEncoder_Utf8AddressEncoder` address encoder and enable the `Swift_Transport_Esmtp_SmtpUtf8Handler` SMTP extension handler:

Listing 2-26
```
$smtpUtf8 = new Swift_Transport_Esmtp_SmtpUtf8Handler();
$transport->setExtensionHandlers([$smtpUtf8]);
$utf8Encoder = new Swift_AddressEncoder_Utf8AddressEncoder();
$transport->setAddressEncoder($utf8Encoder);
```

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 realize 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.

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 2-27
```
1  // Set a single From: address
2  $message->setFrom('your@address.tld');
3
4  // Set a From: address including a name
5  $message->setFrom(['your@address.tld' => 'Your Name']);
```

```
6
7  // Set multiple From: addresses if multiple people wrote the email
8  $message->setFrom([
9    'person1@example.org' => 'Sender One',
10   'person2@example.org' => 'Sender Two'
11 ]);
```

Setting the `Sender:` Address

A `Sender:` address specifies who sent the message and is set with the `setSender()` method of 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 2-28
```
$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.

Bounce notifications will be sent to this address:

Listing 2-29
```
$message->setReturnPath('bounces@address.tld');
```

Signed/Encrypted Message

To increase the integrity/security of a message it is possible to sign and/or encrypt an message using one or multiple signers.

S/MIME

S/MIME can sign and/or encrypt a message using the OpenSSL extension.

When signing a message, the signer creates a signature of the entire content of the message (including attachments).

The certificate and private key must be PEM encoded, and can be either created using for example OpenSSL or obtained at an official Certificate Authority (CA).

The recipient must have the CA certificate in the list of trusted issuers in order to verify the signature.

Make sure the certificate supports emailProtection.

When using OpenSSL this can done by the including the *-addtrust emailProtection* parameter when creating the certificate:

Listing 2-30
```php
$message = new Swift_Message();

$smimeSigner = new Swift_Signers_SMimeSigner();
$smimeSigner->setSignCertificate('/path/to/certificate.pem', '/path/to/private-key.pem');
$message->attachSigner($smimeSigner);
```

When the private key is secured using a passphrase use the following instead:

Listing 2-31
```php
$message = new Swift_Message();

$smimeSigner = new Swift_Signers_SMimeSigner();
$smimeSigner->setSignCertificate('/path/to/certificate.pem', ['/path/to/private-key.pem', 'passphrase']);
$message->attachSigner($smimeSigner);
```

By default the signature is added as attachment, making the message still readable for mailing agents not supporting signed messages.

Storing the message as binary is also possible but not recommended:

Listing 2-32
```php
$smimeSigner->setSignCertificate('/path/to/certificate.pem', '/path/to/private-key.pem', PKCS7_BINARY);
```

When encrypting the message (also known as enveloping), the entire message (including attachments) is encrypted using a certificate, and the recipient can then decrypt the message using corresponding private key.

Encrypting ensures nobody can read the contents of the message without the private key.

Normally the recipient provides a certificate for encrypting and keeping the decryption key private.

Using both signing and encrypting is also possible:

Listing 2-33
```php
$message = new Swift_Message();

$smimeSigner = new Swift_Signers_SMimeSigner();
$smimeSigner->setSignCertificate('/path/to/sign-certificate.pem', '/path/to/private-key.pem');
$smimeSigner->setEncryptCertificate('/path/to/encrypt-certificate.pem');
$message->attachSigner($smimeSigner);
```

The used encryption cipher can be set as the second parameter of setEncryptCertificate()

See *https://secure.php.net/manual/openssl.ciphers* for a list of supported ciphers.

By default the message is first signed and then encrypted, this can be changed by adding:

Listing 2-34
```php
$smimeSigner->setSignThenEncrypt(false);
```

Changing this is not recommended as most mail agents don't support this none-standard way.

Only when having trouble with sign then encrypt method, this should be changed.

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()`:

```
$message->setReadReceiptTo('your@address.tld[1]');
```

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

1. #.-mailto:'your&-37;&-52;&-48;address&-46;tld

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.

Setting the Character Set

The character set of the message (and its 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:

Listing 2-35

```
// Approach 1: Change the global setting (suggested)
Swift_Preferences::getInstance()->setCharset('iso-8859-2');

// Approach 2: Call the setCharset() method of the message
$message = (new Swift_Message())
  ->setCharset('iso-8859-2');

// Approach 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 Encoding

The body of each MIME part needs to be encoded. Binary attachments are encoded in base64 using the `Swift_Mime_ContentEncoder_Base64ContentEncoder`. Text parts are traditionally encoded in quoted-printable using `Swift_Mime_ContentEncoder_QpContentEncoder` or `Swift_Mime_ContentEncoder_NativeQpContentEncoder`.

