

▶ **FileMaker Pro and PHP:
Powerful Open Source Tools Bring Your
FileMaker Data to the Web**

Steve Lane

Vice President, The Moyer Group



FileMaker Pro and PHP: Powerful Open Source Tools Bring Your FileMaker Data to the Web

STEVE LANE
VICE PRESIDENT, THE MOYER GROUP

The Moyer Group
www.moyergroup.com
833 West Chicago Avenue, Suite 203
Chicago, IL 60622
(800) 582-0170

Executive Summary

PHP is a powerful, open-source middleware tool for building web applications, especially data-driven web applications. With native connections to over twenty powerful relational database systems, PHP is the tool of choice for thousands of web developers worldwide. This popular, extremely versatile tool can now also be used to bring your FileMaker Pro data to the web. PHP couples an extremely attractive blend of features with very low cost of ownership. This white paper is intended to introduce the FileMaker Pro web development community to PHP, and to explain its features and benefits.

What is PHP?

PHP is an open-source scripting language. It is primarily intended for use in the development of web applications, but it can be used as embedded scripting language in other types of applications as well. This white paper is concerned with its application as a web development tool.

Open source is a term that is often misunderstood. In general, the term means that:

- the source code for a piece of software is made available for public viewing or distributing.
- the software may be freely used, and redistributed under certain restrictions.
- the modification of the software by other developers is encouraged.

Open source software is *not* necessarily in the public domain, though.

In general, open source software is characterized by rapid evolution, a high degree of stability due to the rapidity with which bugs are fixed, and robust support from Internet development communities. Open source is a movement which is strongest in the Unix world, and many open source software projects have a noticeable Unix orientation. Open source has proven itself capable of producing extremely powerful and robust software. The Apache web server, for example, is an open source project that dominates its application niche (web servers).

PHP is an example of the power of open source development. From modest beginnings in the late 90s it has risen to an installed base of over one million servers serving over nine million domains in the most recent Netcraft survey (<http://www.php.net/usage.php>). Its power, versatility and open source licensing have contributed greatly to its popularity.

Not everyone, by the way, agrees that open source software is a good thing! The strongest opposition to open source has come from companies who feel their market share is threatened by competing open source products. Microsoft Corp. has probably been the most vocal opponent of the movement, but smaller firms occasionally speak out against open source as well.¹ Arguments against open source typically take one of the following forms:

- Open source licensing is a kind of virus, since it forces people who use open source code to give away the software that they create with it. This scheme is destructive of the idea of intellectual property, and removes any commercial incentive to create great software.
- Open source software is not backed by any commercial enterprise. This removes any accountability for software defects (you might call this the nobody-to-sue objection).
- By the same token, open source software is not backed up by an organized technical support body, so there is no source of support if things go wrong with the software.

This paper is not an appropriate forum in which to re-air the lengthy debate surrounding open source software. A rich set of documents can be found at the web site of the Open Source Initiative (www.opensource.org). We'll just provide short counterpoints to the above complaints.

- Intellectual property. It is true that many "open source" software licenses forbid a programmer to charge for any derived piece of software that incorporates or uses the open source software. This is not universally true, though, and in any case, what readers of this paper should understand is that *open source licensing does not prohibit you from charging money for software developed using PHP*.
- Lack of accountability. The modern commercial software industry is in fact remarkable for its lack of accountability. Did software from BigDatabaseCo eat some of your data? Too bad. Their license almost certainly disclaims any liability for any damage or loss this may have caused you. It probably also disavows that its software is in fact, fit for use as a database. No kidding. Open source software is no worse than commercial software in this regard.²
- Lack of technical support. There's actually no lack of technical support at all for open source software, but it takes a different form from commercial tech support (it's free, for one thing). The most popular open source software, since it is used by thousands of users and developers, enjoys very deep support in the Internet community. Typically this support is found on mailing lists and newsgroups, which are frequented by very knowledgeable users of the software, and often by the software's principal developers as well. It is not unusual, in posting a question to a mailing list or newsgroup that deals with an open source application, to receive a reply from one of the software's authors. Needless to say, their replies tend to be highly accurate. If you present evidence of an actual bug or flaw in the software, the authors are usually very diligent in working with you to explore and solve the problem. The principal advantage of open source tech support is that it tends to come from extremely knowledgeable people, if not the software authors themselves, as opposed to hired tech support staff who may still be taking training in the software they are supporting. The principal drawback of open source tech

support is that it is email-based, and may require four to twenty-four hours to produce a sufficiently detailed response to a question.

