MESSAGING THE WEB

by Aaron Rosenfeld

Have you ever wanted to send and receive text messages with your PHP scripts, only to be dissuaded by the astounding price third parties charge to facilitate the exchange? How would you like to learn about a free method of utilizing existing SMS gateways?

In North America, over 76 million people own a text-enabled mobile device. Similarly, in the U.K., France and Germany there has been a steady increase in text messaging, with an estimated 79% of adults sending at least one text message per day in 2006. It's no wonder that more and more websites are harnessing the power of text messaging to provide users with mobile access to their Web based services. However, until recently, this type of communication has generally only been available to those willing to fork out several hundred to several thousand dollars for a GSI modem and software that allows it to interact with the SMS (Short Message Service) protocol, limiting the possibility to big companies such as Facebook and Google.

This medium can now be utilized by many Web developers and, with a little ingenuity, can be turned into a very elaborate system to communicate with the outside world. In this article, I will discuss a free method that

PHP: *Any version*

Other Software: *PHP's IMAP extension (optional)*

Useful/Related Links:

- *IMAP extension docs: http://php.net/imap*
- *imap_open() manual page: http://php.net/imap-open*
- *SMS Gateways: http://en.wikipedia.org/wiki/SMS_gateway*
- *PHP and SMS: http://aaron-rosenfeld.com/phpsms*
- *Usage stats: http://mmetrics.com/press/PressRelease.aspx?article=20061003-sms-shorttext*

TO DISCUSS THIS ARTICLE VISIT:
http://c7y-bb.phparchitect.com/viewforum.php?f=10

you can use to send and receive text messages using pure PHP. In addition, I will go over a few of the ways in which text messaging can prove a useful tool in Web development.

Using PHP to interact with the SMS protocol to send text messages is, of itself, nothing new; there are dozens of companies that provide gateways that will allow PHP scripts to send text messages. However, since this means using both third party hardware and software resources, those companies obviously will charge either a per-message or per-month fee. This can easily add up when you're sending out hundreds or thousands of messages.

In this article, I will focus on an alternative method of sending and receiving text messages that does not rely on any remote services. This is made possible with local scripts, written in PHP, that can directly interact with mobile providers. The main motivation for using this method is not only to cut out all related expenses, but also to keep the actual processing of messages on the local server, allowing greater flexibility and security.

The method discussed in this article also allows for multiple endpoints on one server. In other words, instead of having a single number for all incoming text messages, we can use as many numbers as we need. This makes the message handling scripts significantly smaller, and separates unrelated services. For example, we can have mobile users send a text message to one location if they need to interact with an authentication script, another to query a server's status, and a third to check the local weather forecast, rather than having all this functionality crammed into a single location.

Sending Messages

When you use a third party to send a text message, a connection is made and data is sent to the remote service. After that, the third party handles all the processing, taking the raw information and transforming it into something that the SMS protocol can read.

In our case, however, all this processing must be handled locally. Since we don't have the hardware means to turn Web based data into wireless data, we will go about it by using what's known as an SMS Gateway.

Most mobile providers have a service such as this, available for anyone to use. Basically, an SMS Gateway provides a unique email address for each phone number served by the mobile provider. All email sent to that address is forwarded, in the form of a text message, via the SMS Gateway to the corresponding cellphone or mobile device. For example, to send a text message to a Verizon customer at 123-456-7890, an email could be sent to *1234567890@vtext.com*.

This is a list of some of the major mobile carriers' SMS Gateways, but there are of course many, many more, all over the world:

- AT&T—*number@txt.att.net*
- Bell (CA)—*number@txt.bell.ca*
- Rogers—*number@pcs.rogers.com*
- T-Mobile—*number@tmomail.net*
- Verizon—*number@vtext.com*
- Virgin Mobile (U.S.)—*number@vmobl.com*
- Virgin Mobile (CA)—*number@vmoble.ca*

Using SMS gateways makes sending text messages from a PHP script extremely easy, but it does have the limitation of requiring us to know which service provider our target phone number is associated with. This can usually be overcome by having the user simply select the name of their provider while on your website, or by checking the **From** header of incoming messages. Of which, more later.

