

## History

The first version of PHP, PHP/FI, was developed by Rasmus Lerdorf as a means of monitoring page views for his online resumé and slowly started making a mark in mid 1995. This version of PHP had support for some basic functions, primarily the capability to handle form data and support for the mSQL database. PHP/FI 1.0 was followed by PHP/FI 2.0 and, in turn, quickly supplanted in 1997 by PHP 3.0. PHP 3.0, developed by Andi Gutmans and Zeev Suraski, was where things started to get interesting. PHP 3.0 was a complete rewrite of the original PHP/FI implementation and it included support for a wider range of databases, including MySQL and Oracle. PHP 3.0's extensible architecture encouraged independent developers to begin creating their own language extensions, which served to increase the language's popularity in the developer community. Before long, PHP 3.0 was installed on hundreds of thousands of web servers, and more and more people were using it to build database-backed web applications. PHP 4.0, which was released in 2003, used a new engine to deliver better performance, greater reliability and scalability, support for web servers other than Apache, and a host of new language features, including built-in session management and better OOP support. And, as if that wasn't enough, the current.

## PHP and MySQL: The Well-Matched Couple

It's interesting, at this point, to see what the typical PHP and MySQL application development framework looks like. Usually, such applications are developed on the so-called "LAMP" (Linux, Apache, MySQL, and PHP) platform, wherein each component plays a specific and important role:

- Linux provides the base operating system (OS) and server environment.
- The Apache web server intercepts HTTP requests and either serves them directly or passes them on to the PHP interpreter for execution.
- The PHP interpreter parses and executes PHP code, and returns the results to the web server.
- The MySQL RDBMS serves as the data storage engine, accepting connections from the PHP layer and inserting, modifying, or retrieving data.

Figure 1-1 illustrates these components in action.

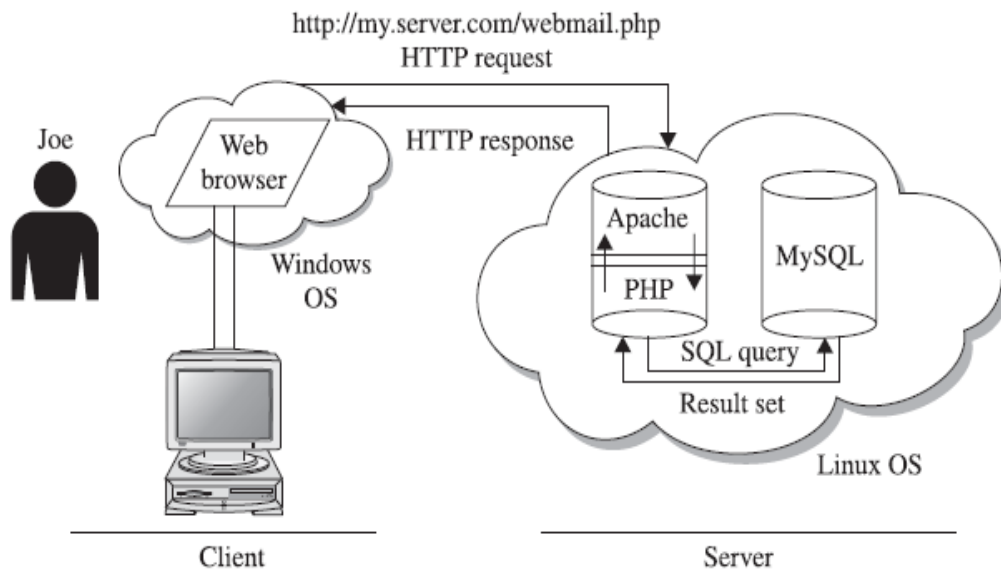


Figure 1-1 The LAMP development framework

Here's what's happening in Figure 1-1:

1. Joe pops open his web browser at home and types in the URL for his online Webmail client. After looking up the domain, Joe's browser (the client) sends an HTTP request to the corresponding server IP address.
2. The Apache web server handling HTTP requests for the domain receives the request and notes that the URI ends with a .php suffix. Because the server is programmed to automatically redirect all such requests to the PHP layer, it simply invokes the PHP interpreter and passes it the contents of the named file.
3. The PHP interpreter parses the file, executing the code in the special PHP tags. If the code includes database queries, the PHP interpreter opens a client connection to the MySQL RDBMS and executes them. Once the script interpreter has completed executing the script, it returns the result to the browser, cleans up after itself, and goes back into hibernation.
4. The results returned by the interpreter are transmitted to Joe's browser by the Apache server.

### **Embedding PHP in HTML**

One of the nicer things about PHP is that, unlike CGI scripts, which require you to write server-side code to output HTML, PHP lets you embed commands in regular HTML pages. These embedded PHP commands are enclosed within special start and end tags, which are read by the PHP interpreter when it parses the page. Here is an example of what these tags look like:

```
<?php
... PHP code ...
```

?>

You can also use the short version of the previous, which looks like this:

<?

