# 29

# Building a Web-Based Email Client with Laravel Part II

Now that you have been made familiar with the fundamental concepts and components of the Laravel framework in the previous chapter, we are ready to discuss their implementation in a real code base. For this project we are going to combine the knowledge we've obtained on Laravel with the IMAP functionality provided in PHP to create a simple web-based email client.

Key topics covered include:

- The fundamental functions for interacting with an IMAP server in PHP, including the design of an object-oriented interface for them
- The design and implementation of a basic Laravel 5 application to read, compose, and reply to emails from a web browser.

## Building a Simple IMAP Client using Laravel

For the purposes of simplicity we will build our email client out of the box to work with Google's Gmail email service (effectively reproducing a subset of the Gmail web interface).

### The PHP IMAP Functions

IMAP, or the Internet Message Access Protocol, is a common standard in use within the Internet today for accessing email messages stored on a server. Compared to other protocols such as POP3 (which typically only temporarily stores emails on behalf of a user

until they are downloaded), IMAP is designed to retain and manage email on the server itself, providing access to designated users.

In PHP access to IMAP servers is provided by the PHP IMAP extension, an extremely robust and rather low-level set of tools that let you manipulate and work with an email server by the IMAP protocol. Not all of this functionality is needed to build a simple email client as we are in this project, so we will focus on that functionality we specifically use.

Like many of the older extensions available in PHP, IMAP functions primarily through the use of resources. In its most basic form, a connection is opened to the IMAP server using the `imap_open()` function and a PHP resource is returned representing that connection. This resource is then used in all further interactions with the server, generally as the first parameter to that function.

### Opening a Connection to an IMAP server

In order to open a connection to an IMAP server we will start with the `imap_open()` function declaration:

```
resource imap_open(string $mailbox_spec, string $username, string $password [, int
$options = 0 [, int $n_retries = 0 [, array $params = null ]]]);
```

In this declaration, `$mailbox_spec` is a specially-constructed string that specifies the IMAP server (and specific mailbox on that server) you wish to connect to. We'll discuss this in more detail below. The `$username` and `$password` parameters are self-explanatory as the authentication credentials for that mailbox. The optional `$options` parameter is a bit-mask of various options to apply to this connection and is a combination of the following constant values:

- `OP_READONLY` – Open the mailbox for read operations only.
- `OP_HALFOPEN` – Open just a connection to the server, but do not access a specific mailbox.
- `CL_EXPUNGE` – Expunge the referenced mailbox upon close of the connection.
- `OP_DEBUG` – Enable debugging of protocol negotiations.
- `OP_SHORTCACHE` – Limit caching.
- `OP_SECURE` – Only connect to the IMAP server using a secure version of authentication.

Note referenced in the PHP documentation there are other constants which can be passed as part of this bitmask, however these have been omitted because they are either typically not useful to the practical developer or exist for the use of the extension maintainers.

The second optional parameter, `$n_retries`, is an integer describing how many connection attempts should be tried before the function ends in error, and `$params` is an array of key/value pairs to set additional options. At the time of writing a single configuration

option `DISABLE_AUTHENTICATOR` is available for this parameter that disables authentication properties when set (please refer to the PHP IMAP documentation for more information).

Let's get back to the most important of parameters, the `$mailbox_spec` parameter, and describe how it works. The concept behind this parameter is similar to the DSN concept previously explained in discussions of the PDO extension except instead of it specifying the connection to a database it specifies the connection to an IMAP server and corresponding mailbox on that server. Fundamentally the basic syntax of this connection string is as shown:

```
"{" server [":"port][flags] "}" [mailbox_name]
```

Or in a more practical example:

```
{imap.gmail.com:993/imap/ssl}/INBOX
```

All flags are specified in the format of a path within the server definition and can be combined as such. For example, one available flag is the `/novalidate-cert` flag, which instructs the connection not to validate the security certificates in a TLS/SSL connection (which would be needed if the certificate of the server was self-signed). The above example of connecting using this flag would be as follows:

```
{imap.google.com:993/imap/ssl/novalidate-cert}/INBOX
```

A complete reference of available IMAP connection flags and their meaning can be found in the PHP documentation of the `imap_open()` function.

### IMAP and Mailboxes

In the IMAP protocol email is organized into a series of folders, or in IMAP terms mailboxes. This provides the obvious benefit to the end user of better organization of their email, and in our simple email client we will be providing the means to switch between these various mailboxes to see their contents. The first step in this process is retrieving a list of all of the mailboxes accessible by the user, which is handled in PHP by using the `imap_list()` function:

```
array imap_list(resource $resource, string $server_ref, string $search_pattern);
```

The first parameter of `imap_list()` is as expected the server resource returned from a call to the `imap_open()` function. The second parameter is the server reference, which is typically identical to the `$mailbox_spec` of the `imap_open()` function (without specifying the mailbox portion). The final parameter is the `$pattern` parameter, which allows you specify the specific portion of the mailbox hierarchy from which to retrieve your list.

The pattern can be a specific mailbox path and provides for two special cases of search. The first, specifying simply `'*'` as the pattern will return all mailboxes in the hierarchy. The second, `'%'` specifies return only the current mailboxes in the specified path. Let's demonstrate this function in action:

```
<?php
```

```
    // Return all mailboxes
    $mailboxes = imap_list($resource, '{imap.google.com:993/imap/ssl}', '*');

    // Return only the mailboxes that exist under 'Archive' (not including their
children)
    $archives = imap_list($resource, '{imap.google.com:993/imap/ssl}',
'Archive/%');

?>
```

Before we can retrieve a list of messages, we must first be in the appropriate mailbox of the account to retrieve the messages of. Although odd, there is no intuitive or obvious function for switching mailboxes available in the PHP extension. Rather, in order to change mailboxes you will need to employ the `imap_reopen()` function to "re-open" the connection under the context of a new mailbox:

```
    bool imap_reopen(resource $resource, string $new_mailbox_ref [, int $options [,
int $n_retries]])
```

As before the `$resource` parameter is the resource returned from the original call to the `imap_open()` function previously introduced. The second parameter, `$new_mailbox_ref` is a string of the identical format of the `$mailbox_ref` of the `imap_open()` function, except referring to the mailbox you would like to change into. The final two parameters `$options` and `$n_retries` have the same purpose as was previously introduced for the `imap_open()` function.

Thus, if you would like to change from the `'INBOX'` mailbox to the `'Archive'` mailbox on an IMAP server, the following is an example of how this could be done:

```
<?php

$connection = imap_open('{imap.gmail.com:993/imap/ssl}/INBOX', $username,
$password);

if(imap_reopen($connection '{imap.gmail.com:993/imap/ssl}/Archive')) {
     echo "Mailbox changed to 'Archive'";
} else {
    echo "Failed to switch mailbox.";
}

?>
```

There are other various functions available to you as the developer when working with mailboxes (such as those for creating and deleting mailboxes), but these tools are outside of the scope of our simple web client and this chapter. If you would like to use them, we encourage you to refer to the PHP documentation which details their use.

### Retrieving Message lists from IMAP

Thus far, we have demonstrated how to make a connection to an IMAP server, list the available mailboxes, and select which mailbox you would like to work with at any given moment. Next we will look at how you retrieve the email(s) contained within that mailbox if any are present.

Logically you likely will want to retrieve a list of emails within an inbox prior to downloading a specific email message. Using the PHP extension, this is fundamentally done using the `imap_fetch_overview()` function, which will return a range of emails in a given inbox that you specify. It is important to note that this function returns only a range and not all of the emails available. Especially in today's modern era of cheap storage, often the contents of a given IMAP mailbox can range in the tens of thousands of emails and thus it is impractical and largely unreasonable to want to retrieve all of them in a single action. Instead, it is much more reasonable to only retrieve a portion of the emails available at any given time and implement some mechanism of pagination to move forward and backward in the list. This will be covered momentarily in this chapter, but for now let's just look at the `imap_fetch_overview()` function itself and how it is used:

```
array imap_fetch_overview(resource $resource, string $sequence [, int $options =
0])
```

The `$resource` parameter is at this point obviously the IMAP resource in question, while the $sequence parameter is a string representing a description of which messages you would like to retrieve from the mailbox. This sequence can take multiple forms, and when we say sequence we are referring to the notion that all of the emails in a given mailbox are given a sequential identifier. Thus, the first email has a sequence value of `1`, the second `2`, etc. When specifying the sequence, you can either enumerate specific sequence values you would like to retrieve as a comma-separated list or you can specify a range using the format `"X:Y"` where `X` is the start of the sequence and `Y` is the end of the range you wish to retrieve:

```
<?php


// Retrieve the first 10 messages in the inbox using the sequence
$messages = imap_fetch_overview($connection, "1:10");


?>
```

You will note the last optional parameter of the `imap_fetch_overview()` function has yet to be described, and that is because it introduces a new concept that requires introduction, which we will do now.

As previously stated, the `$sequence` parameter refers to an integer value representing the specific position of a given email within the entire mailbox. Thus, using a sequence range of `1:10` or `3,5,7` will return the first ten emails or the third, fifth, and seventh emails as an array, respectively. However, there is a second way to reference an email within a mailbox that is using a unique identifier given to that email by the IMAP server. This unique identifier allows for fast retrieval of a specific email without needing to first locate its position in the mailbox (often because you have already previously found it). The unique identifier for an email is returned as part of the array structure representing the email under the `'uid'` array key. When retrieving emails by unique identifier as list using the

`imap_fetch_overview()` function you can set the optional `$options` parameter to the constant `FT_UID` and then specify one or more comma-separated unique identifiers for the `$sequence` parameter. When referencing emails by unique identifier however, the ability to specify a sequence range in the `'X:Y'` format previously introduced is unavailable. The follow code demonstrates this in principle:

```
<?php

// Retrieve the first ten emails in the current inbox
$messages = imap_fetch_overview($connection, '1:10');

// Extract the unique IDs of the third, fifth, and seventh emails (array is zero-indexed)
$unique_ids = [
    $messages[2]['uid'],
    $messages[4]['uid'],
    $messages[6]['uid']
];

// Retrieve only the third, fifth, and seventh emails by unique ID
$subsetMessages = imap_fetch_overview($connection, implode(',', $unique_ids), FT_UID);

?>
```

As the name of the function implies, the `imap_fetch_overview()` function only returns an overview of the email with only those details about the message typically needed when displaying a list of emails in an email client. For each message returned in the resulting array the following key/value pairs will be defined if available for that email message. As not all IMAP servers or even individual emails within a single IMAP server store the same details some of these keys may not be available and they should always be confirmed before attempting to reference them:

- `$email['subject']` – The subject of the email
- `$email['from']` – The sender of the email's address
- `$email['to']` – The recipient of the email's address (in RFC822 format)
- `$email['date']` – The date the email was sent (in RFC822 format)
- `$email['message_id']` – The message ID of the email (not to be confused with sequence or unique ID
- `$email['references']` – A optional message ID indicating this email is in reference to the provided message ID

- `$email['in_reply_to']` – A optional message ID indicating this email is a reply to a previous email indicated by the provided message ID
- `$email['size']` – the Size of the email in bytes
- `$email['uid']` – The Unique ID of the email as specified by the IMAP server
- `$email['msgno']` – The sequence ID of the email within the mailbox
- `$email['recent']` – A flag indicating the message is recent
- `$email['flagged']` – A flag indicating the message is flagged as spam, junk, etc.
- `$email['answered']` – A flag indicating the message has been previously responded to
- `$email['deleted']` – A flag indicating the message is flagged to be deleted
- `$email['seen']` – A flag indicating the message as previously been opened
- `$email['draft']` – A flag indicating the message is marked as a draft

With our introduction to the `imap_fetch_overview()` function complete let us now return to the problem of pagination. Seeing that it is impractical to attempt to simply download every single message in a given mailbox every time, it is necessary for us to split this into multiple steps download a page of emails at a time. To do this effectively in a way that allows us to know how many pages of content we have, we first need to understand how many emails are in a given mailbox. This sort of metadata about a given IMAP mailbox is the purview of the `imap_check()` function we will introduce now.

