Tools and techniques for PHP with Linux

PHP SCRIPTING

PHP is becoming an essential tool for all but the simplest websites. This month we examine PHP in the Linux environment. **BY DAVID SKLAR**

PHP is at home on tiny guestbook sites that handle six visitors a week, and on high-end, high-performance websites with millions of visitors a day. This open-source programming language has a pleasantly smooth on-ramp for new programmers and small projects, but PHP also has the power, flexibility, and developer community to handle just about anything you can throw at it. This month's cover story looks at the world of PHP.

In later articles, we compare some popular PHP Integrated Development Environments (IDEs). Also, we show you how to find PHP scripts in online archives, and we take a close look at PHP development with the open source Eclipse IDE. The final article examines the eZ Components library for PHP.

First, we begin with a hands-on introduction to PHP for the web-savvy user. If you're already experienced with PHP, you may want to skip to the next story, but if you're looking for more background on PHP scripting, read on for a practical introduction to the craft. We hope you enjoy this month's PHP Scripting cover story.

Hello, PHP

The traditional "Hello, World" program in PHP is kind of a let down:

```
Hello, World
```

Passing a file simply containing *Hello, World* to PHP returns *Hello, World*.

Mission accomplished! But what a boring mission.

When given a file to process, PHP only pays attention to the bits between the file "start" and "end" tags. The start tag is *<?php* and the end tag is *?>*. So a more interesting and PHP-centric *Hello, World* looks something like the script shown in Listing 1. Save Listing 1 to a file called *hello.php* on a PHP-enabled web server, and when you visit that page, you'll see a form with:

```
What's your name?
_____________
[ Go! ]
```

Type in your name, hit the Go! button, and then you'll see:

```
Hello, Slartibartfast
```

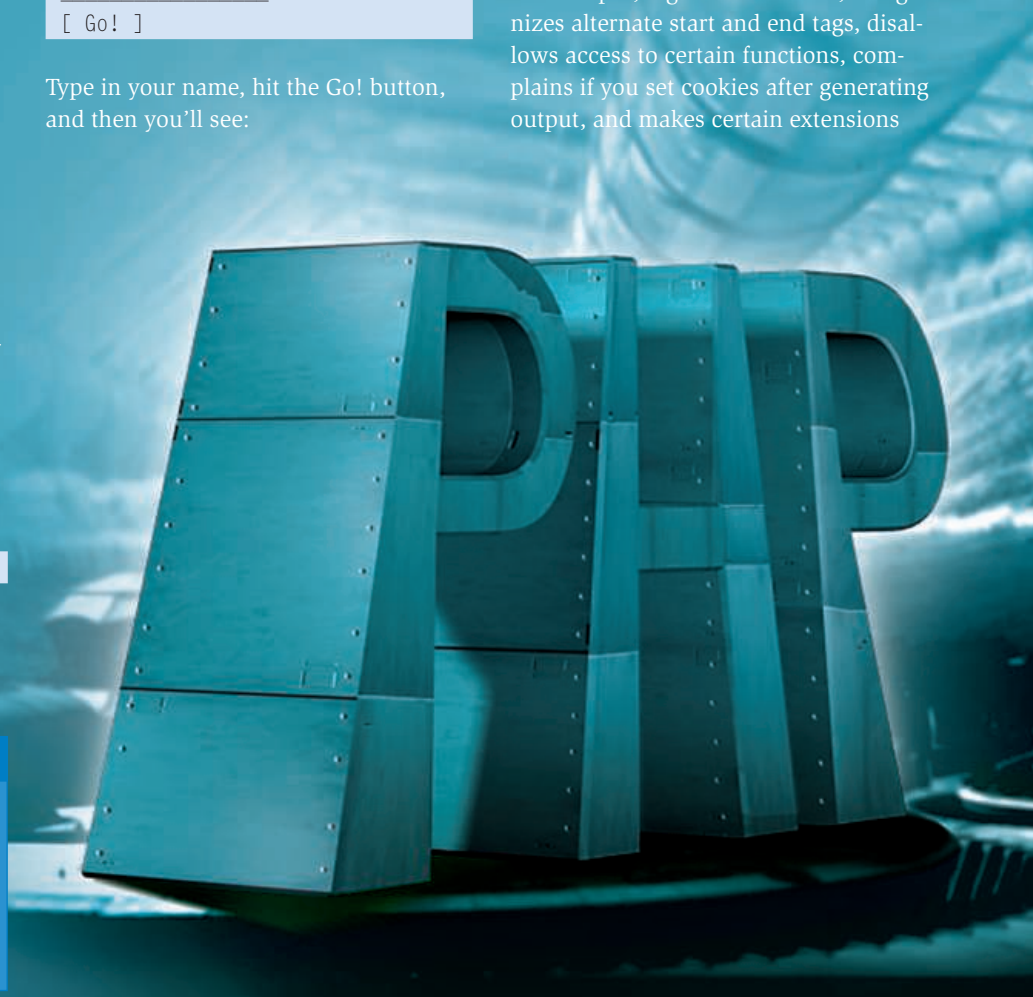
(assuming you've typed *Slartibartfast* into the form field).