Receiving Messages

Receiving messages is a bit trickier than sending them. To start with, there is no direct connection between a wireless phone network and the Internet. In addition, there is obviously no phone number associated with your server. This limits our options, in that all incoming text messages must be sent to an email address rather than a phone number. This is generally not a problem, though, since nearly all modern text-enabled phones are able to send messages to an email address.

The overall approach here is to harness this feature. We can accept a text message sent from a phone (or other mobile device) in the form of an email to a predefined address, and then send the necessary data to a PHP script for further processing.

There are two ways to achieve this. Either email can be forwarded directly to a PHP script, or a cronjob can be set up to connect to a POP3 server and check for new mail.

Message Forwarding Approach

The actual method of setting up message forwarding varies depending on the Web and mail server software but, in general, a message pipe is established in much the same manner as if emails were to be sent to a different email address. The recipient of our forwarded mail, however, is a file—a PHP script in this case—rather than an email address.

If you are using hosting that gives you access to a control panel such as CPanel, there should be an easy-to-use GUI for mail forwarding. If you do not have that luxury, or just prefer using the command line, you will need to modify the file containing the servers' email aliases and forwarders, and then (usually) restart the mail server. Although it is not possible to cover all server setups, I will explain in detail how to achieve this on the most common PHP configuration: a server using the LAMP architecture that runs **sendmail** as the mail server.

Let's say your hostname is **abc.com**, and you want *sms@abc.com* to forward all incoming emails to a script located at */var/httpdocs/sms/handler.php*. In */etc/aliases* (this is the default location and may vary depending on your installation), you would need to add the line:

```
sms: "|/var/httpdocs/sms/handler.php"
```

and then restart **sendmail**:

```
/etc/init.d/sendmail restart
```

If you are familiar with Linux, the line we just added to the aliases should ring a bell. It is exactly the same as piping, hence the leading | in the alias. All incoming mail will now be piped to the specified script via standard input, which can then be read by PHP. Note that, in addition to the body of the email, all the headers are also piped to the script. This makes it easy to determine the originating phone number, carrier and so on.

The *handler.php* script itself contains a single function that reads from **stdin**, see below. Don't forget to make sure the script has execute permissions, or **sendmail** will not be able to invoke the pipe!

```
function readStdIn()
{
    $handle = fopen('php://stdin', 'r');
    $content = '';
    while (!feof($handle)) {
        $content .= fread($handle, 1024);
    }
    fclose($handle);
    return $content;
}
```

The first line of the script returns a handle to **php://stdin** which is then read, 1024 bytes at a time, into **$content**. When the end of the **stdin** data is reached, the entire content of the incoming email will be returned by the script. It can then be passed to a processing script.

The Cronjob Approach

As I mentioned earlier, if it's not possible to set up email piping on your server, a cronjob can be used to regularly check the mail server and retrieve all new mail. These messages can then be processed in the same way as if they had been forwarded directly to the script from the mail server. There are a number of ways to do this, but I will cover what I consider to be the easiest: using IMAP to connect to a mail server.

The most obvious downside of this solution is that messages will not be sent to the PHP script in real time, and there will consequently be a delay between when a message is sent and when it is processed.

Another downside is the sole requirement for the IMAP approach: that the target mailbox must actually exist. In the email forwarding method the mailbox does not actually need to be present on the server, since all incoming messages are forwarded to the script directly.

The first step in this approach is to establish a connection to the mail server using the **imap_open()** function. For the purposes of this article, we are going to connect to a non-authenticated mail server and passively read messages—that is, the message will not be deleted until we explicitly do so. The general code for opening a connection in this manner is:

```
$mbox = imap_open('{wherever.com/notls}INBOX',
                  $username,
                  $pass);
```

Depending on your server, the first argument may vary. The basic form of the argument, however, is to have your mail server's IP or hostname (**wherever.com**), followed by any flags you would like to use (**/notls**). Surround the resulting string with curly braces, and then append the mailbox name (**INBOX**).