Please see the Resources section at the end of this paper for references to other discussions of open source software.

Installing and Using PHP

PHP is freely downloadable from the project site at www.php.net. It is usually distributed in the form of source code, but can be downloaded in precompiled ("binary") form for a variety of platforms, including Windows and Mac OS X. PHP can be built with support for dozens of different modules and additional libraries, and for this reason many developers prefer to compile it themselves from source code. For those lacking the technical expertise or the inclination to build PHP themselves, the available binaries are usually configured with support for many of the most popular add-ins. A basic configuration of PHP also comes pre-installed with Mac OS X, as does an appropriately-configured copy of the Apache web server. As a result, OS X is a great open source web development platform right out of the box.

PHP is configured as an extension to an existing web server. The most popular choices of web servers for this purpose are Apache (on Unix platforms, including Mac OS X) and Internet Information Server (on platforms based on Microsoft technology), though other web servers such as Webstar (for Macintosh) and the Zeus server are also supported. Regardless of platform, PHP is installed in such a way that the host web server will hand off requests for certain types of pages to the PHP processor. Those pages contain PHP code which the PHP processor will execute before returning an HTML page to the web server to be sent to the client. In this way PHP is no different from other widely popular middleware tools like JSP, Perl and ASP.

PHP, again, derives from a Unix-centered software world. Like most open-source software, it does not come in a box with a printed manual, nor does it have a point-and-click graphical installer. Installation on most platforms involves placing the PHP extension and associated libraries into certain specified directories. For operating systems based on Unix, some familiarity with command-line interaction with the machine is helpful. PHP, like many pieces of server-side software, is administered by editing a configuration file using a text editor, and by issuing commands through a command-line interface.

Though no printed documentation is available for PHP, the distributions come with copious, platform-specific electronic documentation that provides detailed installation instructions. Additionally, there are numerous excellent third-party books on PHP (references are provided at the end of this paper). As with the installation of any middleware, you will likely need the cooperation of your web server administrator during the installation process.

PHP Features

At the core of PHP is a lightweight, interpreted scripting language with an economical syntax that draws heavily on the family of syntaxes that are descended from the C programming language. Anyone with a little programming background can readily pick

up the basics of PHP. It is also a fine choice as a first language for those just learning the basics of procedural programming. PHP is a procedural language, with full support for user-defined functions and relatively weak enforcement of data types (in this way it's similar to other scripting languages like Javascript). PHP has all of the syntax elements and flow-of-control structures one would expect of a modern programming language (rich set of built-in operators, *for* loops, *while* loops, *switch* statements and the like). In addition, PHP can also be used in an object-oriented (OO) fashion, to define traditional OO classes and methods, for those who are used to working in an OO programming style.

PHP's core is a concise lightweight language with object-oriented features, but the tool's greatest selling point are its massive function libraries. PHP functions fall into two categories: those are built directly into the core language and are always available, and those that need to be specifically enabled when the software is built. The functions in the second category are usually those that link to other external code libraries that are not themselves a part of PHP. PHP's ability to be linked to hundreds of other widely-available libraries of code is one of the tool's principal strengths.

PHP's functions fall into the following rough categories:

Compression and Encryption: PHP provides a rich set of functions for compressing and decompressing files and data streams, as well as support for some of the more popular modern encryption schemes.

Database Access: PHP can provide native access to some twenty-five popular relational database management systems. Access to these system is accomplished by hooking PHP directly into the native client libraries for these databases, rather than accessing them through slower intervening software such as an ODBC or JDBC driver. Currently PHP offers direct native access to the following database systems: Adabas, SAP-DB, Solid, IBM DB2, Empress, Velocis, DB++, dBase, a variety of dbm-based packages, filePro (different from FileMaker Pro!), Frontbase, Hyperwave, Informix, Ingres II, Interbase, Lotus Notes, mSQL, MySQL, Microsoft SQL Server, Oracle, Ovrimos, PostgresQL, SESAM SQL, and Sybase. Additionally, PHP can access any ODBC-aware database through ODBC drivers. Note that PHP currently contains no native direct access to FileMaker Pro. PHP access to FileMaker Pro, as we will see, is accomplished by using PHP's HTTP and XML functions.