The purpose of the `imap_check()` function is to retrieve various useful pieces of information about the currently active mailbox. It has a simple prototype:

```
object imap_check(resource $connection);
```

The object returned by the `imap_check()` function is an instance of the generic PHP `stdClass` class with the following properties set:

- `$info->Date` – the RFC2822 formatted current system time for the mailbox
- `$info->Driver` – the protocol used to access this mailbox (pop3, imap, nntp are the possible results)
- `$info->Mailbox` – the name of the current mailbox
- `$info->Nmsgs` – The number of email messages in the current mailbox
- `$info->Recent` – The number of recent messages in the current mailbox.

For our purposes of pagination, we will use the `Nmsgs` property of this result to determine the total number of messages in our current mailbox and calculate for example the total

number of pages it will take to retrieve them all (given a pre-determined maximum page size). Putting the `imap_check()` function together with the `imap_fetch_overview()` function we can devise our own function that returns emails from the mailbox by page. Let's take a look at this now in a function called `imap_overview_by_page`:

```
<?php

function imap_overview_by_page($connection, int $page = 1, int $perPage = 25, int
$options = 0)
{
    $boxInfo = imap_check($connection);

    $start = $boxInfo->Nmsgs - ($perPage * $page);
    $end = $start + ($perPage - (($page > 1) ? 1 : 0));

    if($start < 1) {
        $start = 1;
    }

    $overview = imap_fetch_overview($connection, "$start:$end", $options);
    $overview = array_reverse($overview);

    return $overview;
}

?>
```

The defined `imap_overview_by_page()` function is fairly self-explanatory. It accepts an IMAP resource as its first parameter, the page from which to start as the second as the `$page` parameter, the total number of results per page as the third `$perPage` parameter and finally the `$options` parameter which serves the same purpose as the identical parameter in the internal `imap_fetch_overview()` function. Within the function we use the just-described `imap_check()` function to retrieve the total number of messages in the current mailbox and then use that value along with the `$page` and `$perPage` parameters to calculate the start and end sequence values needed. Because of the nature of the sequence value once we retrieve the list we call `array_reverse()` to reorganize that page in the proper logical sequence and return it to the user for further use.

As we introduce these functions it should become more clear how we will go about combining all of these tools into the creation of a simple web-based email client. Now that we have explored how to retrieve the overview of a given email message in a mailbox we now need to discuss how to retrieve the full contents of a specific email, including attachments.

### Retrieving and Parsing Specific Messages from IMAP

Thus far we have only downloaded a brief summary of emails from a given mailbox. To download the actual contents of an email, access any attachments, etc., we must introduce the `imap_body()` function, which retrieves the actual body of a message:

```
imap_body(resource $connection, int $msgId [, int $options = 0]);
```

Where `$connection` is the IMAP mailbox resource being accessed and `$msgId` is the ID of the message to access. There are some useful options available to us when downloading the body of an email message worth discussing as well:

- `FT_UID` – Indicates the `$msgId` parameter is the unique ID for the message given by the IMAP server instead of the sequence of the message within the mailbox.
- `FT_PEEK` – Indicates the message should be downloaded but the IMAP server should set the "seen" flag for the message to true. This is useful if you'd like to download the message for programmatic reasons other than the user actually reading the message.

Thus, as was the case when working with the `imap_fetch_overview()` function, by passing the `FT_UID` constant as an option to `imap_body()`, you can download a specific email somewhere within the list of the mailbox by referencing its unique ID rather than its position within the list as a sequence.

Using the `imap_body()` function to download the body of the message is pretty straightforward. In the simplest manifestation of email one would expect the body of the email to simply be the message itself. This can be true, however in almost all cases the body of the email contains within itself a structure known as the MIME format. This format is an ASCII-based format that allows you to create emails that have multiple different versions (i.e. HTML and plain text), as well as attachments. Thus, the body of a modern email often itself is a structure that needs to be parsed, separating, for example, the message of your email (presented both in HTML and plaintext) from the document you attached to that email as an attachment.

The full scope and understanding of the MIME format is beyond the scope of this chapter, however, thankfully, the IMAP extension within PHP does provide some useful tools to help parse out the structure of an email into its meaningful components. This is done through the use of the `imap_fetchstructure()` function:

```
object imap_fetchstructure(resource $connection, int $msgId [, int $options = 0]);
```

Where, as was the case using `imap_body()`, in this case `$connection` and `$msgId` are the connections to the mailbox and the specific message (either by sequence number or unique ID via the `FT_UID` option).

Even without getting into the details of the MIME format itself, the structure returned by `imap_fetchstructure()` can easily become very complicated. The function itself returns an object (of class `stdClass`), which will be the tree-structure of the email retrieved. We say tree-structure because, as you will see, the object returned is but the first top-level node of the structure, and contained within it are potentially other identical in structure child-

nodes for each part and sub-part of the message. Here is the basic structure of a single node object:

- `type` – The Primary type for this node.
- `encoding` – The encoding this node uses to transmit its content (e.g., base64).
- `ifsubtype` – A boolean indicating if there is a subtype provided.
- `subtype` – The MIME subtype of this node.
- `ifdescription` – A boolean indicating if there is a description provided.
- `description` – The description string.
- `ifid` – A boolean indicating if there was an identification string provided.
- `id` – The identification string.
- `lines` – The number of lines in the content of this node.
- `bytes` – The number of bytes used by this node.
- `ifdisposition` – A boolean indicating if there is a disposition string provided.
- `disposition` – The string containing this node's disposition.
- `ifdparameters` – A boolean indicating if there are disposition parameters provided.
- `dparameters` – An array of disposition parameter objects, each having an 'attribute' and 'value' property itemizing the parameters provided for use with the disposition.
- `ifparameters` – A boolean indicating if there are node parameters provided.
- `parameters` – A similar array of objects as described for the `dparameters` attribute, except relating to the parameters for the node.
- `parts` – An array of child-nodes representing another portion of the message.

As can be deduced from the contents of any given node, the structure of a MIME email can be very complicated and the logic required to implement the processing even more so. Rather that demonstrating this logic now in a way that would be difficult to do meaningfully in a stand-alone fashion we will instead move on to our project at hand of a web-based email client wherein we will demonstrate this logic in detail instead.

## Wrapping up IMAP for our Laravel Application

With an introductory knowledge of both the Laravel framework and the PHP IMAP extension out of the way, let's now put the two together in the first step of building our web-based email client. The first step in this process is to create a relatively small helper library that wraps the functionality provided by the IMAP extension into something more

immediately useful to what will be the rest of our Laravel application. We will assume you are starting from a brand-new Laravel project.

The first thing we will do is create a directory and thus a namespace within our Laravel application to house our IMAP library. You can name this whatever you wish, but we will be creating the `app/Library/Imap` directory which corresponds to the `App\Library\Imap` PHP namespace automatically within Laravel.

The first piece of our IMAP library we need to build is the part that connects to an IMAP server. We have decided to architect this as such where the generic logic of connecting to an IMAP server is abstracted away from the specific implementation details of a single IMAP server implementation. Since for our purposes we will be building an IMAP client to work against the Google Gmail IMAP server, we will start with two classes: `App\Library\Imap\AbstractConnection`, which is an abstract class containing the generic implementation details of an IMAP connection, and `App\Library\Imap\GmailConnection`, which contains the specifics for connecting to Google's Gmail IMAP server.

Let's start with the `App\Library\Imap\AbstractConnection` class shown:

```
<?php

namespace App\Library\Imap;

abstract class AbstractConnection
{
    protected $_username;
    protected $_password;

    protected $hostname = '';
    protected $port = 993;
    protected $path = '/imap/ssl';
    protected $mailbox = 'INBOX';

    public function getUsername() : string
    {
        return $this->_username;
    }

    public function getPassword() : string
    {
        return $this->_password;
    }

    public function setUsername(string $username) : self
    {
```

```
        $this->_username = $username;
        return $this;
    }

    public function setPassword(string $password) : self
    {
        $this->_password = $password;
        return $this;
    }

    public function connect(int $options = 0, int $n_retries = 0,
                            array $params = []) : \App\Library\Imap\Client
    {
        $connection = imap_open(
            $this->getServerRef(),
            $this->getUsername(),
            $this->getPassword(),
            $options,
            $n_retries,
            $params
        );

        if(!is_resource($connection)) {
            throw new ImapException("Failed to connect to server");
        }

        return new Client($connection, $this->getServerDetails());
    }

    protected function getServerDetails()
    {
        return [
            'hostname' => $this->hostname,
            'port' => $this->port,
            'path' => $this->path,
            'mailbox' => $this->mailbox
        ];
    }

    protected function getServerRef()
    {
        if(is_null($this->hostname)) {
```

```
            throw new \Exception("No Hostname provided");
        }

        $serverRef = '{' . $this->hostname;

        if(!empty($this->port)) {
            $serverRef .= ':' . $this->port;
        }

        if(!empty($this->path)) {
            $serverRef .= $this->path;
        }

        $serverRef .= '}' . $this->mailbox;

        return $serverRef;
    }
}
```

The purpose of the `App\Library\Imap\AbstractConnection` class, and any class that extends from it, fundamentally is to provide the necessary logic to make a connection to the IMAP server in question using the PHP IMAP extension, and then return a different class yet to be discussed – the `App\Library\Imap\Client` class, which is given this IMAP connection resource and contains all of the functionality of working with that connection.