There are a variety of different flags that can be passed to this function, detailing whether SSL is needed, whether POP3 or IMAP is to be used, whether messages should be automatically deleted, and so on. A complete list can be found in the **imap_open()**

documentation in the PHP manual. The **/notls** flag used above disables the use of TLS encryption, thereby avoiding authentication. We also start in the default mailbox, **INBOX**. The arguments **$username** and **$pass** should of course contain the username and password associated with that mailbox, respectively.

Assuming no there are no errors, **$mbox** now stores a connection to the mail server. The next step is to read the messages in the inbox. For this article, I will assume that all messages in this mailbox are meant to be processed by our script. If you plan on using a single email address for more than one purpose, I would recommend having messages filtered into other mailboxes and then reading those rather than the inbox.

To read each message, we will use the **imap_fetchbody()** function. Note that this function splits the emails into multiple parts. The first part is the header data, which contains metadata regarding the message, and the second part is the message body itself. We will simply concatenate these two sections, rather than keeping them separate. The reasoning behind this is simply that it will allow us to use the same processing function for both the forwarding and cronjob approaches.

As we read each message, we will use **imap_delete()** to mark each message for deletion. Note, we won't actually delete them, just *mark* them for deletion. After reading all the messages, we will close the connection with a call to **imap_close()**, and force deletion of the marked messages with the **CL_EXPUNGE** flag. Here you can see the entire process:

```
$mbox = imap_open('{wherever.com/notls}INBOX',
                  $username, $pass);
$msgs = imap_num_msg($mbox);

for ($i = 1;  $i <= $msgs; $i++) {
    $message = imap_fetchbody($mbox, $i, 0) .
               imap_fetchbody($mbox, $i, 1);
    // Later, we will process the message here
    imap_delete($mbox, $i);
}

imap_close($mbox, CL_EXPUNGE);
```

Processing Messages

After choosing one of the two methods above to intercept or read incoming emails, you have the basic framework for an entire SMS processing system. We will now look into how the contents of the email can be parsed into a usable format.

If you are using the message-forwarding method, when a text message is sent to *sms@abc.com* the handler script will invoke **readStdIn()**. The function will return the entire content of the text message, along with related header data.

If you are taking the cronjob approach, the imap script above will iterate over each message in the inbox and provide access to the exact same data as that returned by **readStdIn()**.

Listing 1 contains an example of the kind of data either approach will offer. As you can see, there is a great deal of additional information besides the actual text message. All that additional header data provides us with a means for determining, not only the subject of the message and the phone number of the sender, but other important information such as when the message was sent, the encoding scheme and the content type. Parsing all of this out into a usable form, is our first task.

Before I start to take you through that, though, we need to quickly go over the standard format of an email so you know which headers can be relied upon, since individual message headers will vary depending on which mobile provider sent the email.

In a well formatted email, headers are expected to be in this form:

```
identifier: value \n\r
```

Every header should split its key and value by a colon, and should be on its own line. Following the final line of header data, there should be exactly one blank line before the email (or in our case, text message) content itself.

Using this knowledge, we can iterate over each line of the data and, if it matches the pattern of a header,