This simple example illustrates many of the basic principles of PHP.

PHP Principles

The zeroth principle is that there's an exception to just about every one of the other principles. PHP has approximately 127 million different configuration flags, options, directives, and other ways to change its behavior behind the scenes.

Depending on its configuration settings, PHP intermixes error messages with output, logs errors to a file, recognizes alternate start and end tags, disallows access to certain functions, complains if you set cookies after generating output, and makes certain extensions



COVER STORY

Listing 1: Hello, PHP

```
01 <?php if (! isset($_POST['who_
   are_you'])) { ?>
02
03 <form method="post"
   action="hello.php">
04 <p>What's your name?</p>
05 <input type="text" name="who_
   are_you" />
06 <br/>
07 <input type="submit"
   value="Go!" />
08 </form>
09
10 <?php } else {
11
12 print "Hello, " .
   htmlspecialchars($_POST['who_
   are_you']);
13
14 ?>
```

available or unavailable. The list of configuration settings in the PHP manual (*http://php.net/ini*) is a helpful resource if your server is behaving very differently from how the examples in the article say it should.

Web Server Embrace

The first principle is that PHP exists in the warm, comfortable embrace of a web server. The typical execution model is that a request comes to a web server for a URL that the web server has been told PHP should handle (through pattern matching, filename extension, or something else), the web server translates that URL into a particular file, and then the web server invokes the PHP interpreter, telling it to run the code in that file. PHP runs that code and generates output, which is sent by the web server back to the client as the response to the web request.

"Standard Out" in a PHP program is an http response. The console is a web browser.

The next principle is that PHP only cares about what's between *<?php* and *?>* – everything outside those tags is treated as literal output.

The following PHP programs generate the same output:

```
Hello, World

Hello, <?php print 'World' ?>

<?php print 'Hello, World' ?>

<?php print 'Hello, ' ?>World
```

The PHP open and close tags are completely orthogonal to other code groupings – you can start and end PHP mode inside conditional blocks, function definitions, and just about anywhere else in a PHP program.

Third, PHP sets up arrays containing externally submitted data for you to use. Form data submitted with a *POST* request is in an array called *$_POST* (variable names in PHP begin with the venerable *$*). The submitted value of a form

Some Useful Information

Before diving into more code, I will summarize some tips and bits of background information that will make your PHP experience more pleasant. First of all, take advantage of the online manual at *http://php.net*.

For any built-in function or class, visit *http://php.net/<the function or class>* to get details. For example, if you're curious what the *str_split()* function does, hit *http://php.net/str_split*.

Want to know what the arguments are for *imagecolorallocate()*? *http://php.net/imagecolorallocate* tells you. This is a valuable resource in every PHP developer's toolbox.

Variable names begin with *$*. The rest of the variable name is a letter, numeral, or underscore (except the first character after the *$* can't be a numeral.)