It is assumed that the logic of the `App\Library\Imap\AbstractConnection` class is relatively straightforward requiring no detailed explanation of how it works beyond presenting the code for it. As previously said, since we are building our client to work with Google Gmail, we will also create a simple `App\Library\Imap\GmailConnection` class that extends `App\Library\Imap\AbstractConnect` to provide specific connection details for Gmail. Below is this class, followed by an example of how it can be used to return a useful Client object:

```
<?php

namespace App\Library\Imap;

class GmailConnection extends AbstractConnection
{
    protected $hostname = 'imap.gmail.com';
    protected $port = 993;
    protected $path = '/imap/ssl';
    protected $mailbox = 'INBOX';

}
```

```
?>

<?php
$connection = new GmailConnection();

try {
$client = $connection->setUsername($username)
                     ->setPassword($password)
                     ->connect();
} catch(\App\Library\Imap\ImapException $e) {
        echo "Failed to connect: {$e->getMessage()}";
}

?>
```

Once a connection has been created and passed into the `App\Library\Imap\Client` class we are now ready to begin implementing the key pieces of IMAP functionality we'll need for our web client.

### The IMAP Client Class

The `App\Library\Imap\Client` class we are about to create houses all of the functionality required to actually interact with an IMAP server using the PHP IMAP extension. It accepts in its constructor two parameters. The first is the IMAP connection resource itself (provided by a class inheriting from `App\Library\Imap\AbstractConnection`) and the specifications of that connection (used to construct the server reference string as necessary):

```
public function __construct($connection, array $spec)
{
    if(!is_resource($connection)) {
        throw new \InvalidArgumentException("Must provide an IMAP connection
resource");
    }

    $this->_prototype = new Message($connection);

    $this->_connection = $connection;
    $this->_spec = $spec;
    $this->_currentMailbox = $spec['mailbox'];
}
```

One thing you will note in the above constructor is we create an instance of a yet unknown class `Message()` and also pass it the connection. This is another class in our library that serves as a container for an individual message from the IMAP server. We employ a

prototype design pattern (where the parent instance is simply cloned every time we want to create a new message object) to allow the developer to implement a Message class easily. We will, however, discuss the Message class in more detail later for now, we should be simply aware that this prototype Message object can be replaced as shown:

```
public function setPrototype(MessageInterface $obj) : self
{
    $this->_prototype = $obj;
    return $this;
}

public function getPrototype() : MessageInterface
{
    return clone $this->_prototype;
}
```

Like the IMAP extension for PHP, the `App\Library\Imap\Client` class we are creating will function on a single mailbox at a time. In the constructor we assigned the initial mailbox to be the same default provided by the class responsible for creating the client, but next we need to implement the methods that allow us to change the active mailbox or retrieve which mailbox the client class is currently working under. This is accomplished by the implementation of two methods, the `getCurrentMailbox()` and `setCurrentMailbox()` methods as shown:

```
public function getCurrentMailbox() : string
{
    return $this->_currentMailbox;
}

public function setCurrentMailbox(string $box, int $options = 0,
                                  int $n_retries = 0) : self
{
    $this->_currentMailbox = $box;

    if(!imap_reopen($this->_connection, $this->getServerRef() .
                    $this->_currentMailbox, $options, $n_retries)) {
        throw new ImapException("Failed to open Mailbox: $box");
    }

    return $this;
}
```

The first method we create, `getCurrentMail()`, is simply a “getter” method which returns the current value of the `Client::$_currentMailbox` property. However the “setter” method is a bit more involved. Not only does it set this same property within the class but also uses the aforementioned `imap_reopen()` method to actually change the mailbox we

are connected to on the IMAP server, thus ensuring all future operations are done against the mailbox specified.

To meaningfully create a web-based IMAP client that supports multiple mailboxes we need to have a sense of which mailboxes are available to us. This brings us to the next method within the client Class, the `getMailboxes()` method. The purpose of this method is as its name implies, to simply return an array listing the various mailboxes available within the IMAP connection. It does so by using the PHP `imap_list()` function to retrieve a list of mailboxes:

```
public function getMailboxes($pattern = '*')
{

    $serverRef = $this->getServerRef();

    $result = imap_list($this->_connection, $serverRef, $pattern);

    if(!is_array($result)) {
        return [];
    }

    $retval = [];

    foreach($result as $mailbox) {
        $retval[] = str_replace($serverRef, '', $mailbox);
    }

    return $retval;
}
```

Note that as previously introduced, the `imap_list()` method requires a server reference string similar that used for `imap_open()` that defines the scope of the list of mailboxes we seek to retrieve. For our purposes, we are interested in the mailboxes at the "root" of the connection and thus have written our method as such. Since the `imap_list()` function returns the list of mailboxes using a fully qualified reference string (including the server name) we strip this information from our final return array. For your reference, here is the `getServerRef()` method referenced as well, which simply constructs the IMAP server connection string for us:

```
protected function getServerRef()
{
    $serverRef = '{' . $this->_spec['hostname'];

    if(!empty($this->_spec['port'])) {
        $serverRef .= ':' . $this->_spec['port'];
    }
```

```
    if(!empty($this->_spec['path'])) {
        $serverRef .= $this->_spec['path'];
    }

    $serverRef .= '}';

    return $serverRef;
}
```

With the methods we have thus far introduced, our IMAP client class now has the ability to list the available mailboxes and switch between them quickly and easily. Next, we need to retrieve a list of emails contained within any given mailbox. Earlier in this chapter we retrieved such lists using the `imap_fetch_overview()` PHP function. If you will recall, we discussed the necessity for such a function to implement some sort of paging mechanism in order to be of practical use, and even implemented a function that did this paging for us in demonstration. We will reuse most of the same code in the `getPage()` method below to retrieve a list of emails within our client class:

```
public function getPage(int $page = 1, int $perPage = 25, $options = 0) :
\Illuminate\Support\Collection
{
    $boxInfo = imap_check($this->_connection);

    $start = $boxInfo->Nmsgs - ($perPage * $page);
    $end = $start + ($perPage - (($page > 1) ? 1 : 0) );

    if($start < 1) {
        $start = 1;
    }

    $overview = imap_fetch_overview($this->_connection,
                                    "$start:$end",
                                     $options);
    $overview = array_reverse($overview);

    $collection = new Collection();

    foreach($overview as $key => $msg) {
        $msgObj = $this->getPrototype();

        $msgObj->setSubject($msg->subject)
               ->setFrom($msg->from)
               ->setTo($msg->to)
```

```
                ->setDate($msg->date)
                ->setMessageId($msg->message_id)
                ->setSize($msg->size)
                ->setUID($msg->uid)
                ->setMessageNo($msg->msgno);

            if(isset($msg->references)) {
                $msgObj->setReferences($msg->references);
            }

            if(isset($msg->in_reply_to)) {
                $msgObj->setInReplyTo($msg->in_reply_to);
            }

            $collection->put($key, $msgObj);
        }

        return $collection;
    }
```

Unlike our original implementation of pagination with the `imap_fetch_overview()` method, our `getPage()` method for our client class has been written specifically for use in both the Laravel environment and using an object-oriented approach. Specifically, rather than returning an array as our original implementation, we now return an instance of the `Illuminate\Support\Collection` class, which is a Laravel framework implementation of an object-oriented Collection. This type of collection class offers much more flexibility than a simple array (please see the Laravel API documentation for a complete description of its functionality). In addition, we are not storing arrays within our collection but instead instances of the Message prototype we first introduced in the discussion of the client's constructor.

If you would like to return a single message rather than a list from a given mailbox using the client, we can build a `getMessage()` method for this purpose. In a similar fashion to `getPage()`, which returns a list of Message prototype instances, `getMessage()` returns a single instance of the Message prototype:

```
public function getMessage($id, int $options = 0) :
\App\Library\Imap\Message\MessageInterface
{
    $overview = imap_fetch_overview($this->_connection, $id, $options);

    if(empty($overview)) {
        return $this->getPrototype();
    }
```

```
    $overview = array_pop($overview);

    $retval = $this->getPrototype();

    $retval->setSubject($overview->subject)
           ->setFrom($overview->from)
           ->setTo($overview->to)
           ->setDate($overview->date)
           ->setMessageId($overview->message_id)
           ->setSize($overview->size)
           ->setUID($overview->uid)
           ->setMessageNo($overview->msgno);

    return $retval;
}
```

As you might be able to deduce from the `getPage()` and `getMessage()` methods, the Message class we have implemented serves largely as an object wrapper containing the various values returned from the `imap_fetch_overview()` method for each email (see the discussion of `imap_fetch_overview()` earlier in this chapter for details). Let's introduce this class and its architectural basis in more detail now.

### The Message and MessageInterface of the IMAP Client

Earlier in this chapter we discussed the concept that our IMAP client class used a prototype pattern to define the object which would be used as a container for any given individual email message in an IMAP inbox. From an implementation perspective, the IMAP client we are creating requires that we provide it with any object that implements the `App\Library\Imap\Messages\MessageInterface` interface defined as follows:

```
<?php

namespace App\Library\Imap\Message;

interface MessageInterface
{
        public function __construct($connection);
        public function setSubject(string $subject);
        public function getSubject() : string;
        public function setFrom(string $from);
        public function getFrom() : string;
        public function setTo(string $to);
        public function getTo() : string;
        public function setDate(string $date);
        public function getDate() : \DateTime;
```

```
        public function setMessageId(string $id);
        public function getMessageId() : string;
        public function setReferences(string $refs);
        public function getReferences() : string;
        public function setInReplyTo(string $to);
        public function getInReplyTo() : string;
        public function setSize(int $size);
        public function getSize() : int;
        public function setUID(string $uid);
        public function getUID() : string;
        public function setMessageNo(int $no);
        public function getMessageNo() : int;
}
```

As a default implementation of this interface, we create what has been previously introduced as the `App\Library\Imap\Message\Message` class. However, for example, this class can be replaced using the `App\Library\Imap\Client::setPrototype()` method to anything of the developer's choosing. For example, one could implement an Eloquent model implementing the described interface and use it as a way to return email messages from the client that can also be easily saved to the database.

For the sake of brevity we will provide the implementation of the `App\Library\Imap\Message\Message` class without including the various standard "getter" and "setter" methods available to it, but only expound on those methods with substantive logic. Please refer to the complete source code if you would like to examine those methods not discussed.