... PHP code

?>

To see how this works, create this simple test script, which demonstrates how

PHP and HTML can be combined:

```
<html>
```

```
<head><basefont face="Arial"></head>
```

```
<body>
```

```
<h2>Q: This creature can change color to blend in with its surroundings.
```

```
What is its name?</h2>
```

```
<?php
```

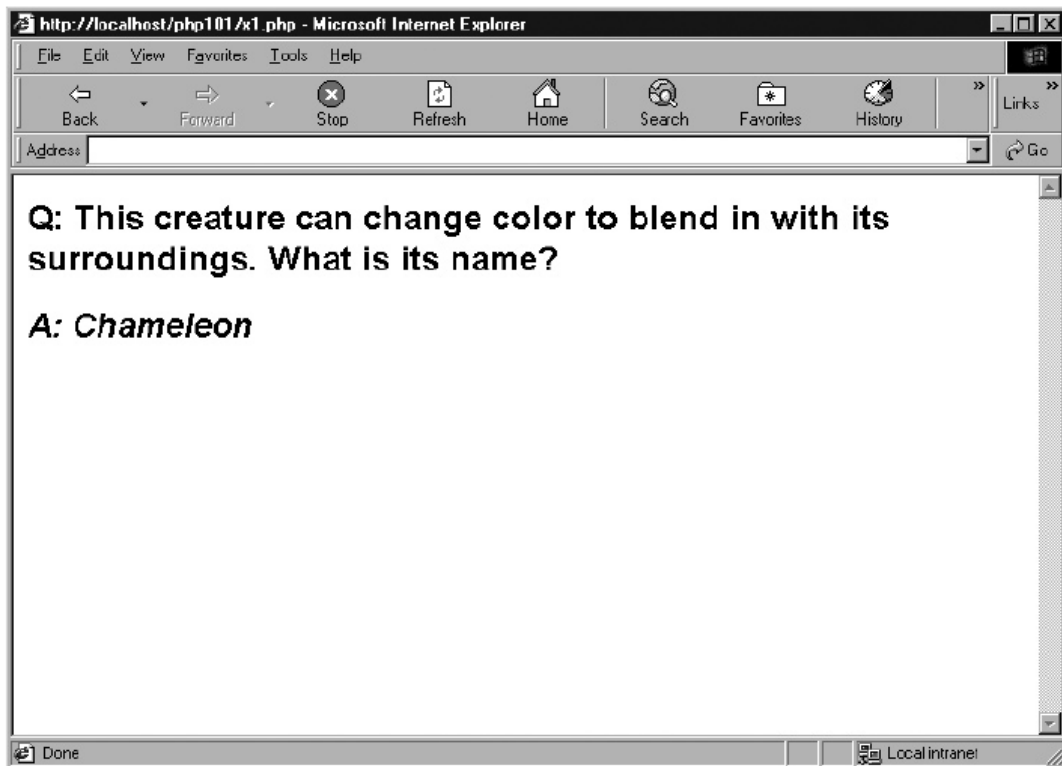
```
// print output
```

```
echo '<h2><i>A: Chameleon</i></h2>';
```

```
?>
```

```
</body>
```

```
</html>
```



## Storing Values in Variables

*Variables* are the building blocks of any programming language. A variable can be thought of as a programming construct used to store both numeric and nonnumeric data. The contents of a variable can be altered during program execution, and variables can be compared and manipulated using operators.

PHP supports a number of different variable types—Booleans, integers, floating point numbers, strings, arrays, objects, resources, and NULLs—and the language can automatically determine variable type by the context in which it is being used. Every variable has a name, which is preceded by a dollar (\$) symbol and it must begin with a letter or underscore character, optionally followed by more letters, numbers, and underscores. For example, \$popeye, \$one\_day, and \$INCOME are all valid PHP variable names, while \$123 and \$48hrs are invalid variable names.

*\*\*\*Variable names in PHP are case-sensitive; \$count is different from \$Count or \$COUNT.*

```
<html>
  <head><basefont face="Arial"></head>
  <body>
    <h2>Q: This creature has tusks made of ivory.
    What is its name?</h2>
    <?php
      // define variable
      $answer = 'Elephant';
      // print output
      echo "<h2><i>$answer</i></h2>";
    ?>
  </body>
</html>
```

## Assigning and Using Variable Values

To assign a value to a variable, use the assignment operator, the equality (=) symbol. This operator assigns a value (the right side of the equation) to a variable (the left side). The value being assigned need not always be fixed; it could also be another variable, an expression, or even an expression involving other variables, as here:

```
<?php
$age = $dob + 15;
?>
```

To use a variable value in your script, simply call the variable by name, and PHP will substitute its value at run time. For example:

```
<?php
$today = "Jan 05 2004";
echo "Today is $today";
?>
```

### **Saving Form Input in Variables**

Forms have always been one of the quickest and easiest ways to add interactivity to your web site. A form enables you to ask customers if they like your products and casual visitors for comments. PHP can simplify the task of processing web based forms substantially, by providing a simple mechanism to read user data submitted through a form into PHP variables. Consider the following sample form:

```
<html>
<head></head>
<body>
<form action="message.php" method="post">
Enter your message: <input type="text" name="msg" size="30">
<input type="submit" value="Send">
</form>
</body>
</html>
```

**The most critical line in this entire page is the <form> tag:**

```
<form method="post" action="message.php">
```

...

```
</form>
```

As you probably already know, the method attribute of the <form> tag specifies the manner in which form data will be submitted (POST), while the action attribute specifies the name of the server-side script (*message.php*) that will process the information entered into the form.

Here is what *message.php* looks like:

```
<?php
// retrieve form data in a variable
$input = $_POST['msg'];
// print it
echo "You said: <i>$input</i>";
?>
```