EMAILING RECEIPTS

The **final.php** script includes **email_receipt.php**, whose role it is to email a receipt to the customer. Because a lot of information could be in this receipt (itemizing multiple products), sending an HTML receipt is a logical choice. Considering that some people like HTML email and others don't, the professional solution is to send an email that's viewable in either HTML (**Figure 10.18**) or plain text format (**Figure 10.19**). That's what **email_receipt.php** will do.

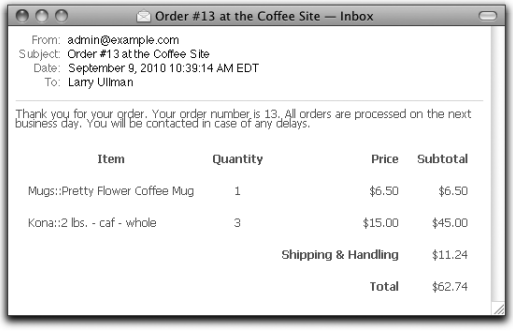

Figure 10.18

Figure 10.19

tip

Authorize.net can send out confirmation emails, too, but you cannot control the format as easily.

In theory, you can create a multipart email (one that's readable in both formats) by just creating the proper body and headers that adhere to the email standard. In my experience, that's much, much easier said than done. A better solution is to use a third-party library that will guarantee accurate and reliable results. For **email_receipt.php**, let's turn to the Zend Framework (**http://framework.zend.com**).

tip

You can also use the PEAR *Mail_Mime* class to send out HTML email.

Installing the Zend Framework

The Zend Framework is created and supported by key PHP developers and has a module for just about anything you'll want to do with PHP. The framework is thoroughly documented and well established. One of the best features of the framework is that you can use pieces of it as needed, without having to embrace or incorporate the entire library. In other words, a site like this one can use just **Zend_Mail** without the entire site being Zend Framework-based. To use the Zend Framework on the site, you'll need to grab a copy of it first.

tip

If you do a lot of PHP development, you ought to be familiar with the Zend Framework, even if you don't routinely use it.

1. Go to **http://framework.zend.com**.
2. Click Downloads > Latest Release.
3. On the Latest Release page, download the *minimal* version.

The framework can be downloaded directly or by registering with Zend.com first. It's up to you which route you choose. The minimal version is an alternative to the full version and includes only the core modules, such as **Zend_Mail**.

4. Expand the downloaded file.

   Depending upon the version you downloaded in Step 3, you'll either have a **.zip** or a **.tar.gz** archive that needs to be expanded.

5. From the expanded framework folder, copy the entire **library** directory to your Web site's root directory.

   You won't actually need the entire Zend Framework library for this site, but there's no harm in copying it all over.

Creating the PHP Script

The **email_receipt.php** page has to send out an email available in two versions: plain text and HTML. This means the script needs to create two separate email bodies.

1. Create a new PHP script in your text editor or IDE to be named **email_receipt.php** and stored in the **includes** directory:

   ```
   <?php
   ```

2. Begin the plain text version of the body:

   ```
   $body_plain = "Thank you for your order. Your order number is
   ➥{$_SESSION['order_id']}. All orders are processed on the next business
   ➥day. You will be contacted in case of any delays.\n\n";
   ```

   The plain text version starts by thanking the customer, indicating the order number, and stating what's to be expected next.

3. Begin the HTML version of the body:

   ```
   $body_html = '<html><head><style type="text/css" media="all">
       body {font-family:Tahoma, Geneva, sans-serif; font-size:100%; line-
       ➥height:.875em; color:#70635b;}
   </style></head><body>
   <p>Thank you for your order. Your order number is ' . $_SESSION[
   ➥'order_id'] . '. All orders are processed on the next business day. You will
   ➥be contacted in case of any delays.</p>
   <table border="0" cellspacing="8" cellpadding="6">
       <tr>
           <th align="center">Item</th>
   ```

(continues on next page)

```
    <th align="center">Quantity</th>
    <th align="right">Price</th>
    <th align="right">Subtotal</th>
  </tr>';
```

The HTML version of the body starts with the beginning HTML code: To create an HTML email, you create an entire HTML page, as if it were to be viewed in a Web browser. You can even include CSS as you would in a standard HTML page.