Date and Time: PHP has a full set of functions for manipulating dates and times, including the ability to reckon time to the millisecond, to work with any time zone in the world, and to translate among different calendar systems.

E-commerce: PHP supports easy credit card verification, and can interface with electronic payment systems like Cybercash, CyberMUT, and Verisign's Payflow Pro.

Filesystem functions: PHP can interact directly with the server's local file system, allowing a wide variety of file manipulation tasks to be handled on the server.

Graphics: PHP can be linked to a variety of powerful graphics libraries. The GD library permits on-the-fly creation of images in a wide variety of popular formats (jpeg, gif, png etc.), with full support for TrueType fonts. Other libraries permit on-the-fly creation of Shockwave animations. A PHP page can, for example, easily draw data from two

different data sources (say, Oracle and FileMaker Pro) and combine the results into a dynamically-generated Flash animation.

Other Languages: PHP can be built with support for direct calls into code written in the Java language. In addition, PHP's ability to send commands to its native operating system (see below under Program Control) mean that its ability to interact with other programs written in other languages is effectively unlimited. (We have written a number of applications, for example, in which PHP calls on custom programs written in the Python language to generate complicated PDF documents that are then viewable in the browser). Note that this support for interprocess communication is stronger on Unix-based platforms, including Mac OS X, than on Windows.

Internet Protocols: PHP can communicate with a very wide variety of Internet-standard services. This includes the ability to send email via SMTP; to receive email by communicating with both POP and IMAP servers; to send and receive files via HTTP and FTP; to communicate with network news servers via NNTP; and to communicate with chat (IRC), calendar (MCAL, LDAP and NIS servers).

PDF Support: PHP can be linked to either of two popular libraries for generating PDFs on the fly, as well to a library that includes support for PDF-based forms (FDF).

Program Execution and Process Control: PHP has a rich library of commands for triggering other processes and commands on a Unix-based operating system. This allows PHP to call on other programs, to provide still further functionality beyond what even the extensive module libraries allow. One popular application, for example, is to have PHP send commands to the GIMP imaging software, to allow for the dynamic generation of differently-sized thumbnail images in response to user selections.

Shared Memory: PHP is able to access Unix shared memory, to provide "application variables" in a persistent data store that does not change from page to page and is global to all current users. This feature is not available on Windows.

Regular Expressions: PHP provides full support for powerful regular expression libraries, including both POSIX- and Perl-compatible regular expressions.

Session Handling: PHP has a powerful set of built-in functions for handling persistent user data ("sessions"). Session data, by default, is written to a server-side file, but PHP also provides hooks for users to create sessions that are stored in a database instead. PHP also includes support for Mohawk msession, a high-speed session server for multi-CPU web server farms.

String Handling: PHP provides a very rich library of string-handling functions. In addition to PHP's nearly 80 built-in string handling functions, and the two forms of regular expression handling mentioned above, PHP also can support multi-byte strings and spell checking.

Windows Functions: Although PHP often betrays its Unix heritage, especially in the areas of process control and shared memory, which are not well supported on Windows, PHP can also access a number of libraries of functions which are specific to Windows only. These include the ability to interact with COM objects, preliminary support for .NET interfaces, and the ability to send commands directly to Windows printers.

XML Features: PHP has very full interoperability with XML. PHP can parse XML documents using either the expat library or a set of interfaces based on the W3C DOM. It contains initial support for XML-RPC, and can also perform transformations of XML data using XSL.

Coding in PHP – General Principles

In a typical web application, a PHP programmer writes pages that are a mix of PHP code and regular HTML code. Here's an example of a page that would display the current time:

```
<html>
<head>
<title>PHP Time Demonstration</title>
</head>
<body>
    Hello! The current time is <? echo time();?>
</body>
</html>
```

Most of the page is HTML. The only actual PHP code is contained inside the `<?...?>` tags. These tags are a signal to the PHP processor that what's contained inside them should be interpreted as PHP code. In this case, the PHP instructions simply "echo" the current server date and time, so that they are inserted into the HTML document at that point.