The encoder of the message or MIME part is set with the `setEncoder()` method.

Quoted-printable is the safe choice, because it converts 8-bit text as 7-bit. Most modern SMTP servers support 8-bit text. This is advertised via the 8BITMIME SMTP extension. If your outbound SMTP server supports this SMTP extension, and it supports downgrading the message (e.g converting to quoted-printable on the fly) when delivering to a downstream server that does not support the extension, you may wish to use `Swift_Mime_ContentEncoder_PlainContentEncoder` in `8bit` mode instead. This has the advantage that the source data is slightly more readable and compact, especially for non-Western languages.

```
$eightBitMime = new Swift_Transport_Esmtp_EightBitMimeHandler(); $transport->setExtensionHandlers([$eightBitMime]); $plainEncoder = new Swift_Mime_ContentEncoder_PlainContentEncoder('8bit'); $message->setEncoder($plainEncoder);
```

Setting the Line Length

The length of lines in a message can be changed by using the `setMaxLineLength()` method on the message:

Listing 2-36
```
$message->setMaxLineLength(1000);
```

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

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.

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:

Listing 2-37
```
// Indicate "High" priority
$message->setPriority(2);
```

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:

- `Swift_Mime_SimpleMessage::PRIORITY_HIGHEST`: 1
- `Swift_Mime_SimpleMessage::PRIORITY_HIGH`: 2
- `Swift_Mime_SimpleMessage::PRIORITY_NORMAL`: 3
- `Swift_Mime_SimpleMessage::PRIORITY_LOW`: 4
- `Swift_Mime_SimpleMessage::PRIORITY_LOWEST`: 5

Listing 2-38
```
// Or use the constant to be more explicit
$message->setPriority(Swift_Mime_SimpleMessage::PRIORITY_HIGH);
```

Chapter 3
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_SimpleMimeEntity`. 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 3-1
```php
$message = new Swift_Message();

// 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();
```

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 3-2
```php
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 3-3
```php
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 3-4
```php
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:

Listing 3-5
```php
$message = new Swift_Message();
$headers = $message->getHeaders();
$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 3-6
```php
$subject = $message->getHeaders()->get('Subject');
$subject->setValue('new subject');
```

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

Listing 3-7
```php
$subject = $message->getHeaders()->get('Subject');
$subject->setValue('amazing subject line');
echo $subject->toString();

/*

Subject: amazing subject line

*/
```

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 3-8
```php
$subject = $message->getHeaders()->get('Subject');
$subject->setValue('contains - dash');
echo $subject->toString();

/*
```

```
6
7  Subject: contains =?utf-8?Q?=E2=80=93?= dash
8
9  */
```

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 3-9
```php
$message = new Swift_Message();
$headers = $message->getHeaders();
$headers->addParameterizedHeader(
  'Header-Name', 'header value',
  ['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 3-10
```php
$type = $message->getHeaders()->get('Content-Type');

// setParameters() takes an associative array
$type->setParameters([
  '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 3-11
```php
$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 3-12

```php
$attachment = new Swift_Attachment();
$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 as a DateTimeImmutable object. The object is used to create a correctly structured RFC 2822 formatted date with timezone 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 3-13
```php
$message = new Swift_Message();
$headers = $message->getHeaders();
$headers->addDateHeader('Your-Header', new DateTimeImmutable('3 days ago'));
```

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

Listing 3-14
```php
$date = $message->getHeaders()->get('Date');
$date->setDateTime(new DateTimeImmutable());
```

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

Listing 3-15
```php
$date = $message->getHeaders()->get('Date');
echo $date->toString();

/*

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

*/
```

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:

Listing 3-16

```php
$message = new Swift_Message();
$headers = $message->getHeaders();
$headers->addMailboxHeader('Your-Header-Name', [
      'person1@example.org' => 'Person Name One',
      'person2@example.org',
      'person3@example.org',
      'person4@example.org' => 'Another named person'
]);
```

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

Listing 3-17
```php
$to = $message->getHeaders()->get('To');
$to->setNameAddresses([
  'joe@example.org' => 'Joe Bloggs',
  'john@example.org' => 'John Doe',
  'no-name@example.org'
]);
```

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:

Listing 3-18
```php
$to = $message->getHeaders()->get('To');
$to->setAddresses([
  'joe@example.org',
  'john@example.org',
  'no-name@example.org'
]);
```

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 3-19
```php
$to = $message->getHeaders()->get('To');
$to->setAddresses('joe-bloggs@example.org');
```

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

Listing 3-20
```php
$to = $message->getHeaders()->get('To');
$to->setNameAddresses([
  '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>

*/
```