LISTING 1

```
From 1234567890@VTEXT.COM Wed Aug 06 12:23:17 2008
Received: from njbdmta1.airbridge.net ([66.174.3.138])
by sub.somedomain.com with esmtps (TLSv1:DES-CBC3-SHA:168)
(Exim 4.69)
(envelope-from <1234567890@VTEXT.COM>)
id 1KQlnB-0001oB-60
for sms@abc.com; Wed, 06 Aug 2008 12:23:17 -0400
Received: from njbd-wigdb1 (njbdmta [66.174.3.21])
by njbdmta1.airbridge.net (8.13.6/8.13.6) with ESMTP id m76GNCfp023084
for <sms@abc.com>; Wed, 6 Aug 2008 16:23:16 GMT
Date: Wed, 6 Aug 2008 16:23:12 GMT
Message-ID: <14758162.8581726351192.JavaMail.root@njbd-wigdb1>
From: 1234567890@VTEXT.COM
To: sms@abc.com
Subject: Subject In Here
Mime-Version: 1.0
Content-Type: text/plain; charset=utf-8
Content-Transfer-Encoding: 7bit

Contents will be down here
```

add it to a **$headers** array. When the first blank line is encountered, the remainder of the data will be concatenated together and regarded as the message body. My implementation of this is given in Listing 2.

The function in that listing, **parseEmail()**, returns an associative array with two elements: **headers**, which contains an array pairing header keys such as **From**, **To** and **Date** with their respective values (**1234567890@VTEXT.COM**, **sms@abc.com**, **Wed, 6 Aug 2008 16:23:12 GMT**), and **message**, which contains the body of the text message.

And that's all there is to it! A text message has been sent from a phone or other mobile device and has been parsed by a PHP script. The values parsed out of the email can now be used for anything you like.

Interacting With The Server

Now that we have a two-way communication bridge open between mobile users and the server itself, we can jump into some more involved programming.

Since it is physically more difficult to type using a cellphone keypad than it is on a computer keyboard, text messages are generally quite brief. This makes them well suited for sending short commands to the server.

An excellent way to work with the limitation of short commands while still providing a robust and extensible list of functions is to follow the UNIX command structure. Every command has a short identifier, usually three to five characters long, followed by the arguments to be passed to the function represented by that identifier. In addition, every command returns a help message if invalid arguments are passed, or if a help argument is passed.

To make this useful within our text messaging system, we need to look for commands when we process the message body. Listing 3 shows an extract from an extensible command processing function. Note that this function is actually a method of the **MessageHandler** class shown in Listing 6, hence the class variables.

Calling **processOperation()** and passing it the message body and **From** header will initiate the processing of the message. The function looks up the command that has been sent and then calls the associated callback function. How is this command-to-callback relationship set up?

The **events** property of the **MessageHandler** class holds an array that is populated during object construction. Acceptable commands make up the keys of this array, and the associated value is the callback function that should be invoked when the command is sent. These callbacks could be globally defined functions, or they could be class methods.

If a command is intended to invoke a globally available function, the callback value is just the name of the function as a string, like so:

LISTING 2

```
<?php

public static function parseEmail($content)
{
    $lines = explode(chr(10), $content);
    $in_message = false;
    $headers = array();
    $message = '';
    $cur_head = '';

    foreach ($lines as $line) {

        if (!$in_message) {
            if (preg_match("/^([a-zA-Z\-]+): (.*)/", $line,
                                                    $matches)) {
                $cur_head = strtolower($matches[1]);
                $headers[$cur_head] = trim($matches[2]);
            } elseif($cur_head != '') {
                $headers[$cur_head] .= ' ' . trim($line);
            }
        } else {
            $message .= $line . "\n";
        }

        if (trim($line) == '') {
            $in_message = true;
        }
    }

    return array('headers' => $headers,
                 'message' => $message);
}

?>

```

LISTING 3

```
<?php

public function processOperation($msg, $to)
{
    // Split the message into command and arguments
    list($cmd, $args) = split(' ', trim($msg), 2);
    $cmd = trim(strtolower($cmd));
    $args = trim($args);

    // Check if the command exists
    if (array_key_exists($cmd, $this->events)) {

        // Call the function specified as the callback
        if (is_callable($this->events[$cmd])) {
            $response = call_user_func($this->events[$cmd], $args,
                                                            $to);
        }

        // If a response should be issued, send it
        if (is_array($response) && sizeof($response) == 2) {
            $this->pushToPhone($response[0], $response[1], $to,
                                                    $this->from);
        }
    }
}

?>

```

```
function function_name($args, $from)
{
    // Processing goes on here
    return array('response subject here',
                 'response message here');
}
```

If a callback needs to be the method of a class, regardless of whether the method is static or the class instantiated, the callback value must be a two-element array. The first element should be set to the name of a static class or a variable containing an instance of an instantiated class. The second element should be set to the name of the method to be called. See the **staticfunc** and **instancefunc** commands in Listing 4 for examples of this.