PHP, as mentioned above, provides all of the features that one would expect in a modern procedural language. Variables are distinguished by a dollar-sign prefix, and data typing is fairly fluid. Consider this short PHP program:

```
<html>
<head>
<title>PHP Variable Demonstration</title>
</head>
<body>
    <?
        $myName = "Jane Coder";
        for ($i = 1; $i <=5; $i++) {
            echo ( "$myName loop count = $i<br />");
        }
    ?>
</body>
</html>
```

This will produce a web page that looks like this:

```
Jane Coder loop count = 1
Jane Coder loop count = 2
Jane Coder loop count = 3
Jane Coder loop count = 4
```

Jane Coder loop count = 5

The PHP code here contains an ordinary *for* loop (the syntax here is very similar to C, C++ and Java). Inside the loop, the value of the \$myName variable is repeatedly printed, along with the current value of the loop counter (\$i). Notice that the loop counter is able to act as a numeric value in the for loop, and as a component of a text-string value for purposes of output, without any explicit translation between the two data types. This kind of convenience is typical of weakly-typed languages in general.

PHP's syntax is extremely concise and economical.³ But for problems that may require lengthier units of code, PHP allows the programmer to package complex operations into functions, which aid in both abstraction and code reuse.

Functions in PHP

One of the most important features of PHP is that, like most modern programming languages, it allows the user to define functions – self-contained blocks of code that can be used over and over again. Function definition is essential to creating a modular, extensible software application.

For example: suppose you have a small piece of code that takes an email address and checks to see whether it is formally valid (that is, it checks the syntax of the address but does not actually contact a mail server to verify that it is a real address). You could write the following function to do this:⁴

```
<?
function is_valid_email($email){
    if(ereg("([:alnum:]\.\.-]+)(\@[[:alnum:]\.\.-]+\.)", $email)){
        return 1;
    }else{
        return 0;
    }
}
?>
```

The function is very simple. It just applies a regular expression to the email address to see whether it violates the commonly-accepted email address formats. This would still be cumbersome to write each time we wanted to do email validation. But now that the function is defined, it can be reused very easily:

```
<? if (is_valid_email( $formEmail ) ) {
    sendMailAnd Close();
} else {
    echo ("That does not appear to be a valid email address. Please
try again");
}
} ?>
```

The function can now be called with a single line of code. This is convenient enough as it is, but now imagine that the *is_valid_email* function also attempted to check for the existence of the specified mail domain, check that its MX records were valid, and whether the specified address was a legitimate address for that mail domain. The function to do this might take almost a page of PHP code to write, but once it was written, it could be reused at will, and copied out for reuse in other applications.

Object-Oriented Features of PHP

Object-oriented (OO) programming is an important concept in modern software design, though it's an area of programming that many FileMaker Pro developers may have had little exposure to. Properly applied, in a language with full support for OO features, OO programming makes it much easier to control the complexity of large applications (by which we mean gigantic applications, with hundreds of database tables and thousands of users, or applications with very deep and complex business rules). Even for small and medium-size applications, the benefits of OO can still be considerable as a simplifying tool.

PHP has fairly full support for defining and using classes and objects (the building blocks of OO programming). Classes can inherit from other classes, and PHP provides support for polymorphic function calls. Developers familiar with OO programming will find that PHP makes it easy to define and use classes and objects.

As an example of how to use objects to control complexity, let's consider a banking application. We're working on a portion of the application that manages user accounts. All accounts are stored in a single database table with the following fields:

```
account_id  
account_type  
account_balance  
account_owner_id  
debits_today
```

The system recognizes three different types of accounts: savings, checking and IRA accounts. The account types all share some common features. For example, they are all written to and read from the underlying database the same way. None of them is allowed to have a balance of less than zero. But they have differences as well. For example, the daily withdrawal limit for a checking account is \$500, but for savings it's \$1000. Checking accounts have a minimum balance, below which every transaction incurs a service charge. IRA withdrawals can't be made unless the user confirms they understand the tax penalties. And so forth.

Let's think about the features we need to have. We need to be able to read and write any account to and from an underlying database. We need to be able to make deposits and withdrawals against any type of account. How can we program this? Well, we need some kind of data structure to hold the information about the account; and we need code that will perform each of the functions listed above.