Strings delimited with *'* and *"* behave as you probably expect them to. Everything is literal in *'*-delimited strings, except you must backslash-escape backslash and *'*. Double-quoted strings support more backslash escape characters and variable interpolation. Concatenate two strings together with a period.

Strings are just byte sequences. If you're working with non-ASCII data, check out *http://php.net/mbstring* to learn about the multibyte extension and the tools it gives you for working with multibyte encodings and character sequences. PHP 6 (to be released sometime before Perl 6) has drastically overhauled internationalization support and a comprehensive reworking of multibyte string handling.

If *$alice* is an array, then *$alice[$bob]* gives you the element of *$alice* whose key or index is the value in *$bob*. There's not much difference in PHP between arrays whose keys are all whole numbers and arrays whose keys may be strings. Other languages might call them arrays, maps, hashes, dictionaries, sets, lists – in PHP they're all just arrays.

PHP does give you a few shortcuts if you want numeric array keys – auto-assigns keys starting with *0*: *$dessert = array ('icecream', 'cake');* means that *$dessert[0]* is *ice cream* and *$dessert[1]* is cake. Plus, you can use *[]* to toss something on to the end of an array. *$dessert[] = 'candy'* means that now *$dessert[2]* is *'candy'*.

If you're having trouble with your PHP configuration or things aren't behaving as you expect, make liberal use of the *phpinfo()* function. Just drop a page on your website that contains only:

```
<?php phpinfo(); ?>
```

Then visit the page in your browser. You'll get a dump showing what extensions are loaded and how PHP is configured. Don't leave *phpinfo()* pages wide open for anyone on the Internet to visit, however. Some of the configuration information could make it easier for an attacker to break into your server.

Another handy low-tech debugging tool is the built-in function *var_dump()*. Pass the function any variable – a scalar, an array, whatever – and it will output the variable's type and value. This is handy for a quick check on a variable if it is not behaving as expected.

Last, remember that, because you're typically viewing PHP output in a browser, the formatting of the text you're viewing obeys the rules of HTML, not a terminal. If your PHP script prints out a bunch of strings separated by newline characters, you're going to see them all crammed together on one line in your browser, since a plain newline character (according to the rules of HTML) does not cause a line break.

To get proper line breaks, either use your browser's "View Source" mode or have your code output HTML such as *
*, which tells the browser to insert a newline in the display.

element named *who_are_you* is put in an array element named *who_are_you*.

Some other arrays containing similar kinds of data are:

```
$_GET: query string variables
$_COOKIE: ➥
submitted cookie values
$_SERVER: assorted server-y ➥
data such as the current URL, ➥
current host name, ➥
values from HTTP request headers
$_ENV: environment variables
```

Security Matters

In practice, programs implement varying degrees of authentication, access control, data filtering, and data escaping. A huge portion of PHP security problems are the result of an attack called *cross-site scripting*.

This attack boils down to allowing Evil Alice to upload some content to an innocent (but poorly coded) website such that, when User Bob visits the site, the content is sent to User Bob and then it does something bad to Bob.

There are intricacies to protecting against this sort of attack (and some further reading is listed at the end of the article) but the *htmlspecialchars()* function, used in Listing 1, gets you most of the way. Apply this function to any untrusted external data before you include it in web-page output. The function transforms characters that have special meaning in HTML to their HTML-entity equivalents; the characters it transforms are *&*, *"*, *<*, and *>*. This means that

```
<?php
print htmlentities➥
("<script>alert('hello!');➥
</script>");
?>
```

causes the following to be part of the web page output:

```
&lt;script&gt;alert('hello!');➥
&lt;/script&gt;
```

Without the entity encoding, the browser would see *<script>alert('hello!');</script>* and treat the business inside the *<script/>* tag as JavaScript, popping up a little alert dialog box.

More malicious exploits, however, are the norm. With the entity encoding, the browser displays a *<* when it encounters *<* and a *>* when it encounters *>*. This prevents malicious script code from running on unsuspecting User Bob's computer.