Each of the callback functions listed in the **events** array must accept two arguments. The first of these will take any arguments that follow the base command. As an illustration, if the message:

```
somefunc arg1 arg2 arg3
```

is being processed, **somefunc** is assumed to be the command and **arg1 arg2 arg3** will be passed to the associated callback function as **$args**. The second argument passed to the callback, **$from**, is simply the **From** header that was sent along with the original email.

The callbacks referenced in **$events** should also *return* an array that has two elements. This array will be used to send a response back to the phone that originally sent the command. The first element stores the contents of the **Subject** header, and the second the body of the message.

The code in Listing 4 will cause the global function **testFunc()** to be invoked when a text message is sent containing the command **globalfunc**. Assuming all is well, a text message with the subject "Message received" and body "Your test message has been processed" will be sent back. Similarly, if the **staticfunc** command is sent, it will call **static_class::method_name()**, and if the **instancefunc** command is sent, it will call **$class_instance->method_name()**.

LISTING 4

```
<?php

function testFunc($args, $from)
{
    // Do stuff...
    return array('Message received',
                 'Your test message has been processed');
}

$events = array('globalfunc'   => 'testFunc',
                'staticfunc'   => array('static_class',
                                        'method_name'),
                'instancefunc' => array($class_instance,
                                        'method_name'),
);

?>

```

Putting It All Together

We will now use all the elements I have covered here to create a reusable library for handling incoming and outgoing text messages. This will make it easy to set up multiple Web services, since we only need to change the contents of the **$events** array in order to allow different functions to be executed. Two classes, **MessageHandler** and **EmailPipe**, will make up the reusable part of the library, and a single handler script will be used for each service that needs its own endpoint (read: email address).

You can see the **EmailPipe** class in Listing 5, and the **MessageHandler** class in Listing 6.

LISTING 5

```
<?php

class EmailPipe
{
    public static function readStdIn()
    {
        $handle = fopen('php://stdin', 'r');
        $content = '';
        while (!feof($handle)) {
            $content .= fread($handle, 1024);
        }
        fclose($handle);
        return $content;
    }

    public static function parseEmail($content)
    {
        $lines = explode(chr(10), $content);
        $in_message = false;
        $headers = array();
        $message = '';
        $cur_head = '';

        foreach ($lines as $line) {

            if (!$in_message) {
                if (preg_match("/^([a-zA-Z\-]+): (.*)/", $line,
                                                         $matches)) {
                    $cur_head = strtolower($matches[1]);
                    $headers[$cur_head] = trim($matches[2]);
                } elseif($cur_head != '') {
                    $headers[$cur_head] .= ' ' . trim($line);
                }
            } else {
                $message .= $line . "\n";
            }

            if (trim($line) == '') {
                $in_message = true;
            }
        }

        return array('headers' => $headers,
                     'message' => $message);
    }
}

?>

```

The **EmailPipe** class contains two of the methods you saw earlier, **readStdIn()** and **parseEmail()**. The class is used purely to read incoming email from standard input and then separate the headers and body of the message. Since it needs no configuration or special treatment between different handler scripts, **EmailPipe** is a static class and need not be instantiated.

Most of the time, the output of **EmailPipe::readStdIn()**—the message body along with all headers in a single string—will be sent directly to **EmailPipe::parseEmail()**. However, I separated the functionality into two methods so that **parseEmail()** could be sent messages from other sources—namely, a cronjob, where message forwarding is not available.