The body then begins with the same message as in Step 2, plus the start of a table definition.

4. Retrieve the order contents:

```
$r = mysqli_query($dbc, "CALL get_order_contents({$_SESSION[
➥'order_id']})");
while ($row = mysqli_fetch_array($r, MYSQLI_ASSOC)) {
```

The **get_order_contents()** stored procedure returns the details—what products, and in what quantities and at what price—associated with a given order number. The procedure does not return anything regarding the customer, which is fine in this situation.

5. Add each item to both versions of the body:

```
$body_plain .= "{$row['category']}::{$row['name']} ({$row['quantity']})
➥@ \${$row['price_per']} each: $" . $row['subtotal'] . "\n";
$body_html .= '<tr><td>' . $row['category'] . '::' . $row['name'] . '</td>
    <td align="center">' . $row['quantity'] . '</td>
    <td align="right">$' . $row['price_per'] . '</td>
    <td align="right">$' . $row['subtotal'] . '</td>
</tr>
';
```

For the plain text version, the item's name, quantity, price, and subtotal is listed on a single line. For the HTML version, a table row is created listing the same information.

6. Store the shipping and order total for later use:

```
$shipping = $row['shipping'];
$total = $row['total'];
```

After the loop has completed—after every item has been added to the email—the cost of shipping and the total should be appended to the email body. In order to make those values available after the execution of the loop, they're assigned to other variables here.

7. Complete the loop and clear the next results:

```
} // End of WHILE loop.
mysqli_next_result($dbc);
```

Because the **get_order_contents()** stored procedure performs a **SELECT** query, an extra set of results will be returned. These results should be addressed so that other stored procedures that run (by **final.php**, if applicable) won't cause problems. As explained earlier in the chapter, invoking the **mysqli_next_result()** function will mitigate the potential complication.

8. Add the shipping:

```
$body_plain .= "Shipping & Handling: \$$shipping\n";
$body_html .= '<tr>
    <td colspan="2"> </td><th align="right">Shipping &amp;
    ➥Handling</th>
    <td align="right">$' . $shipping . '</td>
</tr>
';
```

For the plain text version, the ampersand can be used. I'm also using double quotation marks in assigning plain text values, because I want to conclude most lines with a newline (**\n**). The **\$$shipping** construct prints a literal dollar sign (the first dollar sign is escaped), followed by the value of **$shipping**. If you tried to use **$$shipping** instead, you'd create a *variable variable*, and PHP would try to insert the value of, say, **$11.24**, which wouldn't work.

For the HTML version, another table row is added. To create the HTML, single quotes are used so as not to conflict with all the double quotes around the attributes. The ampersand has to be represented by the entity version in HTML.

9. Add the total:

```
$body_plain .= "Total: \$$total\n";
$body_html .= '<tr>
    <td colspan="2"> </td><th align="right">Total</th>
    <td align="right">$' . $total . '</td>
</tr>
';
```

10. Complete the HTML body:

```
$body_html .= '</table></body></html>';
```

At this point, both email bodies have been generated and the email can be created and sent.

11. Add the library folder to the include path:

```
set_include_path('./library/');
```

The *Zend_Mail* class may need to include other Zend classes, so the entire Zend library folder needs to be added to PHP's include path. The path value here is relative to **final.php**, which includes **email_receipt.php**.

12. Include the *Zend_Mail* class:

```
include ('Zend/Mail.php');
```

13. Create a **Zend_Mail** object:

```
$mail = new Zend_Mail();
```

This line creates a variable named **$mail**, which will be an object of type *Zend_Mail*. The rest of the code will use this object.