Let us first start with the constructor for the class, which accepts a single parameter (the IMAP connection for the message). It is a straightforward constructor that simply sets the default Date and Time of the message to the present time for consistency within the object:

```
public function __construct($connection)
{
    $this->_date = new \DateTime('now');

    if(!is_resource($connection)) {
        throw new \InvalidArgumentException("Constructor must be passed IMAP
resource");
    }

    $this->_connection = $connection;
}
```

The intended use of this object as you have previously seen in the code for the IMAP client we have created is to on a basic level provide a wrapper to the various properties of a single email message retrieved from a call to `imap_fetch_overview()`. However, it was implemented in an object-oriented fashion to expand upon that basic information to include

the body of the email as well. Specifically, our message class implements the `fetch()` method, which, when populated with the basic parts of the message by the client, uses the PHP IMAP extension to download and process the full content of the email it represents from the mailbox. This method is defined as follows:

```
public function fetch(int $options = 0) : self
{
    $structure = imap_fetchstructure($this->_connection,
                        $this->getMessageNo(), $options);

    if(!$structure) {
        return $this;
    }

    switch($structure->type) {
        case TYPEMULTIPART:
        case TYPETEXT:
            $this->processStruct($structure);
            break;
        case TYPEMESSAGE:
            break;
        case TYPEAPPLICATION:
        case TYPEAUDIO:
        case TYPEIMAGE:
        case TYPEVIDEO:
        case TYPEMODEL:
        case TYPEOTHER:
            break;
    }

    return $this;
}
```

As is shown, the `fetch()` method uses the `imap_fetchstructure()` method explained earlier in this chapter to retrieve the general structure of the body of the email in question. Based on the type property of the root of the email body we can further process the contents of the email appropriately. For the purposes of our library we are only interested in two types when it comes to the "root body" of the email: plain text and MIME multipart emails. These two combined constitute the vast majority of emails in the world today and should be sufficient for our discussions. For both of these root structure types, we handle them using the `processStruct()` method that will be explained next.

The `App\Library\Imap\Message\Message::processStruct()` method is the most complicated code discussed in this chapter. Its function is to take a root-level structure array from the `imap_fetchstructure()` function and, if it is plain text or a multipart MIME

message, deconstruct it into its component parts. Due to the nature of a multipart MIME message, the `processStruct()` method is a recursive method, calling itself repeatedly as it processes deeper and deeper into a multipart MIME message. Here is the logic of the method:

```
protected function processStruct($structure, $partId = null)
{
    $params = [];
    $self = $this;

    $recurse = function($struct) use ($partId, $self) {
        if(isset($struct->parts) && is_array($struct->parts)) {

            foreach($struct->parts as $idx => $part) {
                $curPartId = $idx +1;

                if(!is_null($partId)) {
                    $curPartId = $partId . '.' . $curPartId;
                }

                $self->processStruct($part, $curPartId);
            }
        }

        return $self;
    };

    if(isset($structure->parameters)) {
        foreach($structure->parameters as $param) {
           $params[strtolower($param->attribute)] = $param->value;
        }
    }

    if(isset($structure->dparameters)) {
        foreach($structure->dparameters as $param) {
            $params[strtolower($param->attribute)] = $param->value;
        }
    }

    if(isset($params['name']) || isset($params['filename']) ||
       (isset($structure->subtype) &&
strtolower($structure->subtype) == 'rfc822')) {
```

```
        // Process attachement

            $filename = isset($params['name']) ? $params['name'] :
$params['filename'];

            $attachment = new Attachment($this);

            $attachment->setFilename($filename)
                       ->setEncoding($structure->encoding)
                       ->setPartId($partId)
                       ->setSize($structure->bytes);

            switch($structure->type) {
                case TYPETEXT:
                    $mimeType = 'text';
                    break;
                case TYPEMESSAGE:
                    $mimeType = 'message';
                    break;
                case TYPEAPPLICATION:
                    $mimeType = 'application';
                    break;
                case TYPEAUDIO:
                    $mimeType = 'audio';
                    break;
                case TYPEIMAGE:
                    $mimeType = 'image';
                    break;
                case TYPEVIDEO:
                    $mimeType = 'video';
                    break;
                default:
                case TYPEOTHER:
                    $mimeType = 'other';
                    break;
            }

            $mimeType .= '/' . strtolower($structure->subtype);

            $attachment->setMimeType($mimeType);

            $this->_attachments[$partId] = $attachment;
```

```
            return $recurse($structure);
    }

    if(!is_null($partId)) {
        $body = imap_fetchbody($this->_connection,
                               $this->getMessageNo(), $partId, FT_PEEK);
    } else {
        $body = imap_body($this->_connection, $this->getUID(),
                          FT_UID | FT_PEEK);
    }

    $encoding = strtolower($structure->encoding);

    switch($structure->encoding) {
        case 'quoted-printable':
        case ENCQUOTEDPRINTABLE:
            $body = quoted_printable_decode($body);
            break;
        case 'base64':
        case ENCBASE64:
            $body = base64_decode($body);
            break;
    }

    $subtype = strtolower($structure->subtype);

    switch(true) {
        case $subtype == 'plain':
            if(!empty($this->_plainBody)) {
                $this->_plainBody .= PHP_EOL . PHP_EOL . trim($body);
            } else {
                $this->_plainBody = trim($body);
            }
            break;
        case $subtype == 'html':
            if(!empty($this->_htmlBody)) {
                $this->_htmlBody .= '<br><br>' . $body;
            } else {
                $this->_htmlBody = $body;
            }
            break;
    }
```

```
    return $recurse($structure);
}
```

It is reasonable for the first reaction to such a complicated function would be to feel daunted, but we will break its operations down step by step. We start by initializing a few variables to be used, followed by defining the `$recurse` variable, which is actually an anonymous function. This anonymous function's purpose is to determine if, for the given "part" of the MIME email we are working with, that part itself contains any other sub-parts. If it does, we recurse into the `processStruct()` method, again this time performing the same operations on the child parts. Thus, starting with the root structure, we traverse into the child structures recursively pulling out all relevant data:

```
    $recurse = function($struct) use ($partId, $self) {
        if(isset($struct->parts) && is_array($struct->parts)) {

            foreach($struct->parts as $idx => $part) {
                $curPartId = $idx +1;

                if(!is_null($partId)) {
                    $curPartId = $partId . '.' . $curPartId;
                }

                $self->processStruct($part, $curPartId);
            }
        }

        return $self;
    };
```

You will note that when the `processStruct()` method was called from the `fetch()` method above we only passed the structure array returned by `imap_fetchstructure()`, where in our anonymous function `$recurse` we provide two parameters: the first, the child structure we wish to process next and secondly, a parameter we build in the `$curPartId` variable.

In a multipart MIME message, a good way to think of it is as a tree of nodes, each node consisting of a portion of the content of the email message. For example, let's say you have a hypothetical email message that is available both as formatted text as well as audio, and also includes an attachment. As a multipart MIME message, the hierarchical structure of such an email would be as shown in Figure 29.1.

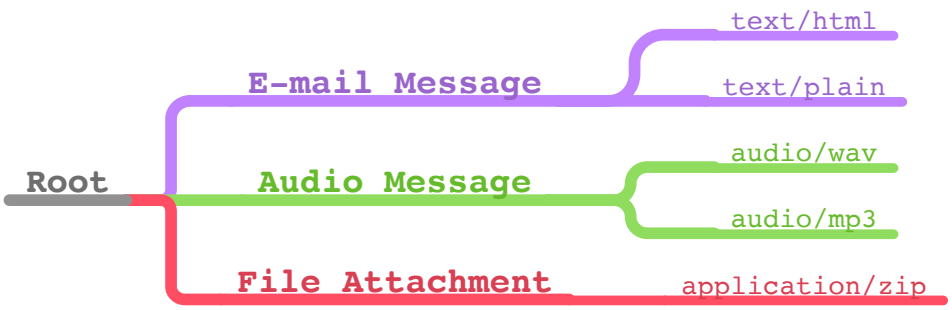


**Figure 29.1.** Structure of an email message.

As you can see, the email would be broken down into three primary parts, the text portion of the email, the audio portion of the email, and the attached file. For each of these parts, however, there may be multiple sub-parts. In this case, for the text portion there are two different versions provided, a plaintext version (text/plain) and an HTML-formatted version (text/html). Likewise for the audio portion, two different versions are provided as a wav file (audio/wav) and an MP3 file (audio/mp3). The attachment, however, does not have a need for any additional parts.

If you were to designate each part by some sort of ID, an easy approach would be to number them based on the relative depth, such as shown in Figure 29.2.

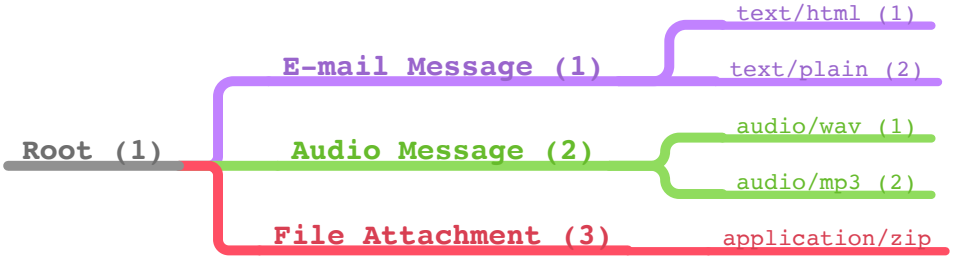


**Figure 29.2.** Numbering each part.

Using a numbering scheme such as this, the plain text version of the email message would be represented by the string "1.1.1", where the Wav version of the audio message would be identified by "1.2.1" and the attached file would simply be "1.3".

This string is known as a part identifier, and it is used by the PHP IMAP extension to identify which portion of a multipart message you are referring to in operations. As our `processStruct()` method recursively traverses the contents of the message, node by node, we build an appropriate part identifier representing it for use later in the method.

The next step in our `processStruct()` method is to combine the part parameters and disposition parameters (if they exist) into a single array, `$params`. By convention, the part parameters and disposition parameters never use the same identifiers even though the

MIME specifically technically allows for it. Since in practicality there is no need to distinguish them, combining them and formatting them into a normalized list (by lowercasing all the identifiers) allows us to quickly find what we are looking for later:

```
if(isset($structure->parameters)) {
        foreach($structure->parameters as $param) {
           $params[strtolower($param->attribute)] = $param->value;
        }
    }

    if(isset($structure->dparameters)) {
        foreach($structure->dparameters as $param) {
            $params[strtolower($param->attribute)] = $param->value;
        }
    }
```

Next we look at the current part we are processing to determine if we think it is a file attachment or not. We do this by examination of various values available to us, such as the aforementioned parameters, and subtype the type the part in question is identified as. For our purposes we specifically look to see if the ‘name’ or ‘filename’ parameter has been provided for this part, which generally means it is an attachment. Additionally, we examine to see if a subtype was specified and if it is equal to ‘rfc822’, which would also indicate an attachment.

```
if(isset($params['name']) || isset($params['filename']) ||
       (isset($structure->subtype) &&
strtolower($structure->subtype) == 'rfc822')) {

        // Process attachement

           $filename = isset($params['name']) ? $params['name'] :
$params['filename'];

           $attachment = new Attachment($this);

           $attachment->setFilename($filename)
                      ->setEncoding($structure->encoding)
                      ->setPartId($partId)
                      ->setSize($structure->bytes);

           switch($structure->type) {
               case TYPETEXT:
                   $mimeType = 'text';
                   break;
               case TYPEMESSAGE:
```

```
                $mimeType = 'message';
                break;
            case TYPEAPPLICATION:
                $mimeType = 'application';
                break;
            case TYPEAUDIO:
                $mimeType = 'audio';
                break;
            case TYPEIMAGE:
                $mimeType = 'image';
                break;
            case TYPEVIDEO:
                $mimeType = 'video';
                break;
            default:
            case TYPEOTHER:
                $mimeType = 'other';
                break;
        }

        $mimeType .= '/' . strtolower($structure->subtype);

        $attachment->setMimeType($mimeType);

        $this->_attachments[$partId] = $attachment;

        return $recurse($structure);
    }
```

If, based on the parameters and subtype of the part, we determine that it is an attachment, we create an instance of a yet-introduced Attachment class to represent it and add it to the `Message::$_attachments` array property using the part ID as its key. For the attachment object we make sure to include the encoding type used (which is the way the attachment was encoded in the email – for example base64), and based on the type of the part we can determine the nature of the attachment itself. We will come back to attachments later, but for now we are only interested in extracting such an attachment for later processing. Once we have done this we are done for the moment and thus we call our anonymous function `$recurse` to determine if any sub parts exist and start the process over again.