The **MessageHandler** class contains the majority of the code to parse the text message, invoke the requested functions and handle the dispatching of responses. As you can see, it also has two properties that haven't yet been introduced, **$from** and **$authorized**, both of which are set in the constructor alongside the **$events** array.

$from should simply contain the email address that will be used as the **From** header for outgoing messages. This will usually be the same email address that is piping messages into the script.

LISTING 6

```
<?php

class MessageHandler
{
    private $from;
    private $events;
    private $authorized;

    public function __construct($from, $events,
                                $authorized = array('*'))
    {
        $this->from = $from;
        $this->events = $events;
        $this->authorized = $authorized;
    }

    public function processIncoming($msg, $headers)
    {
        if (in_array('*', $this->authorized) ||
            in_array(strtolower($headers['from']),
                     $this->authorized)) {
            $this->processOperation($msg, $headers['from']);
        }
    }

    public function processOperation($msg, $to)
    {
        list($cmd, $args) = split(' ', trim($msg), 2);
        $cmd = trim(strtolower($cmd));
        $args = trim($args);

        if (array_key_exists($cmd, $this->events)) {

            if (is_callable($this->events[$cmd])) {
                $response = call_user_func($this->events[$cmd], $args,
                                           $to);
            }

            if (is_array($response) && sizeof($response) == 2) {
                $this->pushToPhone($response[0], $response[1], $to,
                                   $this->from);
            }
        }
    }

    private function pushToPhone($sub, $msg, $to, $from)
    {
        $headers = 'From: ' . $from . "\r\n";
        $headers .= 'Reply-To: ' . $from . "\r\n";
        $headers .= 'Return-Path: ' . $from . "\r\n";

        mail($to, $sub, $msg, $headers, '-f ' . $from);
    }
}

?>
```

> **"The header used to determine where a message came from varies from mobile device to mobile device."**

The **$authorized** property holds an array of all the email addresses that are allowed to use the script. Having *1234567890@vtext.com* in the array, for example, will allow the Verizon customer with the phone number 123-456-7890 to send commands to the server. It's sometimes desired that all incoming messages be processed, regardless of their **From** header, however. This can be achieved by including an asterisk in the **$authorized** array, as the default **MessageHandler** constructor does.

To check incoming messages against the list of email addresses, the public method **processIncoming()** was added to the class. It filters out any unauthorized messages, thereby ensuring that it only relays text messages from allowed numbers to the **processOperation()** method that actually invokes the command.

Do not rely solely on this method for protection; it's far from foolproof. A forged **From** header could easily bypass this kind of security measure, thereby allowing illegitimate access to the script.

I wanted to briefly go over the **pushToPhone()** method within the **MessageHandler** class as well. The header that is used to determine where a message came from varies from mobile device to mobile device. Some devices use the **From** header, but others—including Verizon—will ignore it completely and use the

unchangeable **Received** header instead, which offers a little more security. The fourth argument that is passed to the **mail()** function, **'-f ' . $from**, will force **sendmail** (or whatever mail server is running) to send the message as if it were coming from the **$from** email address. In the event that this argument is *not* passed, even if the **From** header is set, PHP will send from whatever user Apache is running under—generally, that will be **nobody**. If this happens, the mobile user will see text messages coming from *nobody@abc.com*, which will most likely lead to confusion!

The handler script for the email forwarding approach, the file to which emails are sent via stdin, is given in Listing 7. The only portion of code that needs to be changed for different endpoints is in the configuration. For obvious reasons, I would not recommend that you put your callback functions directly into this file.

The way the code in Listing 7 processes a message should be relatively straight forward. The email is read from standard input and split into a headers array and body string, which are then passed to an instance of **MessageHandler** for final handling. Don't forget that you'll need the shebang line at the top of the script if it is being executed by a forwarder.

The handler script for the cronjob approach, as shown in Listing 8, is very similar but will step through and process all of the emails in the inbox rather than a single message.