14. Set the *from* and *to* parameters:

```
$mail->setFrom('admin@example.com');
$mail->addTo($_SESSION['email']);
```

The *from* address should be something appropriate for the site. The *to* address is the customer's email, stored in the session on **checkout.php**. The **addTo()** method is used to add recipients to the email.

15. Set the email subject:

```
$mail->setSubject("Order #{$_SESSION['order_id']} at the Coffee
➥Site");
```

The subject includes the order ID.

16. Set the plain and HTML bodies:

```
$mail->setBodyText($body_plain);
$mail->setBodyHtml($body_html);
```

17. Send the email:

```
$mail->send();
```

18. Save the file.

tip

This is a good example how object-oriented programming (OOP) allows you to use existing class definitions without knowing much about OOP yourself.

tip

Using *Zend_Mail*, you can send a single email to as many recipients as you want, and you can also use Cc and Bcc.

tip

The Zend Framework manual has more on *Zend_Mail*, including how to use a specific SMTP server to send the message.

FOR YOUR CONSIDERATION

Even though this chapter presents the checkout process in as streamlined, yet comprehensive, a way as possible, the content still required more than 50 pages and there are any number of variations you could implement. Let's look at a few of the most logical alterations and additions you could make.

Top-Notch Customer Service

One of the best general things you could add would be several obvious ways to contact the site's administrator or support team. This could be done using a combination of a contact form, a help menu, an FAQ page, or a direct phone number. Make it easy and as immediate as possible, for the customer to get help and answers to their questions.

Checking Order Status Online

Even though customers don't have a true account with the site (that is, the ability to log in and log out), it'd be nice if they could check the status of an order. To do that, you could create a form where customers supply their email address and order number, easily found on the original receipt (shown in the Web browser) or the confirmation email.

When the form is submitted, the page would just confirm that the email address matches the order number. You could modify the **get_order_contents()** procedure so that it also returns the shipping status (the date each item shipped), thereby revealing this information to the customer online.

You could also add a comments field to the **orders** table, wherein the administrator could make notes regarding the order as a whole for the customer to see. Another comments field could be added to **order_contents** for notes particular to a given product.

Improving the Security

The security measures taken in this site are fairly tight, and I can recommend using the code and functionality in good conscience. Because all form data is thoroughly validated using regular expressions, most of the functionality remains within the database, and the payment request is made behind the scenes, it's fairly secure.

Outside the Web site itself, one recommendation I would make, which Authorize.net also suggests, is that you change your Authorize.net identifying information regularly. This includes the user account password (used to access the Merchant Interface), the login ID, and the transaction key. These values can all easily be changed (in the Merchant Interface). After you change the login ID and transaction ID, only two lines in **gateway_process.php** need to be updated to account for the changes.

Regardless of what payment gateway you use, change the password frequently!

Preventing Duplicate Orders

The **billing.php** script can take a few moments to execute, once the form is submitted, because it has to send a request to the gateway and await a response. This extra delay can fool customers into thinking their form was not submitted, causing them to perhaps click the submit button again to "correct" the problem. With the site as written, this won't actually create two orders because the gateway will reject duplicate submissions within a default time period of two minutes. Still, it'd be better to avoid this potential problem entirely. And it'd be better to give an indication to the customer that their order is being processed and that a delay is to be expected. Although this is a valuable approach, I omitted it from the **billing.php** script because it requires JavaScript, and I didn't want to further confuse an already complex system.

You can adjust the Authorize.net duplicate-order time window by setting **x_duplicate_window** to some value in seconds.

When it comes to JavaScript, you can either create your own code or use a framework. I'm comfortable with either approach, but frameworks are so easy to use that I recommend that route for beginners. My current favorite framework of choice (and the favorite of many) is jQuery (**www.jquery.com**). I like it because it has excellent browser support, is easy to use, and degrades nicely. To update the billing form to use jQuery...