If the part we are examining is not in our assessment an attachment to the email, then it must be some portion of the actual message itself. Thus, we need to retrieve the content of the part and determine what to do with it. For the sake of simplicity we assume that if the content of the part is not an attachment, then it must be part of the actual email message. Furthermore, we assume that the actual email message will be either in HTML or plain text format (perhaps both).

The first step is to actually extract this non-attachment content from the segment. If we have been given a part identifier when the `processStruct()` method was called we use the `imap_fetchbody()` function to retrieve that specific part from the message. If we haven't been given a part identifier, then this is not a multiple MIME message and the body is simply the body of the email, and as such we use the `imap_body()` function to retrieve it. In both cases, we make sure to specify the `FT_PEEK` constant so we don't mark the message as "seen" when we get this data:

```
if(!is_null($partId)) {
        $body = imap_fetchbody($this->_connection,
                               $this->getMessageNo(), $partId, FT_PEEK);
    } else {
        $body = imap_body($this->_connection, $this->getUID(),
                          FT_UID | FT_PEEK);
    }
```

Next, we must decode the body we've extracted from whatever encoding was used into its original form by looking at the current structure's encoding property and, based on the encoding specified, decode it using the appropriate PHP function. For the sake of simplicity we only support three encoding types: plain text (no decoding require), quoted-printable, and base64:

```
switch($structure->encoding) {
        case 'quoted-printable':
        case ENCQUOTEDPRINTABLE:
            $body = quoted_printable_decode($body);
            break;
        case 'base64':
        case ENCBASE64:
            $body = base64_decode($body);
            break;
    }
```

At this point in the execution of the `processStruct()` method we have determined a few things. Firstly, we have determined this segment is not an attachment, which by our processor means it must be part of the actual message. Secondly, we have extracted this message and decoded it into its original form. Next, we need to try to determine the nature of this message, specifically determine if it is a plain text message or if it is HTML formatted. To do this, we examine the subtype property of the structure and, based on this subtype, behave accordingly:

```
$subtype = strtolower($structure->subtype);


    switch(true) {
        case $subtype == 'plain':
            if(!empty($this->_plainBody)) {
                $this->_plainBody .= PHP_EOL . PHP_EOL . trim($body);
```

```
            } else {
                $this->_plainBody = trim($body);
            }
            break;
        case $subtype == 'html':
            if(!empty($this->_htmlBody)) {
                $this->_htmlBody .= '<br><br>' . $body;
            } else {
                $this->_htmlBody = $body;
            }
            break;
    }
```

For our purposes, we only support either plain or HTML versions of a message, with all other sub types being ignored. For plain text messages we assign the content of the part to the `Message::$_plainBody` property, where for HTML we similarly assign it to the `Message::$_htmlBody` property. Note that it is possible that a single email message may have multiple segments that contain plain text or HTML, thus in order to ensure all are rendered we include logic to simply append the additional content to the appropriate property.

With all of the necessary steps complete, we again call the `$recurse` method to process any sub types that might exist. Eventually, all of the recursive calls to the `processStruct()` method will return and the end result will be that the `Message::$_htmlBody`, `Message::$_plainBody`, and `Message::$_attachments` properties will all be populated with the contents of the message in question for use elsewhere in the application.

### The Attachment Class

As the result of calling `Message::fetch()` and, in turn, the `processStruct()` method, one or more Attachment classes previously introduced will be created. Each of these objects constitutes a single attachment within a given email message. To round out our IMAP library for this chapter, let's briefly talk about it now. As was the case for the Message class, we will forego including in our discussion "getter" and "setter" methods and only focus on that which is of educational meaning.

Like email messages themselves, and the Message class, the actual content of an attachment is not retrieved from the server until it is explicitly requested. Thus, while calling the `Message::fetch()` method will retrieve the email message in question and define all of the attachments included within it, we do not actually populate the content of the attachment until its own `fetch()` method has been called as shown below:

```
public function fetch() : self
{
    $body = imap_fetchbody(
                $this->_message->getConnection(),
                $this->_message->getMessageNo(),
```

```
                $this->_partId,
                FT_PEEK);

        switch($this->getEncoding()) {
            case 'quoted-printable':
            case ENCQUOTEDPRINTABLE:
                $body = quoted_printable_decode($body);
                break;
            case 'base64':
            case ENCBASE64:
                $body = base64_decode($body);
                break;
        }

        $this->setData($body);

        return $this;
    }
```

After discussing in depth the `Message::processStruct()` method, there should be nothing in the `Attachment::fetch()` method that does not make sense. Like before, we use the `imap_fetchbody()` function to retrieve the specific relevant part of the email message that contains the content of the attachment. Then the body of the attachment is decoded based on the specified encoding before it is simply set using the `setData()` method. Once completed the class now contains the full, decoded contents of the attachment of the email and it can be then sent to the user for download, saved to the file system, etc.

We will circle back on Attachments again later as we implement the actual web interface of our web-based email client.

# Pulling it All Together to Build a Web-based Email Client

We've discussed a lot of different technologies in these two chapters, from the PHP IMAP extension (which we have used to implement a simple object-oriented library to interact with) to an introduction to the Laravel framework. Now, we are going to pull all of these technologies together into the final product – a simple web-based IMAP email client (using the IMAP server provided by your standard Google Gmail account).

We are going to start the process of building this web-based email client using a standard base Laravel 5 project as described in the previous chapter. The first thing that needs to be done once we have created our Laravel project is to incorporate the second piece of technology we've built, which is the object-oriented library that was built using the PHP IMAP extension. We will put this library in the `app\Library\Imap` directory so that it will

be automatically available in the `App\Library\Imap` namespace throughout the Laravel project.

Thanks to the Laravel framework itself, as well as the work we've already done in making this simple IMAP library, building a web-based email client is really just a matter of attaching this library and creating the necessary controllers and views in our Laravel project. Let's start by wiring in the IMAP library we have created by creating a Service Provider for it.

## Implementing the ImapServiceProvider

We will create a simple Laravel Service Provider, the `App\Providers\ImapServiceProvider` class, to gain access to our IMAP client library. If you will recall from earlier in this chapter our library implemented a connection class `GmailConnection()` that served as the entry point into accessing an IMAP server through our library. The purpose of the Laravel Service Provider we are creating then will be to provide to our Laravel application this connection class in such a way that it is ready to be used without the need to further configure it.

For this, we will register from our Service Provider class a new singleton container that will be responsible for configuring the connection and return an instance ready to use:

```
<?php

namespace App\Providers;

use Illuminate\Support\ServiceProvider;
use App\Library\Imap\GmailConnection;

class ImapServiceProvider extends ServiceProvider
{
    public function register()
    {
        $this->app->singleton('Imap\Connection\GMail', function($app) {
            return new GmailConnection(
                        config('imap.gmail.options'),
                        config('imap.gmail.retries')
                        config('imap.gmail.params')
            );
        });
    }
}
```

Looking at the `register()` method of our Service Provider class, we include a simple amount of logic to register a closure to be executed any time something in the application requests an instance of the class identified as `'Imap\Connection\Gmail'`. This closure

itself is also simple, only returning an instance of the `GmailConnection()` class of our library and injecting configuration values into it. The source of the configuration values is the Laravel application configuration, where a reference such as `imap.gmail.retries` refers to the `config/imap.php` file, which should return an array containing the key 'retries'. Thus, if this file exists and has an array with the 'retries' key set, the value will be injected as the second parameter here. If no value exists, the Laravel `config()` function returns null.

Before this Service Provider will be used by our application, we must tell the Laravel framework of its existence. This is done by modifying the 'providers' key of the `config/app.php` configuration file and adding the following value to the array:

```
App\Providers\ImapServiceProvider::class
```

Being added, among other techniques, retrieval of the singleton instance of our fully configured `GmailConnection` class can now be done using the Laravel `App::make()` method and specifying the object reference string mentioned:

```
$gmailConnection = \App::make('Imap\Connection\Gmail');
```

## The Web Client Authentication Page

For our purposes the authentication page of the client should accept a Google Gmail username and password as form inputs. These inputs will be taken by our application and we will attempt to authenticate against the IMAP server using them. Assuming we are successful in our authentication we will then store those credentials in the user's session for use elsewhere. If the authentication is not successful, we will inform the user and ask them to try again.

Let's start by creating the actual HTML view of the authentication page. For the sake of versatility, we will break this as demonstrated before into two blade templates. One, will be a generic layout template and the second will be the specific content of our login page form. This is done so that, if so desired, you can expand the non-authenticated portion of this web client easily from an HTML perspective. The layout blade template we will create should be stored in the `resources/views/layouts/public.blade.php` file and contain the following basic HTML:

```
<html>
<head>
    @section('stylesheets')
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7"
crossorigin="anonymous">
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css"
integrity="sha384-fLW2N01lMqjakBkx3l/M9EahuwpSfeNvV63J5ezn3uZzapT0u7EYsXMjQV+0En5r"
crossorigin="anonymous">
    @show
```

```
</head>
<body>
    <div class="container">

    @if (count($errors) > 0)
        <div class="alert alert-danger">
            <ul>
                @foreach ($errors->all() as $error)
                    <li>{{ $error }}</li>
                @endforeach
            </ul>
        </div>
    @endif

    @yield('main')
    </div>
</body>
    @section('javascript')
    <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js"
integrity="sha384-0mSbJDEHialfmuBBQP6A4Qrprq5OVfW37PRR3j5ELqxss1yVqOtnepnHVP9aJ7xS"
crossorigin="anonymous"></script>
    @show
</html>
```

To greatly simplify the necessary layout and styling of our web site we are going to use the Bootstrap CSS framework as a foundation. Looking at the layout, one can see we define a number of blade sections which can later be extended as necessary in child blade templates. Specifically, we define the 'stylesheets' section (which is located in the `<head>` tag of our document) to make references to stylesheets we might use and includes the necessary Bootstrap framework stylesheet, and the 'javascript' section located at the bottom of the layout to include the necessary JavaScript code used by the Bootstrap framework. We include the JavaScript at the bottom and the stylesheets at the top because browsers typically load resources in order that they are referenced, and thus it is a best practice to always load the JavaScript for a given page after the rest of the content itself has loaded.