A Word On Security

Always remember that **From** headers can be forged. I cannot stress enough that you should never, ever rely on them alone for security. This is especially true if you plan on using text messaging for server control or command execution at the shell level. Anyone can do just as we are doing here and use the PHP **mail()** function to send a spoofed email from *1234567890@vtext.com*.

If you plan on implementing commands that could potentially have severe repercussions if executed by unauthorized users, be sure to require the sender to respond to a message you send from your script. Even if a script spoofs a **From** header, that script will not be able to read a message you send back to that address if it doesn't have access to the spoofed account.

In addition to this, I would recommend logging every single message that goes into and out of the handler script. Review the logs frequently, and be sure to limit externally accessible commands to safe operations.

Potential Uses

There is a plethora of uses for an SMS-to-PHP library such as this, and I wanted to cover a few of those that I have personally made use of. Each of them can easily be integrated into an existing system to allow your Web applications to interact with mobile phone users.

LISTING 7

```
#!/usr/bin/php -q
<?php

// Set up configuration
$sms_config = array('from'       => 'sms@abc.com',
                    'events'     => array('test' => 'testFunc'),
                    'authorized' => array('12345678900@vtext.com',
                                          '0987654321@txt.bell.ca',
                                          ),
                    );

// Read standard input into string
$message = EmailPipe::readStdIn();

// Parse the string into header and body
$email = EmailPipe::parseEmail($message);

// Create message handler instance
$handler = new MessageHandler($sms_config['from'],
                              $sms_config['events'],
                              $sms_config['authorized']);

// Process the message
$handler->processIncoming($email['message'], $email['headers']);

?>
```

LISTING 8

```
#!/usr/bin/php -q
<?php

// Set up configuration
$sms_config = array('test'       => 'sms@abc.com',
                    'events'     => array('test' => 'testFunc'),
                    'authorized' => array('12345678900@vtext.com',
                                          '0987654321@txt.bell.ca',
                                          ),
                    );

// Open an IMAP connection to the inbox
$mbox = imap_open('{wherever.com/notls}INBOX', 'username', 'pass');

// Create the message handler
$handler = new MessageHandler($sms_config['from'],
                              $sms_config['events'],
                              $sms_config['authorized']);

$msgs = imap_num_msg($mbox);

for ($i = 1; $i <= $msgs; $i++) {
    $message = imap_fetchbody($mbox, $i, 0) .
               imap_fetchbody($mbox, $i, 1);

    // Process the message content
    $email = EmailPipe::parseEmail($message);
    $handler->processIncoming($email['message'], $email['headers']);

    // Mark the message for deletion
    imap_delete($mbox, $i);
}

imap_close($mbox, CL_EXPUNGE);

?>
```

> "Much of the content on your site could be just as easily sent via a text message as via a browser."

LISTING 9

```
<?php

function checkPort($args, $from)
{
    $s = split(':', $args);

    if (sizeof($s) != 2 || !is_numeric($s[1])) {
        return array('Invalid service',
                     'Arguments for this command must be ' .
                     'in the form SERVER:PORT');
    }

    $server = $s[0];
    $port = intval($s[1]);
    $errno = 0;
    $errstr = '';
    $conn = @fsockopen($server, $port, $errno, $errstr, 5);

    if ($conn) {
        fclose($conn);
        return array('Service online',
                     'The service running on ' .
                     $server . ':' . $port . ' is ONLINE');
    }

    return array('Service offline',
                 'The service running on ' .
                 $server . ':' . $port . ' is OFFLINE ' .
                 ' - Error: ' . $errno);
}

?>

```

Accessing small areas of website content is the most obvious and widely implemented use for text messaging from a Web service. A **weather** command, for example, could return a ten day forecast for a given area. Much of the content that is currently on your site can probably be just as easily sent via a text message as via a browser.