We could build this by using a PHP array to hold the account information, and writing a number of functions to perform the different operations we just listed. The functions might look like this:

```
readAccount ($accountData)
writeAccount($accountData)
deposit( $accountData, $amount )
withdraw( $accountData, $amount)
```

Then, within the deposit() and withdraw() functions, we could have conditional code that says, in effect, "if the account is checking, do this; if it's savings, do this instead; but if it's an IRA, don't do either of those, do this". This amounts to writing several different pieces of functionality within a single PHP function. If that's not to our liking, we could instead write multiple versions of each function:

```
depositChecking( $accountData, $amount)
depositSavings( $accountData, $amount)
withdrawChecking( $accountData, $amount)
withdrawSavings( $accountData, $amount)
```

Each function will now contain exactly the right logic for the account type, but this doesn't solve the problem, it simply moves it. Now, in order to know which function to call for a given account, we first have to inspect the account, figure out what type it is, then say "if it's checking, call withdrawChecking(), else if it's savings, call "withdrawSavings()", just the same as before.

Objects make this kind of dilemma easy to solve. First we'll define a base class to handle all the behavior that's shared among the different account types:

```
class Account {

    $var account_id;
    $var account_balance;
    $var account_owner_id;
    $var debits_today;

    $var max_daily_withdrawal;

    function Account( $account_id ) {
        $this->account_id = $account_id;
    }

    function read() {
        [This function would contain code to read the record from a database]
    }

    function write() {
        [This function would contain code to write the record to a database]
    }
}
```

```
function deposit( $amount ) {  
    $this->account_balance += $amount;  
}  
}
```

This class is able to handle reading and writing the accounts to and from a database. This behavior is the same regardless of account type. Notice that we define the behavior for deposits here as well, since deposits work the same way across all account types. Now we need a couple of subclasses to define type-specific behavior:

```
class CheckingAccount extends Account {  
  
    $var minimum_balance;  
  
    function CheckingAccount( $account_id ) {  
        $this->Account($account_id);  
        $this->minimumBalance = 2500;  
        $this->max_daily_withdrawal = 500  
    }  
  
    function withdraw( $amount ) {  
        if ( $amount + $this->debitsToday > $this->max_daily_withdrawal ) {  
            return "You have exceeded your daily withdrawal limit";  
        } else {  
            $this->account_balance -= $amount;  
            if ( $this->account_balance < $this->minimum_balance ) {  
                $this->account_balance -= 3; // service charge below min balance  
            }  
            $this->write();  
            return($amount);  
        }  
    }  
}
```

```
class SavingsAccount extends Account {  
  
    function SavingsAccount( $account_id ) {  
        $this->Account($account_id);  
        $this->minimumBalance = 5000;  
        $this->max_daily_withdrawal = 1000  
    }  
  
    function withdraw( $amount ) {  
        if ( $amount + $this->debitsToday > $this->max_daily_withdrawal ) {  
            return "You have exceeded your daily withdrawal limit";  
        } else {  
            $this->account_balance -= $amount;  
            $this->write();  
            return($amount);  
        }  
    }  
}
```

Having the code divided into classes makes maintaining the application much easier. If you need to change the way accounts are stored in the database, edit the base Account class. If you need to change the rules for savings or checking accounts, edit the appropriate subclass. And if you need a new account type altogether, create a new subclass of Account and add the functionality you need. Say for example you needed a new kind of checking account in which all deposits over a certain amount were reported to the IRS. You could write a new class that extends CheckingAccount, and simply provide a new implementation of the deposit() function, which is the only functionality you would need to change.

PHP's classes are a powerful organizational tool, but they should be used with some caution. Because the class definition must be fully loaded with each new web page, using a dense class hierarchy with many classes included in a single page, or with an inheritance tree that is more than a couple of levels deep, can cause your pages to load more slowly. If your application is complex enough that you need a full OO technology, you might be better off considering Java Server Pages (JSPs). JSP pages can communicate with FileMaker Pro via FileMaker's JDBC driver. JSP programming can draw on the full power of the Java language, which, in addition to its gigantic class and function libraries (many times larger than even those of PHP), is a fully object-oriented language designed to support complex inheritance trees and extensive design abstraction.⁵

Objects in PHP are best seen as a very useful organizational tool that can make your code easier to read and maintain. The upcoming 4.3 release of PHP will add many improvements to PHP's OO capabilities as well.