You'll also note in our layout that we include a conditional checking of the `$errors` variable. In blade templates there is always available an `$errors` object that serves as a standard structure wherein Laravel stores error messages to be displayed in a view generated within a controller. We will look at that aspect of this in a moment, but for now simply note we include this in the layout so that for any view extending it we provide a standard means of rendering errors to users.

This layout is extended by the specific content for rendering the authentication form we need, which will be stored in the `resources/auth/login.blade.php` file and should look as follows:

```
@extends('layouts.public')

@section('main')
<div class="col-lg-5 col-lg-offset-2">
    <div class="panel panel-default">
        <div class="panel-heading">
            Please Login
        </div>
        <div class="panel-body">
            <form action="/auth/login" method="POST">
                {!! csrf_field() !!}
                <div class="form-group">
                    <label for="email">GMail Username</label>
                    <input type="text" name="email" id="email"
placeholder="user@gmail.com">
                </div>
                <div class="form-group">
                    <label for="password">GMail Password</label>
                    <input type="password" name="password" id="password">
                </div>
                <div class="form-group">
                    <input type="checkbox" name="remember"> Remember Me
                </div>
                <button class="btn btn-block btn-primary" type="submit"><i
class="glyphicon glyphicon-lock"></i> Login</button>
            </form>
        </div>
    </div>
</div>
@stop
```

By in large this view template is basically just as you would expect, an HTML form that provides a means for the user to input their authentication credentials. The only notable aspects of it are that, because it extends from the `layouts.public` blade template, it incorporates this layout when rendering and the inclusion of `{!! csrf_field() !!}` immediately following the `<form>` tag. This is a template function provided by Laravel to inject a special hidden HTML form field designed to prevent cross-site request forgery (CSRF) security vulnerabilities.

With our views created we now need to build the logic behind their function. We will start with defining the necessary route. In this project this route and its corresponding logout route serves as the sole non-authenticated route in our entire application. Since these routes are also designed to be executed via an HTTP request we want to apply the 'web' middleware to the request as well (provided by the Laravel framework). Thus, we define in our `app/routes.php` file the following routes:

```
Route::group(['middleware' => ['web']], function() {
    Route::get('auth/login', [
        'as' => 'login',
        'uses' => 'Auth\AuthController@getLogin'
    ]);

    Route::get('auth/logout', 'Auth\AuthController@getLogout');
    Route::post('auth/login', 'Auth\AuthController@postLoginGMail');
});
```

Now defined, we can implement our `App\Http\Controllers\Auth\AuthController` class that will contain the actual logic of the login form.

```
<?php

namespace App\Http\Controllers\Auth;

use App\User;
use Validator;
use App\Http\Controllers\Controller;
use Illuminate\Foundation\Auth\ThrottlesLogins;
use Illuminate\Foundation\Auth\AuthenticatesAndRegistersUsers;
use Illuminate\Foundation\Auth\AuthenticatesUsers;
use Illuminate\Http\Request;
use App\Library\Imap\ImapException;

class AuthController extends Controller
{
    use AuthenticatesUsers;

    public function __construct()
    {
        $this->middleware('guest', ['except' => 'logout']);
    }

    public function postLoginGMail(Request $request)
    {
        $connection = \App::make('Imap\Connection\GMail');

        $connection->setUsername($request->get('email'))
                   ->setPassword($request->get('password'));

        try {
            $client = $connection->connect();
```

```
        } catch(ImapException $e) {
            return $this->sendFailedLoginResponse($request);
        }

        $credentials = [
            'user' => $request->get('email'),
            'password' => $request->get('password')
        ];

        \Session::put('credentials', $credentials);

        return redirect('inbox');
    }
}
```

Even though there isn't a lot of code in our authentication controller, there is more going on than meets the eye. The `AuthController` class only specifies two methods, one of which is the class constructor, but relies on the Laravel-provided `AuthenticatesUsers` trait to fill in the gaps. This trait implements the methods needed to render the login form, handle logging out, etc.

Thus, the logic of authentication is handled in the following manner:

1. Render the Login form (handled by the `AuthenticatesUsers` trait)
2. Submit the login form (handled by `AuthController::postLoginGMail()`)
3. Logout the user (handled by the `AuthenticatesUsers` trait)

The only method actually implemented by us in `AuthController` is the `postLoginGMail()` method, which uses the singleton client provided by our Service Provider to set the username and password given to us by the user and attempt to connect to the IMAP server using it. If this connection fails, the `connect()` method will throw an `ImapException` error that we can catch to respond to the user that authentication failed. If the authentication was successful, we store the credentials in a session variable 'credentials' using the `Session::put()` method and redirect the user to the named route 'inbox'.

One aspect that we have not yet discussed is how the application will be able to determine an authenticated user who should be given access to the (yet to be shown) authenticated routes vs. an unauthenticated user who can only access the login page. In Laravel this is done via Middleware as previously discussed earlier in the chapter. For our purposes, we will need to modify the default `App\Middleware\Authenticate` class provided in a basic Laravel project to judge if the user is authenticated by our own unique means. Specifically, access should be allowed or denied based on the existence of the 'credentials' Session variable set in the `postLoginGMail()` method of the `AuthController`. Here's what our modified Authenticate class looks like:

```
<?php
```

```
namespace App\Http\Middleware;

use Closure;
use Illuminate\Support\Facades\Auth;

class Authenticate
{
    public function handle($request, Closure $next, $guard = null)
    {

        if(!\Session::has('credentials')) {
            if($request->ajax()) {
                return response('Unauthorized.', 401);
            }

            return redirect()->guest('auth/login');
        }

        return $next($request);
    }
}
```

As with most Laravel middleware, a single method `handle()` is implemented that is provided an instance of the request and a closure representing the "next" middleware to be executed in the chain. In this method we determine if the 'credentials' session variable exists and use that to determine if the user is "authenticated" or not. If they are not authenticated we redirect the user to the login page for a normal request or simply return a 401 HTTP error if the request was done via AJAX. If, however, we determine the user is indeed authenticated we do nothing to change the flow and simply call the next middleware in the chain.

Putting all of these things together we now have a primitive demonstration of a custom Google Gmail authentication mechanism for our IMAP web client and can now move on to implementing the actual feature set of the client itself.

## Implementing the Main View

Moving forward, all of the logic of our web-based email client will be contained within a single controller class `App\Http\Controllers\InboxController` for simplicity. In a more complicated application it would be wise to separate logic into multiple controllers, but for the sake of demonstration this will not be necessary here.

The features we will be implementing largely reflect the features of the IMAP library we built earlier in the chapter:

- Retrieve a list of emails from a given mailbox.
- Display a list of available mailboxes and change between them.
- Read a specific message inside of a given mailbox, including attachments.
- Delete a specific message from a mailbox.
- Compose a new message.

For each of these tasks, the same formula will apply in terms of workflow of the application:

- Ensure the user is authenticated.
- Retrieve the credentials for the user from the session.
- Use the IMAP library we built to connect to the server.
- Perform some set of actions against the server.
- Render the results to the user.

As we discussed in the last section we have already modified Laravel's built in middleware to work for our authentication needs. However, before this middleware will be used it needs to be registered to be used in conjunction with the relevant routes. So the first step in building out the remainder of our application is to define the routes we need and indicate that all of them must first pass an authentication check before being made available. To do this we will again return to our `app/routes.php` file and add a new route group employing our custom authentication middleware:

```
Route::group(['middleware' => ['web', 'auth']], function () {

    Route::get('inbox', [
        'as' => 'inbox',
        'uses' => 'InboxController@getInbox'
    ]);

    Route::get('read/{id}', [
        'as' => 'read',
        'uses' => 'InboxController@getMessage'
    ])->where('id', '[0-9]+');

    Route::get('read/{id}/attachment/{partId}', [
        'as' => 'read.attachment',
        'uses' => 'InboxController@getAttachment'
    ])->where('partId', '[0-9]+(\.[0-9]+)*');
```

```
    Route::get('compose/{id?}', [
        'as' => 'compose',
        'uses' => 'InboxController@getCompose'
    ])->where('id', '[0-9]+');

    Route::get('inbox/delete/{id}', [
        'as' => 'delete',
        'uses' => 'InboxController@getDelete'
    ])->where('id', '[0-9]+');

    Route::post('compose/send', [
        'as' => 'compose.send',
        'uses' => 'InboxController@postSend'
    ]);

});
```

As shown, we define a new route group that contains six routes matching roughly to the five features we are developing. There are six because, including attachments, two separate routes are needed to contain the logic for reading a message from a mailbox.

Since the vast majority of the methods we implement will require a valid IMAP connection through the IMAP library previously developed, let's start off by introducing the `getImapClient()` method, which simply returns a valid connected instance of our `Imap` client object:

```
protected function getImapClient()
{
    $credentials = \Session::get('credentials');

    $client = \App::make('Imap\Connection\GMail')
                  ->setUsername($credentials['user'])
                  ->setPassword($credentials['password'])
                  ->connect();

    return $client;
}
```

Let's now begin with the primary page of our application, the named route 'inbox', which is responsible for displaying to the user all of the email within a given mailbox in a pleasant and paginated form. The code for the `\App\Http\Controllers\InboxController::getInbox()` method is as follows:

```
public function getInbox(Request $request)
{
    $client = $this->getImapClient();
```

```
    $currentMailbox = $request->get('box', $client->getCurrentMailbox());

    $mailboxes = $client->getMailboxes();

    if($currentMailbox != $client->getCurrentMailbox()) {

        if(in_array($currentMailbox, $mailboxes)) {
            $client->setCurrentMailbox($currentMailbox);
        }
    }

    $page = $request->get('page', 1);
    $messages = $client->getPage($request->get('page', 1));

    $paginator = new LengthAwarePaginator(
            $messages,
            $client->getCount(),
            25,
            $page, [
                'path' => '/inbox'
    ]);

    return view('app.inbox', compact('messages', 'mailboxes', 'currentMailbox',
'paginator'));
}
```