Utilizing commands and arguments, it would be possible to have a search feature for your site. If you run a restaurant listing site, for example, a user could text the message **search Philadelphia**—using **Philiadelphia** as the single argument—to find a listing of restaurants in Philadelphia. The possibilities are endless!

A great bonus of using text messaging in Web development is that the receiving script will automatically have the sender's phone number available to it. Using this information alongside a country code and area code database, it is fully possible to localize scripts and provide content that is relevant to your users.

Another use for text messaging is for authentication. Preventing users from creating fake accounts that use up your resources is a problem that has plagued message boards, chat rooms and registration forms for over a decade. The most common website solution is to send the user a confirmation email in which a code or link is given to activate the account. However, with today's technology setting up a temporary email account is far from difficult—even for spam bots.

Text messaging can provide a better alternative. When a user registers on your website, they should enter a phone number and mobile provider in your

registration form. A text message is then dispatched via a method similar to **MessageHandler->pushToPhone()**. The message contains a randomly generated number that has simultaneously been placed in a database. The user must then enter the code that was sent into another online form, which is checked against the database. This guarantees that the user does in fact have access to that phone number and is most likely not a bot.

Server Monitor

Servers run 24/7; however, the server administrator does not always work the same hours. If a server is offline for several hours without anyone being aware of the problem, it can prove catastrophic for a business.

One way to deal with this might be to have a handler script set up for incoming text messages that can send out the server status as a response to a command. This would make it easier to ascertain whether the service is in fact down before calling the server administrator. Listing 9 contains an example of a callback function that could be used to process messages in the form **cmd server:port** to query the status of a service on a given server and port.

Creating an automated version of this server monitor is of course also possible. In Listing 10, you can see an example of a similar script that could be set to run as a scheduled task or cronjob. The script queries three services running on different servers and alerts an administrator, via text message, if any of those services are offline or not responding within an acceptable timeframe. As you can see, the **$ports** array simply holds a lookup for the default ports associated with different services. The **$services** array stores the URL or IP address and the name of the service that should be monitored at that address.

Keep in mind that this is a very simplistic version of a monitoring script, intended purely to illustrate the concept. It has no flood control, and will send a message to **$admin** every time the script is run. If you were to fully implement this functionality, it would be a much better idea to ensure that the administrator will only be alerted once per service outage. To achieve this, a database or even a text file could be used to track the last time notifications were sent.

Signing Off

Although it may not seem obvious, text messaging can be extremely useful for PHP programmers. It can provide us with an easy-to-use method of communicating with those that are not necessarily at a computer but still need access to online services. It can be useful both to end users, in terms of accessing site content, and to IT staff, in terms of server monitoring and maintenance.

I hope that this article has motivated you to use text messaging in your own PHP applications, and demonstrated an easy-to-use, free approach of doing so.

The code I have used throughout this article is a modified version of a more sophisticated script I use for my own text messaging needs. I would love to hear from developers that decide to use this code, or even just the overall concept, as a basis for developing their own SMS scripts.

LISTING 10

```
<?php

$admin = '1234567890@vtext.com';
$ports = array('ftp'  => 21,
               'ssh'  => 22,
               'http' => 80,
         );
$services = array(array('www.some-url.com', 'http'),
                  array('www.some-url.com', 'ssh'),
                  array('www.some-other-url.com', 'ftp'),
                  array('127.0.0.1', 'http'),
            );

foreach ($services as $service) {
    // Assume an existing instance of MessageHandler where the
    // command 'mon' is associated with the callback checkPort()
    $handler->processOperation('mon ' . $service[0] . ':' .
                               $ports[$service[1]], $admin);
}

?>

```

AARON ROSENFELD is currently a freelance Web developer and programmer, but plans to focus on computational programming in the future. He is majoring in Computer Science at a university in Philadelphia, Pennsylvania, and is also preparing for his Zend certification. Aaron actively works on a number of PHP projects that can be found on his programming-related blog at http://aaron-rosenfeld.com. Aaron can be directly contacted at aaron@aaron-rosenfeld.com or through his website.