1. At the end of **billing.html**, add:

   ```
   <script src="https://ajax.googleapis.com/ajax/libs/jquery/1.4.2/
   ➥jquery.min.js" type="text/javascript" charset="utf-8"></script>
   ```

   This line will load the jQuery library from Google's API system. There's a potential performance benefit in using the Google-provided version of the jQuery library instead of adding a copy to the site: If the customer has visited another site that also used this same jQuery library from Google, the user's browser won't have to download the library again, thereby improving how quickly your page loads.

2. Add a **<SCRIPT>** block:

   ```
   <script type="text/javascript" charset="utf-8">
   </script>
   ```

Loading JavaScript near the end of the page can help the page to load faster in the Web browser.

You cannot load an external JavaScript file and execute some JavaScript code using the same **<SCRIPT>** block, so the JavaScript code that does the work will go within these two tags.

3. Within the **<SCRIPT>** block, add:

```
$('#billing_form').submit(function(){
});
```

This is jQuery magic. The **$('#billing_form')** part is a way of selecting an element on the page, specifically the element with an ID value of *billing_form*. The **.submit()** says that when the selected element is submitted, the inline function should be executed. That function's code comes next.

4. Within the curly brackets added in Step 3, disable the submit button:

```
$('#billing_form').submit(function(){
    $('input[type=submit]', this).attr('disabled', 'disabled');
});
```

tip

Alternatively, the submit button can be selected by giving it a unique ID and referring to **$('#submit_id_value')**.

The **$('input[type=submit]', this)** part selects all inputs found within the **#billing_form** element (represented by the special keyword **this**) whose type is *submit*. The **.attr('disabled', 'disabled')** code adds the disabled attribute to the selected element (the submit input), with a value of *disabled*. In sum, when the form is submitted, JavaScript will dynamically turn the submit button's HTML into:

```
<input type="submit" value="Place Order" class="button"
➥disabled="disabled" />
```

tip

I chose to format the processing message using the same button class, because it's prominent, but you'd likely want to use a style that looks distinctly different from the original button.

5. On the next line, still within the **<SCRIPT>** block but after the **});** just created, add:

```
$('#submit_div').html('<p class="button">Processing...</p>');
```

You may be picking up on this already: **$('#submit_div')** selects the element on the page with an ID value of *submitDiv*. The **.html()** method, applied to that selection, can be used to assign new HTML to the element. The specific HTML to be assigned is the **<P>** tag. The effect of this line will be the replacement of the submit button with this message (**Figure 10.20**).

Figure 10.20

6. Add an ID attribute to the form with a value of *billing_form*:

   <form action="/billing.php" method="POST" id="billing_form">

7. Add an ID attribute to the **DIV** that contains the submit button with a value of *submit_div*:

   ** **

8. Save the file.

9. Test the file in your Web browser.

   If you already had the **billing.php** page open, you'll need to reload the page to enable the JavaScript.

tip

Debugging JavaScript can be really tedious for the novice. If you have problems, look online for answers or post a question in my support forums.

Improved Gateway Communications

The **gateway_setup.php** and **gateway_process.php** scripts are the heart of this e-commerce site: requesting monies from the customer and transferring them to the business. With that in mind, you may want to add to what these scripts do.

As a precaution, you could add checks to the **gateway_process.php** script so that it validates the required information prior to attempting the cURL request. The problem, as I've mentioned before, with one file including another, is that there are often assumptions made as to what variables are available and what did or did not happen. Rather than rely upon those assumptions, actual validation would be preferred.

Authorize.net also lets you send along your own custom fields as part of the transaction. A custom field is simply any piece of data with a name not already reserved by the Authorize.net system (essentially, this means anything not named *x_something*). If there is more information you'd like to be associated with the order, you can pass that along. Those custom values will be in the returned response (they aren't stored in the Authorize.net system) and can appear on the email receipt that Authorize.net sends to the customer (if you choose that option). But never use custom fields for sensitive information.

Finally, Authorize.net can take line items as part of the transaction. This would be useful if you had Authorize.net send out the email receipts. See the Authorize.net AIM manual for instructions or other possible values.