The *isset()* function in Listing 1 tells you whether there's a value for that element of the array. If there is, assume the form has been submitted and go on to print out some data. If not, then display the form. In a real program, you'd probably check the length of the string or its contents here as well.

Fourth, it is incredibly easy to write insecure PHP programs. The task that PHP is usually put to involves web pages that accept submitted data and then manipulate or display that submitted data. Another way of describing that task is "allow anybody in the world to throw arbitrary data into your application and allow anybody in the world to view whatever data you've got."

See the box titled "Security Matters" for more about security in PHP.

Talking to a Database

The most common use for PHP is stuffing information from a web form into a relational database and then generating web pages with information from the database. With recent versions of PHP, the easiest way to do this is with the PDO extension, which provides a standardized access interface no matter which relational database you're talking to. The only thing that changes is how you specify what database you connect to and, potentially, any vendor-specific SQL you want to use. Listing 2 is an extension of the *Hello, World* example above so that the entered names get saved in a database, and then a list of them is displayed after a name is entered.

Save this program as *hello-db.php* to use it on your computer, or change the value of the *action* element of the form to whatever you save the program as.

Database Interaction

Aside from the additional check on the length of the submitted name (*using strlen()*), all the new code in this exam-

Listing 2: Hello, Database

```
01 <?php
02 // Connect to the database
03 $dbh = new PDO('sqlite:/tmp/
   guestbook.db');
04 // Report errors if there are
   database problems
05 $dbh->setAttribute(PDO::ATTR_
   ERRMODE, PDO::ERRMODE_
   WARNING);
06
07 // Make sure a name is
   submitted and it's not empty
08 if (! (isset($_POST['who_are_
   you']) && strlen($_POST['who_
   are_you'])))
09 { ?>
10 <form method="post"
   action="hello-db.php">
11 <p>What's your name?</p>
12 <input type="text" name="who_
   are_you" />
13 <br/>
14 <input type="submit"
   value="Go!" />
15 </form>
16
17 <?php } else {
18   // Print out the
   just-entered name
19   print "Hello, " .
   htmlspecialchars($_POST['who_
   are_you']) . '<br/>';
20
21   // Get other entries from
   the database
22   print "Other entries:";
23   print '<ul>';
24   foreach ($dbh->query('SELECT
   * FROM article_name_test') as
   $row) {
25     printf("<li>%s on %s</
   li>", $row['name'],
26           date('F j, Y',
   strtotime($row['inserted_
   on'])));
27   }
28   print '</ul>';
29
30   // Insert the just-entered
   name into the database
31   $stmt =
   $dbh->prepare('INSERT INTO
   article_name_test
   (name,inserted_on) VALUES (?,
   ?)');
32   $stmt->execute(array($_
   POST['who_are_you'],
   date('c')));
33 }
```

Listing 3: XML Example Output

```
01 MyCorgi.com: A Network for
   Welsh Corgis & Their Owners
02 interstate: For Skaters...By
   Skaters...
03 You WILL Experience the Day of
   the Ninja
04 Ballroom Dance Channel Social
05 You've Got to Hide Your Love
   Away
06 Loud Old Guys
07 Meet Pete
08 Duke City Fix: The Inside Line
   on Albuquerque, NM
09 Mahalo Daily
```

ple has to do with database interaction. The PDO object, created at the beginning of the example, provides access to the database. Exactly which database it's providing access to depends on the DSN (Data Source Name) provided to the *PDO* constructor.

This example uses SQLite, a snappy filesystem-based database that comes with PHP. SQLite is handy for many tasks because it doesn't require installation and tuning of separate software or monitoring of other processes.

The DSN in the example provides the pathname (*/tmp/guestbook.db*) that you'll use for the database.

After connecting to the database and creating a new PDO object, the code calls *$dbh->setAttribute()* to adjust PDO's error-reporting mode.

The *PDO::ERRMODE_WARNING* mode is useful for exploration and debugging because it causes any database errors to be reported as part of PHP's regular output stream.

The rest of the database interaction in the example happens after a valid name is submitted. First, the *$dbh->query()* method is used to retrieve some rows from the table. The method acts as an iterator in PHP, so you can slap it directly inside a *foreach()* loop, and the loop variable (*$row*) is populated with each row from the result set.