The meat of the `getInbox()` method starts by determine the current mailbox we are working with. If no new mailbox is specified, we default to whatever the current mailbox of our IMAP client is. If the current mailbox has been changed compared to what the client is using, we update the current mailbox appropriately. We then call the client's `getMailboxes()` method which, as previously discussed, returns a list of all the available mailboxes for this connection. We then retrieve the current page of emails within the current mailbox (using the page number specified by the user's input) and create a Laravel paginator object to provide a means to paginate our results in the UI. Finally, we pass all of this data into the `app.inbox` view shown next to render it to the user:

> **Note**
>
> The Laravel Paginator component is a useful class available in Laravel projects to provide painless pagination of large collections of data, such as we are potentially dealing with in an email mailbox. When working with native Laravel collections (such as when using Eloquent) this functionality is baked-in to models. Since we are doing something more unique we create an instance of the `LengthAwarePaginator()` manually and feed it the necessary data. Please see the Laravel documentation for a complete description of this component's API.

```
@extends('layouts.authed')

@section('stylesheets')
@parent
<link href="/css/app.css" rel="stylesheet"/>
@stop

@section('main')
<div class="row">
    <div class="col-md-3">
        <div class="text-center"><h2>Mailboxes</h2></div>
        <div class="panel panel-default">
            <div class="panel-body">
                <a href="/compose" class="btn btn-primary btn-block">Compose</a>
                <ul class="folders">
                    @foreach($mailboxes as $mailbox)
                    <li>
                        <a href="/inbox?box={{{ $mailbox }}}"><i class="glyphicon
glyphicon-inbox"></i> {{{ $mailbox }}}</a>
                    </li>
                    @endforeach
                </ul>
            </div>
        </div>
    </div>
    <div class="col-md-9">
    <div class="text-center"><h2>Webmail Demo - {{{ $currentMailbox }}}</h2></div>
        <div class="panel panel-default">
            <div class="panel-body">
                <ul class="messages">

                    @foreach($messages as $message)
                    <li>
                        <a href="/read/{{ $message->getMessageNo() }}"
class="nohover">
                            <div class="header">
                                <span class="from">
                                    {{{ $message->getFrom() }}}
                                    <span class="pull-right">
                                        {{{ $message->getDate()->format('F jS, Y
h:i A') }}}
                                    </span>
                                </span>
```

```
                            {{{ $message->getSubject() }}}
                        </div>
                    </a>
                    <hr/>
                </li>
                @endforeach
            </ul>
        </div>
    </div>
    <div class="text-center">
        {{ $paginator->render() }}
    </div>
</div>
</div>
@stop
```

Looking at the `app.inbox` view, you will note that we are extending from a different yet-introduced blade layout template, `layouts.authed`. This layout is identical to the `layouts.public` blade template we discussed when we looked at authentication but with one difference, which is that we include a number of conditionals that check session variables and render alert-bars to the user. This allows us, as you will see, to pass informational messages back to the user after an action is taken in an eloquent way. In the interest of completeness this new layout is also provided below:

```
<html>
<head>
    @section('stylesheets')
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap.min.css"
integrity="sha384-1q8mTJOASx8j1Au+a5WDVnPi2lkFfwwEAa8hDDdjZlpLegxhjVME1fgjWPGmkzs7"
crossorigin="anonymous">
    <link rel="stylesheet"
href="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/css/bootstrap-theme.min.css"
integrity="sha384-fLW2N01lMqjakBkx3l/M9EahuwpSfeNvV63J5ezn3uZzapT0u7EYsXMjQV+0En5r"
crossorigin="anonymous">
    @show
</head>
<body>
    <div class="container">

    @if (count($errors) > 0)
        <div class="alert alert-danger">
            <ul>
                @foreach ($errors->all() as $error)
                    <li>{{ $error }}</li>
                @endforeach
```

```
            </ul>
        </div>
    @endif

    @if(Session::has('success'))
    <div class="alert alert-success" role="alert">{{ Session::get('success')
}}</div>
    @endif

    @if(Session::has('error'))
    <div class="alert alert-danger" role="alert">{{ Session::get('error') }}</div>
    @endif

    @if(Session::has('warning'))
    <div class="alert alert-warning" role="alert">{{ Session::get('warning')
}}</div>
    @endif

    @if(Session::has('info'))
    <div class="alert alert-info" role="alert">{{ Session::get('info') }}</div>
    @endif

    @yield('main')
    </div>
</body>
    @section('javascript')
    <script
src="https://maxcdn.bootstrapcdn.com/bootstrap/3.3.6/js/bootstrap.min.js"
integrity="sha384-0mSbJDEHialfmuBBQP6A4Qrprq5OVfW37PRR3j5ELqxss1yVqOtnepnHVP9aJ7xS"
crossorigin="anonymous"></script>
    @show
</html>
```

Returning to the `app.inbox` layout, we divide the interface into two columns. The sidebar column, which lists the available mailboxes and the link to compose a new message, and the primary segment that renders the list of emails within the current inbox for the designated page, giving the user the ability to click on one to open it. At the bottom of the template we output the result of a call to the `render()` method of the `$paginator` we passed in, which automatically renders a nice widget that allows the user to move through the pages of messages within the inbox easily (Figure 29.3).

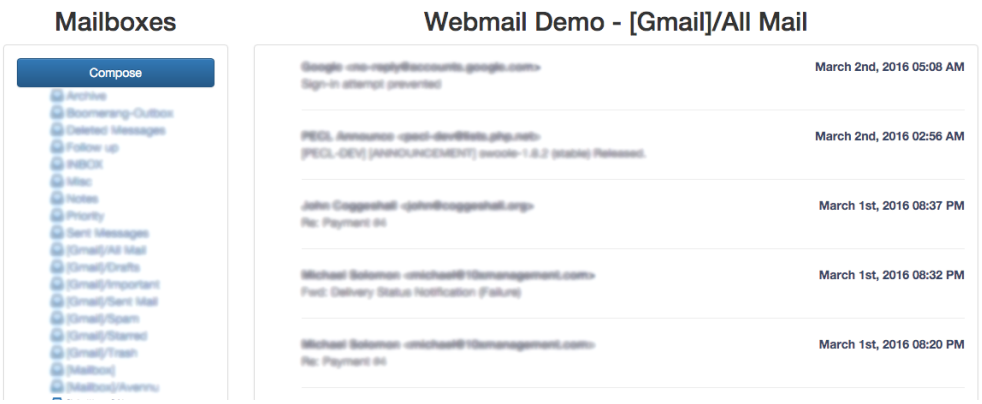


**Figure 29.3.** An example output of the InboxController::getInbox() method.

The next method piece of functionality to implement is the ability to read a message within a given Mailbox. Looking at our blade template for rendering the list of available emails, we can see the route used in our application `/read/{id}` mapping to the `InboxController::getMessage()` method.

### Implementing View Message

The next piece of functionality we will implement is the ability for a user to read an email within the web client when clicked. This is handled by the `InboxController::getMessage()` method shown below:

```
public function getMessage(Request $request)
{
    $client = $this->getImapClient();

    $currentMailbox = $request->get('box', $client->getCurrentMailbox());

    $mailboxes = $client->getMailboxes();

    if($currentMailbox != $client->getCurrentMailbox()) {

        if(in_array($currentMailbox, $mailboxes)) {
            $client->setCurrentMailbox($currentMailbox);
        }
    }

    $messageId = $request->route('id');

    $message = $client->getMessage($messageId)->fetch();
```

```
    return view('app.read', compact('currentMailbox', 'mailboxes', 'message'));
}
```

In a number of ways the `InboxController::getMessage()` method is similar to the `InboxController::getInbox()` method we just introduced, because both views share very similar requirements. Like the main view of our client, the view to read an email still shows the list of mailboxes, so both methods open with the same logic. The `InboxController::getMessage()` method distinguishes itself only in the last few lines where we retrieve the message ID of the message we would like to read from the route parameters (by using the Laravel `Request::route()` method) and then actually retrieving the details of the message by calling our IMAP client's `getMessage()` method to return an "unloaded" version of the message and then the `fetch()` method, which actually downloads the message from the server. We then pass all of these pieces of data into the `app.read` view to be rendered, shown below:

```
@extends('layouts.authed')

@section('stylesheets')
@parent
<link href="/css/app.css" rel="stylesheet"/>
@stop

@section('main')
<div class="row">
    <div class="col-md-3">
        <div class="text-center"><h2>Mailboxes</h2></div>
        <div class="panel panel-default">
            <div class="panel-body">
                <a href="/compose" class="btn btn-primary btn-block">Compose</a>
                <ul class="folders">
                    @foreach($mailboxes as $mailbox)
                    <li>
                        <a href="/inbox?box={{{ $mailbox }}}"><i class="glyphicon
glyphicon-inbox"></i> {{{ $mailbox }}}</a>
                    </li>
                    @endforeach
                </ul>
            </div>
        </div>
    </div>

    <div class="col-md-9">
    <div class="text-center"><h2>Webmail Demo - {{{ $currentMailbox }}}</h2></div>
        <div class="panel panel-default">
```

```
            <div class="panel-body">
                <div class="header">
                    <span class="from">
                        {{{ $message->getFrom() }}}
                    </span>
                    <span class="subject">
                        {{{ $message->getSubject() }}}
                        <span class="date">
                            {{{ $message->getDate()->format('F jS, Y') }}}
                        </span>
                    </span>
                </div>
                <hr/>
                <div class="btn-group pull-right">
                    <a href="/compose/{{ $message->getMessageNo() }}" class="btn
btn-default"><i class="glyphicon glyphicon-envelope"></i> Reply</a>
                    <a href="/inbox/delete/{{ $message->getMessageNo() }}"
class="btn btn-default"><i class="glyphicon glyphicon-trash"></i> Delete</a>
                </div>
                <div class="messageBody">
                {{ $message }}
                @if(!empty($message->getAttachments()))
                    <hr/>
                    @foreach($message->getAttachments() as $part => $attachment)
                        <a href="/read/{{ $message->getMessageNo() }}/attachment/{{
$part }}"><i class="glyphicon glyphicon-download-alt"></i> {{ $attachment-
>getFilename() }}</a><br/>
                    @endforeach
                @endif
                </div>
            </div>
        </div>
    </div>
</div>

@stop
```

Like the controller, the `app.read` blade template resembles the `app.inbox` template in most ways, except for the major content block of the template which renders a single message in a detailed view instead of a list of messages in the mailbox. This view also provides a number of actions to be taken, specifically replying and deleting the message. Looking at the content immediately following the body of the message, we can see a call to the message object's `getAttachments()` method. Recalling from earlier in the chapter this method returns an array of our IMAP library's Attachment class representing any

attachments found for a given email message. Note that, like the message object, these Attachment classes do not actually download the content of their attachment until told to do so. This is a useful distinction, as for the sake of rendering the email message we do not need the full download of the attachment. Rather, in this view we iterate through the attachments and provide download links to each.