Database Access in PHP

PHP has direct support for over twenty relational database management systems. PHP has support for the popular open source database MySQL built in and enabled by default, though it can be turned off if desired. Support for other databases must be enabled when PHP is compiled, and generally requires the presence of client libraries for the specific database to be available at the time the software is being built.

Once support for a database has been built into PHP, the functions for accessing that database are then available to the programmer. Each database has its own set of functions, and, though there is substantial consistency among the features PHP lets you access for each database type, there are variations as well. Let's take a look at some code that works with PostgreSQL, a popular open source database system (www.postgresql.org). The following code opens a connection to a Postgres database server and counts the number of customers in a customer database:

```
$databaseHandle = pg_connect("host=bigben.fmpro.com port=5432 dbname=crm
user=lamb password=bar");
if ( !$databaseHandle ) {
    echo "Unable to connect to database!";
    exit();
}
```

```

$query = "SELECT count(*) as customer_count from customer";
if ( ! $result = pg_query( $databaseHandle, $query ) ) {
    $error = pg_error( $databaseHandle );
    echo "There was an error accessing the database: $error";
    exit();
}
$dataRow = pg_fetch_array( $result, 0 );
$customerCount = $dataRow['customer_count'];

```

The code snippet above uses four Postgres-specific functions (other databases will have analogous functions with different names). First, we try to make a connection to a Postgres database using `pg_connect`. We tell the function the name of the database host, the port the database is running on, the name of the database to connect to, a user name, and a password. If `pg_connect` returns a null value, the connection attempt has failed, so we return an error and quit the script. Otherwise, we create an SQL query string that queries the customer database for the total number of records it contains. We then use `pg_query` to run that query over the current database connection.

Once again we need to check for failure. If `pg_query` returns a null value, once again we give an error message and quit the script. This time, though, we know we at least got a good connection to the database, so we can use the `pg_error` function to ask the database what went wrong, and pass the information back to the user before exiting.

Finally, if everything worked out all right and our query executed successfully, we can use the `pg_fetch_array` function to pull the results of our query into a PHP array, and then extract the `customer_count` field from that array. We can now do something with that value, whether that means displaying it in an HTML page or using it in some other calculation.

Relational databases products from different vendors have differing capabilities, and the PHP function libraries reflect this. Postgres, for example, is able to store binary files directly in the database (much like FileMaker Pro's ability to store movies and pictures in a container field, but extended to any kind of binary file), and PHP gives functions to access that capability. In general, each database's function set is slightly different.

When writing web applications that talk to databases, it's often useful to be able to use a consistent interface to access several different types of database. This capability is referred to as a *database abstraction layer*. The virtue of a database abstraction layer is twofold: in the first place it simplifies the programmer's learning task, since the programmer need not learn multiple different function sets for different databases, and in the second place it makes it much easier to switch your application from one database to another if need be.

PHP has several answers to the problem of database abstraction. Some of these are capable of being built directly into PHP. For example, the Unified ODBC functions can be used to access Adabas, IBM DB2, Sybase SQL Anywhere, and Solid. There is also an abstraction layer for databases in the Berkeley family of dbm databases. Probably the best basis for a database abstraction layer in PHP, though, lies in the code in the PEAR library that is distributed with PHP. PEAR is a rich library of PHP code, generally object-oriented, with classes that add functionality in a number of areas. PEAR contains a

database abstraction layer that covers about ten of the most popular databases in a single interface, and is an excellent choice for those who want to program to a single database interface.⁶

Session Control with PHP

HTTP, the underlying protocol of the Web, is often referred to as a "stateless" protocol. In a typical client-server environment, the server remains aware of the identity of each client that's connected to it, from one client request to the next. The web works differently: given two requests from a web browser, for two different pages on a web site, the web server has no built-in means to know that it was the same browser requesting both pages. But this is an important capability. The classic example is a shopping cart application: as you shop on an ecommerce site, and add items to a cart, the web server must make sure always to return you to the contents of your cart, and not someone else's.