By default, rows are represented as arrays, so you can use the column names for array keys to access the data – *$row['name']* contains the value of the *name* column of the row and *$row['inserted_on']* contains the value of the *inserted_on* column of the row.

PHP's *printf()* function acts just like its C-library counterpart, and so it is a tidy way to mix dynamic data into a string. The *strtotime()* function turns the ISO datestamp that comes out of the database into a Unix Epoch timestamp, and the *date()* function uses the format string *F j, Y* to turn that timestamp into a string such as *December 3, 2008*.

The last bit of code in the example inserts the newly submitted name into the database table.

Prepared Statement

The *$dbh->prepare()* function creates a *prepared statement* – a template of an SQL statement that can be executed many times. In each execution, the ?s (called *placeholders*) in the prepared statement are replaced by actual values.

To execute a prepared statement, call the *execute()* method on the statement object and pass it an array containing values to bind to each placeholder.

Two very good reasons exist for using prepared statements instead of building literal SQL queries as strings containing dynamic data: performance and security.

Speed

Depending on the database engine you're using, executing a prepared statement a few times is faster than building a new SQL query with the dynamic data directly inside it each time. With the prepared statement, the database engine can plan ahead and optimize the query.

Security

The potential performance increase isn't such a big deal in this small example; however, the security difference is. Prepared statements protect you from SQL Injection attacks because they take care of properly escaping the dynamic data. In the example, if the *$_POST['who_are_you']* variable contains characters that have special meaning in an SQL query, such as ' (the string delimiter), PDO takes care of escaping them properly so they don't cause any problems.

Without prepared statements, you would have to make sure you escape all dynamic data going into a query; one mistake and you open up your database to all sorts of scurvy treachery or data leakage.

The table used in Listing 2 has the following structure:

```
CREATE TABLE article_name_test (
  name VARCHAR(255) NOT NULL,
  inserted_on DATETIME NOT NULL
)
```

You could create this table in the database with the following PHP program:

```
<?php

$dbh = new PDO➥
('sqlite:/tmp/guestbook.db');
$dbh->exec➥
('CREATE TABLE ➥
article_name_test (
  name VARCHAR(255) NOT NULL,
  inserted_on DATETIME NOT NULL
)');

?>
```

Processing XML

Another popular PHP task is dealing with XML. PHP 5 gives you two splendid choices: SimpleXML module for basic XML parsing, and the DOM module for Document Object Model API action.

With SimpleXML, an XML document is represented as a PHP object:

```
<?php
$feed = simplexml_load_file➥
('http://blog.ning.com/➥
atom.xml');

foreach ➥
($feed->entry as $entry) {
    print $entry->title . "\n";
}
?>
```

This PHP program prints something like the output in Listing 3.

The *simplexml_load_file()* function, like most file-access functions in PHP, can accept URLs as well as local file paths. With one step, the function loads the Atom feed at *http://blog.ning.com/atom.xml* into the *$feed* object.

Because the structure of an Atom feed means the root element has a number of children named <*entry*/>, the entry property of the *$feed* object can be treated as an array with *foreach()*.

Similarly, because each <*entry*/> has a <*title*/> child, the value of the title

property of the *$entry* object is the text content of that *<title/>* child. (Note that while the object properties, such as *entry* and *title*, explicitly correspond to XML element names, there is no requirement that the PHP variable names be the same as the XML element names – it is only for convenience that the example code uses PHP variable names such as *$feed* and *$entry*.)

The SimpleXML module is super for doing something with an Atom or RSS feed, or other straightforward XML processing tasks.

For heavier XML lifting, PHP also supports the full DOM Level 3 API. Level 3 DOM also can be a good choice for XML processing in PHP if you're already familiar with the DOM API from another language, such as JavaScript.

Listing 4 uses the DOM API to build a small XML document.