Internationalized domains are automatically converted to IDN encoding:

Listing 3-21
```php
$to = $message->getHeaders()->get('To');
$to->setAddresses('joe@ëxämple.org');

```

```
4  echo $to->toString();
5
6  /*
7
8  To: joe@xn--xmple-gra1c.org
9
10 */
```

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 to the 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 3-22
```
$message = new Swift_Message();
$headers = $message->getHeaders();
$headers->addIdHeader('Your-Header-Name', '123456.unqiue@example.org');
```

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

Listing 3-23
```
$msgId = $message->getHeaders()->get('Message-ID');
$msgId->setId(time() . '.' . uniqid('thing') . '@example.org');
```

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

Listing 3-24
```
1  $msgId = $message->getHeaders()->get('Message-ID');
2  echo $msgId->toString();
3
4  /*
5
6  Message-ID: <1234955437.499becad62ec2@example.org>
7
8  */
```

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:

Listing 3-25
```
$message = new Swift_Message();
$headers = $message->getHeaders();
$headers->addPathHeader('Your-Header-Name', 'person@example.org');
```

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

Listing 3-26
```
$return = $message->getHeaders()->get('Return-Path');
$return->setAddress('my-address@example.org');
```

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

Listing 3-27
```php
$return = $message->getHeaders()->get('Return-Path');
$return->setAddress('person@example.org');
echo $return->toString();

/*

Return-Path: <person@example.org>

*/
```

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

Listing 3-28
```php
// Adding a custom header to a message
$message = new Swift_Message();
$headers = $message->getHeaders();
$headers->addTextHeader('X-Mine', 'something here');

// Adding a custom header to an attachment
$attachment = Swift_Attachment::fromPath('/path/to/doc.pdf');
$attachment->getHeaders()->addDateHeader('X-Created-Time', time());
```

Retrieving Headers

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

Listing 3-29
```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();
```

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.

Check if a Header Exists

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

Listing 3-30

```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';
}
```

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.

Removing Headers

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

Listing 3-31

```php
$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);
```

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.

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:

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

Listing 3-32

```php
$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([
  'person@example.org' => 'Person',
  'thing@example.org'
]);

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

Chapter 4
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.

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 4-1

```
// Create the Transport
$transport = (new Swift_SmtpTransport('smtp.example.org', 25))
  ->setUsername('your username')
  ->setPassword('your password')
  ;

/*
You could alternatively use a different transport such as Sendmail:

// Sendmail
$transport = new Swift_SendmailTransport('/usr/sbin/sendmail -bs');
*/

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

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

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

Transport Types

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:

Listing 4-2
```
* ``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_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).

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.

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

Listing 4-3
```
1  // Create the Transport
2  $transport = new Swift_SmtpTransport('smtp.example.org', 25);
3
4  // Create the Mailer using your created Transport
5  $mailer = new Swift_Mailer($transport);
```

```
6
7  /*
8  It's also possible to use multiple method calls
9
10 $transport = (new Swift_SmtpTransport())
11   ->setHost('smtp.example.org')
12   ->setPort(25)
13   ;
14 */
```

Encrypted SMTP

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

Listing 4-4

```
1  // Create the Transport
2  $transport = new Swift_SmtpTransport('smtp.example.org', 587, 'ssl');
3
4  // Create the Mailer using your created Transport
5  $mailer = new Swift_Mailer($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()`.

If you are using *Mailcatcher*[1], make sure you do not set the encryption for the `Swift_SmtpTransport`, since Mailcatcher does not support encryption.

SMTP with a Username and Password

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

Listing 4-5

```
1  // Create the Transport the call setUsername() and setPassword()
2  $transport = (new Swift_SmtpTransport('smtp.example.org', 25))
3    ->setUsername('username')
4    ->setPassword('password')
5    ;
6
7  // Create the Mailer using your created Transport
8  $mailer = new Swift_Mailer($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.

1. https://mailcatcher.me/

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 `new Swift_SendmailTransport()` 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.

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 4-6
```php
// Create the Transport
$transport = new Swift_SendmailTransport('/usr/sbin/exim -bs');

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

Available Methods for Sending Messages

The Mailer class offers one method for sending Messages -- `send()`.

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. `send()` 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.

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 4-7

```php
// Create the Transport
$transport = new Swift_SmtpTransport('localhost', 25);

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

// Create a message
$message = (new Swift_Message('Wonderful Subject'))
  ->setFrom(['john@doe.com' => 'John Doe'])
  ->setTo(['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))
{
  echo "Sent\n";
}
else
{
  echo "Failed\n";
}

*/
```

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`, 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.