Each download link maps to the route specified by `/read/{messageId}/attachment/{partId}`, and referring back to the routes we have defined this maps to the `InboxController::getAttachment()` method. Let's take a look at this method now:

```
public function getAttachment(Request $request)
{
    $client = $this->getImapClient();

    $messageId = $request->route('id');
    $attachmentPart = $request->route('partId');

    $message = $client->getMessage($messageId)->fetch();

    $attachment = $message->getAttachmentByPartId($attachmentPart)->fetch();

    return response()->make($attachment->getData(), 200, [
        'Content-Type' => $attachment->getMimeType(),
        'Content-Disposition' =>
                "attachment; filename=\"{$attachment->getFilename()}\""
    ]);

}
```

The `InboxController::getAttachment()` method is unique among the controller methods we will look at because its output does not use the View component of Laravel. In this case, we want to present the data to the user as a downloadable file. To accomplish this, using the route parameters provided, first we download the message in question and then the specific attachment using our IMAP library. Once we have the data of the attachment, we use Laravel's `response()` function to construct a custom HTTP response in lieu of rendering a blade template using the View component. The body of this response will be the data of the attachment, and the headers of the response will set the appropriate Content-Type (as specified in the email) along with the Content-Disposition HTTP header. This header will inform the browser it should not attempt to immediately open and process the response, but rather save the response as a file using the filename specified in the header (also taken from the email).

## Implementing Deleting and Sending Message

Next we need to implement the final two functional aspects of our project – deleting a message and sending messages. Deleting a message in our web client is the simplest of logical tasks in our project, implemented by the `InboxController::getDelete()` method shown, which simply uses the already written functionality within our IMAP client library to delete the message and redirect the user back to the primary interface:

```
public function getDelete(Request $request)
{
    $client = $this->getImapClient();

    $messageId = $request->route('id');

    $client->deleteMessage($messageId);

    return redirect('inbox')->with('success', "Message Deleted");
}
```

Note that, when we redirect the user after deleting the message we use the `with()` method to pass a message back to the user. This method takes two parameters, the first is the identifier for the message type (in this case 'success'), and the second is the message itself. This message is stored in the session and destroyed after the completion of the next request and is displayed as part of the `layouts.authed` blade template discussed earlier, which renders the message if it exists in a pleasant alert bar.

Next we need to implement sending messages. In a real-world web-based email client such as this, sending of messages would be handled through the corresponding SMTP servers of the IMAP client we were connecting to. However, this implementation is beyond the scope of this project and chapter. Rather, we will simply use Laravel's built in mail-sending facilities. Technically this is not ideal, as Laravel's mail facilities are almost certainly not the proper ones for the IMAP server we are connecting to, but it does however give us some degree of functionality.

The first method we will look at for sending messages is the `InboxController::getCompose()` method. This method's job is to render the UI for sending an email either as a new message or as a response to an existing email in the mailbox (a reply). The logic for this method is as follows:

```
public function getCompose(Request $request)
{
    $client = $this->getImapClient();

    $mailboxes = $client->getMailboxes();

    $messageId = $request->route('id');

    $quotedMessage = '';
    $message = null;
```

```
    if(!is_null($messageId)) {
        $message = $client->getMessage($messageId)->fetch();
        $quotedMessage = $message->getPlainBody();

        $messageLines = explode("\n", $quotedMessage);

        foreach($messageLines as &$line) {
            $line = ' > ' . $line;
        }

        $quotedMessage = implode("\n", $messageLines);
    }

    return view('app.compose', compact('quotedMessage',
                                       'message', 'mailboxes'));
}
```

Because we want to be able to respond to messages as well as compose new messages, the `InboxController::getCompose()` message has slightly more logic within it than simply fetching the mailboxes and returning an HTML form for the user to enter their message into. If provided an ID of a message to reply to (an optional route parameter), this method needs to fetch the body of the message (in plain text) and "quote" it in the typical way for email clients. Since we don't support sending HTML email in this project, this is simply a matter of some basic string manipulation, appending the string ">" to every line of the email we are responding to. This quoted message, the original message object, and mailbox list are then given to the view for render using the `resources/views/app/compose.blade.php` template:

```
@extends('layouts.authed')

@section('stylesheets')
@parent
<link href="/css/app.css" rel="stylesheet"/>
@stop

@section('main')
<div class="row">
    <div class="col-md-3">
        <div class="text-center"><h2>Mailboxes</h2></div>
        <div class="panel panel-default">
            <div class="panel-body">
                <a href="/compose"
                   class="btn btn-primary btn-block">Compose</a>
```

```
            <ul class="folders">
                @foreach($mailboxes as $mailbox)
                <li>
                    <a href="/inbox?box={{{ $mailbox }}}">
                       <i class="glyphicon glyphicon-inbox"></i>
                       {{{ $mailbox }}}
                    </a>
                </li>
                @endforeach
            </ul>
        </div>
    </div>
</div>

<div class="col-md-9">

<div class="text-center">
    @if(is_null($message))
    <h2>Webmail Demo - Compose</h2>
    @else
    <h2>Webmail Demo - Reply</h2>
    @endif
</div>
    <div class="panel panel-default">
        <div class="panel-body">
            <form action="/compose/send" method="post">
                {!! csrf_field() !!}
                <div class="header">
                    @if(!is_null($message))
                        <span class="from">
                            From: <input class="form-control"
                                   type="text" name="from"
                                   value="{{ $message->getToEmail() }}"/>
                        </span>
                        <span class="to">
                            To: <input class="form-control"
                                type="text" name="to"
                                value="{{ $message->getFromEmail() }}"/>
                        </span>
                        <span class="subject">
                            Subject: <input type="text"
                             class="form-control" name="subject"
```

```
                            value="RE: {{{ $message->getSubject() }}}"/>
                        </span>
                    @else
                        <span class="from">
                            From: <input type="text" name="from"
                                        value="" class="form-control"/>
                        </span>
                        <span class="to">
                            To: <input class="form-control" type="text"
                                       name="to" value=""/>
                        </span>
                        <span class="subject">
                            Subject: <input type="text" name="subject"
                                      value="" class="form-control"/>
                        </span>
                    @endif
                </div>
                <hr/>
                <div class="messageBody">
                <textarea class="form-control replybox"
               name="message" rows="10" >{{{ $quotedMessage }}}</textarea>
                </div>
                <hr/>
                <input type="submit" class="btn btn-block btn-primary"
                       value="Send Email"/>
            </form>
        </div>
    </div>
</div>

@stop
```

This blade template is really two very similar templates in one. If we are provided a message (as indicated by whether the `$message` template variable is null or not), we provide a reply form that fills in the to, from, subject of the email, and also populates the body with the quoted response as generated in the controller. If we are not provided a message then we leave these fields blank, allowing the user to fill them in as necessary. The blade template, of course, also follows the same standards as the other views by displaying all of the mailboxes in the sidebar column and the compose form in the major column of the layout.

The last step in our web-based email client is to implement the logic to actually send this email. This compose/reply form is submitted back to the server and ultimately handled by

the `InboxController::postSend()` method. This is the method responsible for sending the email message using Laravel's standard Mail component, and then redirecting the user back to the main view of our application. We also will make use of the Laravel validation component, which allows us a straightforward way to ensure the input data we have received makes sense before sending the email:

```
public function postSend(Request $request)
{
    $this->validate($request, [
        'from' => 'required|email',
        'to' => 'required|email',
        'subject' => 'required|max:255',
        'message' => 'required'
    ]);

    $from = $request->input('from');
    $to = $request->input('to');
    $subject = $request->input('subject');
    $message = $request->input('message');

    \Mail::raw($message, function($message) use ($to, $from, $subject) {
        $message->from($from);
        $message->to($to);
        $message->subject($subject);
    });

    return redirect('inbox')->with('success', 'Message Sent!');
}
```

The first step in any form submission should always be to verify and validate the input data received from the user as best as possible. This not only simplifies the handling of this data later on but can prevent significant security vulnerabilities. Laravel provides a very robust validation component just for this purpose, accessible using the `validate()` method built into every Laravel controller.

The `validate()` method takes two parameters. The first is the input to validate, which can be any array or `ArrayAccess` object. In this case, we pass the Laravel Request class directly in as this parameter because it does implement the PHP `ArrayAccess` interface. The second parameter is an array of key/value pairs, where the key of each record is the name of the input variable in the first array, and the value is a string of validation rules.

The full breadth of validation rules available to you as a developer through this component is well outside the scope of this Chapter. However, looking at our code we can see all rules follow the same basic format:

```
<rule>[:param1[,param2 [,…]]]
```

Where `<rule>` is the name of the rule (see the Laravel Validator documentation for a list of rules) and each rule may have one or more parameters, indicated by first a colon then a comma-separated list of values. Not all rules have parameters, and each rule to be applied to an input variable should be separated by a pipe "`|`" character.

We make use of the required, email, and max rules in our `InboxController::postSend()` method. The required rule, as its name implies, ensures the input variable exists. The email rule, as its name also implies, ensures the input matches what would be a valid email address (of course, we won't know it for certain until the email is sent and received). The max rule imposes length restrictions on the content, in this case we ensure that the subject is no bigger than 255 characters.

Calling the `validate()` method of a controller is enough to validate the input according to the rules. If the input is invalid, Laravel will automatically take the necessary steps to return the user to the form and inform them of their invalid entry. So, once we've validated the input, we can immediately begin processing it using whatever logic we choose. In our case, we immediately turn our attention to another Laravel component that we will use to send the message.

Like Validation, the Mail component in the Laravel framework is very robust and its full set of features is outside of the scope of this Chapter. It can be said that it does easily allow you to send HTML as well as plaintext mail using a very wide range of transport methods from sendmail, all the way to email service providers like Mandrill. For our usage, we are going to use the `Mail::raw()` method to send a "raw" message without the benefit of a blade template to help us lay it out:

```
\Mail::raw($message, function($message) use ($to, $from, $subject) {
    $message->from($from);
    $message->to($to);
    $message->subject($subject);
});
```

As shown, the first parameter of the `Mail::raw()` method is the content of the message body as a string, in this case exactly what we received from the user. The second parameter is a closure which accepts a Laravel message object. This closure allows you to build any logic needed to set the details of your message such as subject, from, and to fields as shown. Once the closure has been executed, Laravel will take this message object and proceed to send the message using the transport configured by the application. The result in our case is that we send the email composed by the user to the address specified.

# Conclusion

If you've made it this far, then congratulations! You have been exposed to a huge amount of material covering how to build Laravel applications, the PHP IMAP extension, and even a little architectural work that builds a useful object-oriented library. Each of the subjects could be expanded upon into a chapter (or, in some cases, an entire book), so further study

is left as an exercise for the reader. Here are two great resources for you to continue learning about subjects we've discussed here:

- Laravel Framework Documentation: https://laravel.com/docs/
- PHP 7 IMAP Extension Documentation: http://php.net/imap