The XML document that this example prints looks like:

```
<?xml version="1.0"?>
<people>
  <person flavor=➥
  "chocolate">alice</person>
  <person flavor=➥
  "vanilla">bob</person>
  <person flavor=➥
  "strawberry">charlie</person>
</people>
```

Although the syntax in Listing 4 is PHP-specific, the classes and methods are all part of the language-neutral DOM API. A document is represented by a *DOMDocument* object, *DOMDocument::createElement()* creates elements that *DOMDocument::appendChild()* adds to the document, and *DOMElement::setAttribute()* adds an attribute to an element.

The *DOMDocument::saveXML()* method returns the XML representation of the document. If you pass the method a filename, it writes the XML to that file instead.

What Next?

If you're running Linux or Mac OS X, your computer probably already has PHP on it. Generally, Linux distributions are pretty good at bundling up-to-date PHP versions.

Mac OS X is not as good – head over to *entropy.ch* to download Marc Liyanage's great OS X PHP packages. The main PHP distribution site has Windows binaries as well as source code [1].

The PHP manual [2] is a fantastic resource, not only for the comprehensive function and class documentation, but also for all the user-contributed comments. PHP Planet [3], which is an aggregation of popular PHP-themed blogs, is also a good place to keep up with what's going on in the PHP universe.

PECL [4] is the place to go for binary PHP extensions, such as modules for embedding Lua in PHP programs, hooking up with the ImageMagick library, or implementing all sorts of other third-party integrations.

Good places to look for libraries and extensions written in PHP (so no messy compilation for installation) are the PHP Application and Extension Repository (PEAR) [5], Zend Framework [6], and eZ Components [7] sites.

Some popular MVC-style web frameworks for PHP are CakePHP [8], Symfony [9], and Solar [10].

Conclusions

As I've already mentioned, it is easy to write insecure programs in PHP. Don't add to the depressingly large amount of insecure PHP code.

A good resource for learning about more secure PHP programming is *Essential PHP Security* by Chris Shiflett [11]. If you're a server administrator, take a look at Suhosin [12], which prevents assorted malicious local and remote attacks. ■

Listing 4: Building a Doc with DOM

```
01 <?php
02
03 // Some data to use
04 $flavors = array('alice' =>
   'chocolate',
05                 'bob' =>
   'vanilla',
06                 'charlie' =>
   'strawberry');
07
08 // Create a new document
09 $doc = new DOMDocument();
10
11 // Create a root container
   element
12 $root = $doc->createElement('p
   eople');
13 // And add it to the document
14 $doc->appendChild($root);
15
16 // Add an element for each
   person that
17 // contains an attribute with
   the flavor
18 foreach ($flavors as $who =>
   $what) {
19     $el =
   $doc->createElement('person',
   $who);
20
   $el->setAttribute('flavor',
   $what);
21     $root->appendChild($el);
22 }
23
24 // Ensure readable output
   formatting
25 $doc->formatOutput = true;
26 // Display the XML
27 print $doc->saveXML();
28
29 ?>
```

INFO

[1] PHP distribution site: *http://www.php.net/downloads.php*

[2] PHP manual: *http://php.net/manual*

[3] PHP Planet: *http://planet-php.net*

[4] PECL: *http://pecl.php.net*

[5] PHP Application and Extension Repository (PEAR): *http://pear.php.net*

[6] Zend Framework: *http://framework.zend.com*

[7] eZ Components: *http://ez.no/components*

[8] CakePHP: *http://www.cakephp.org*

[9] Symfony: *http://www.symfony-project.org/*

[10] Solar: *http://solarphp.com/*

[11] Shiflett, Chris. *Essential PHP Security*. O'Reilly, 2005.

[12] Suhosin: *http://www.suhosin.org*

THE AUTHOR

David Sklar is a software architect for Ning, Inc. He has written three books on PHP programming. *Learning PHP 5* (O'Reilly) provides a gentle introduction to PHP. *PHP Cookbook* (O'Reilly), co-authored by Adam Trachtenberg, is a collection of recipes for PHP tasks. *Essential PHP Tools* (Apress) is a guide to PHP add-on modules and libraries.