Make sure to add only valid email addresses as recipients. If you try to add an invalid email address with `setTo()`, `setCc()` or `setBcc()`, Swift Mailer will throw a `Swift_RfcComplianceException`.

If you add recipients automatically based on a data source that may contain invalid email addresses, you can prevent possible exceptions by validating the addresses using `Egulias\EmailValidator\EmailValidator` (a dependency that is installed with Swift Mailer) and only adding addresses that validate. Another way would be to wrap your `setTo()`, `setCc()` and `setBcc()` calls in a try-catch block and handle the `Swift_RfcComplianceException` in the catch block.

Handling invalid addresses properly is especially important when sending emails in large batches since a single invalid address might cause an unhandled exception and stop the execution or your script early.

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 4-8

```php
// Create the Transport
$transport = new Swift_SmtpTransport('localhost', 25);

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

// Create a message
$message = (new Swift_Message('Wonderful Subject'))
  ->setFrom(['john@doe.com' => 'John Doe'])
  ->setBody('Here is the message itself')
  ;

// Send the message
$failedRecipients = [];
$numSent = 0;
$to = ['receiver@domain.org', 'other@domain.org' => 'A name'];

foreach ($to as $address => $name)
{
  if (is_int($address)) {
    $message->setTo($name);
  } else {
    $message->setTo([$address => $name]);
  }

  $numSent += $mailer->send($message, $failedRecipients);
}

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:

Listing 4-9

```
$mailer = new Swift_Mailer( ... );

$message = (new Swift_Message( ... ))
  ->setFrom( ... )
  ->setTo([
    'receiver@bad-domain.org' => 'Receiver Name',
    'other@domain.org' => 'A name',
    'other-receiver@bad-domain.org' => 'Other Name'
  ))
  ->setBody( ... )
  ;

// Pass a variable name to the send() method
if (!$mailer->send($message, $failures))
{
  echo "Failures:";
  print_r($failures);
}

/*
Failures:
Array (
  0 => receiver@bad-domain.org,
  1 => other-receiver@bad-domain.org
)
*/
```

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.

Chapter 5
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.

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's `registerPlugin()` method. It takes two constructor parameters: the number of emails to pause after, and optionally the number of seconds to pause for.

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 5-1

```
// Create the Mailer using any Transport
$mailer = new Swift_Mailer(
  new Swift_SmtpTransport('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.

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:

Listing 5-2

```
// Create the Mailer using any Transport
$mailer = new Swift_Mailer(
  new Swift_SmtpTransport('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( ... );
}
```

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.

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 5-3

```
// Create the Mailer using any Transport
$mailer = new Swift_Mailer(
 new Swift_SmtpTransport('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( ... );
}
```

```
20
21  // Dump the log contents
22  // NOTE: The Echologger dumps in realtime so dump() does nothing for it
23  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 5-4

```
1  $replacements = [];
2  foreach ($users as $user) {
3    $replacements[$user['email']] = [
4      '{username}'=>$user['username'],
5      '{resetcode}'=>$user['resetcode']
6    ];
7  }
```

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 5-5

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

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

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

Listing 5-6

```
1  $message = (new Swift_Message())
2    ->setSubject('Important notice for {username}')
3    ->setBody(
4      "Hello {username}, you requested to reset your password.\n" .
5      "Please visit https://example.com/pwreset and use the reset code {resetcode} to set a new password."
6    )
7    ;
```

```
8
9  foreach ($users as $user) {
10   $message->addTo($user['email']);
11 }
```

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 5-7

```
1 Subject: Important notice for smilingsunshine2009
2
3 Hello smilingsunshine2009, you requested to reset your password.
4 Please visit https://example.com/pwreset and use the reset code 183457 to set a new password.
```

While another use may receive the message as:

Listing 5-8

```
1 Subject: Important notice for billy-bo-bob
2
3 Hello billy-bo-bob, you requested to reset your password.
4 Please visit https://example.com/pwreset and use the reset code 539127 to set a new password.
```

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 5-9

```
1  class DbReplacements implements Swift_Plugins_Decorator_Replacements {
2    public function getReplacementsFor($address) {
3      global $db; // Your PDO instance with a connection to your database
4      $query = $db->prepare(
5        "SELECT * FROM `users` WHERE `email` = ?"
6      );
7
8      $query->execute([$address]);
9
10     if ($row = $query->fetch(PDO::FETCH_ASSOC)) {
11       return [
12         '{username}'=>$row['username'],
13         '{resetcode}'=>$row['resetcode']
14       ];
15     }
16   }
17 }
```

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

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

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

Listing 5-10

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 6
Using Swift Mailer for Japanese Emails

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

Call the `Swift::init()` method with the following code as early as possible in your code:

Listing 6-1
```
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!