A good FileMaker Pro analogy is provided by FileMaker Pro's global fields. Global fields are data associated with a single user, that stay the same as the user moves from one screen to another. This ability to have a storehouse of data, associated with a single user, that persists in time no matter the user does or where she goes, is precisely the ability that's lacking in plain HTTP. In order to do this, the web server needs some way to identify who a particular request is coming from, and then a way to associate that with some data stored somewhere on the server. This mechanism is commonly referred to as a session.

Most middleware products provide support for sessions. In general, a middleware tool needs the two features mentioned above: a way to identify a request with a session, and a way to store data associated with the session. Identifying which session a user belongs to is usually done by communicating a *session key* to the web server. Most middleware by preference will store the session key in a cookie. If a user disables cookies, though, a good middleware product will revert to a method that communicates the session key inside each URL or form posting inside a site, so that the key comes back to the server with each new request. In general, well-structured middleware stores only the key this way, *not* any of the potentially sensitive user data associated with the session. This user data is usually stored somewhere else on the web server: possibly in a simple flat file, possibly in a database.

Session support is built directly into PHP. By default, PHP will store its session keys in a client-side cookie, and store session data in a flat file on the server. If cookies are disabled, and PHP is configured correctly, the session key will be propagated in URLs and HTML forms instead. All such propagation is completely transparent to the programmer and requires no additional coding.

By default PHP will store the actual session data in a disk file on the server. This behavior is easy to override, however. PHP's session handler functions can be rewritten to store the sessions in a database of the programmer's choice. This is useful if you want to log user access to the site.

Sessions in PHP are extremely easy to use. Once PHP is configured to track and store sessions, the programmer only needs to call the `session_start()` function at the top of each page that needs to be session-aware. The `session_register()` function can be used to add

variables to a session – these variables then are automatically available on all subsequent pages that are session-aware. In that respect, session variables behave exactly like FileMaker's global variables. (They're a bit better, actually, because they're not tied to a particular database table).

What About Connecting to FileMaker Pro?

As noted above, FileMaker Pro support is not built directly into PHP. To understand this, we need to compare FileMaker's client-server architecture to that of some of the other database management systems supported by PHP.

Many database systems (indeed, *most* modern database systems) don't tightly couple the client and the server. In general, such database servers are written so that they can be accessed in a variety of different ways – via the web, via a front end written in Java or Python or Visual Basic, via an ODBC-based report generator, etc. In order to allow this, database servers often are accessible through a variety of different interfaces. Often a server will provide interfaces that can be accessed through a variety of programming languages (C, C++ and Java are probably the most common), as well as through a variety of higher-level interfaces (ODBC and JDBC probably being the most common). The server makes these different interfaces available to programmers, and doesn't worry much about what kinds of applications they will write that use these interfaces.

FileMaker Server, though, is a bit different. The server proper only provides a single interface, which is the interface to the FileMaker Pro client software. This is also a proprietary interface, so in fact there is currently only one piece of software that can communicate with FileMaker server, and that's the FileMaker Pro client software itself. Anyone wanting to get data from FileMaker files needs to somehow do so through the FileMaker client. The FileMaker Pro client, in turn, offers a number of interfaces to other applications that might want to share data with FileMaker. These interfaces include simple import/export, AppleScript or Visual Basic connectivity, and the FileMaker Web Companion. Of these, the Web Companion is the most flexible, and in some sense the most standards-based: it communicates over HTTP, and can broadcast data in the form of XML. Many other applications that query FileMaker Pro to extract data from it ultimately come through the Web Companion.⁷ This turns out to be true for PHP as well.

Access to FileMaker Pro is accomplished by using PHP's HTTP functions to send commands to the FileMaker Web Companion, and PHP's XML functions to parse the resulting data. Especially with the new XML features in FileMaker Pro 6, XML is now positioned as the standard format in which FileMaker Pro will deliver its data to other applications. Or, to put it differently, FileMaker Pro's best external data access API consists of XML data served up by the Web Companion.

These abilities are a good fit for PHP. PHP can pull data from any web-based data source by just making an HTTP call. And PHP can also parse any XML-based data using PHP's XML libraries. This is all good so far, but the best news is that a PHP library already exists that automates this entire process. The library is called FX.php: it's written and maintained by Chris Hansen and is available at <http://www.iviking.org>. Chris' FX library automates the entire process of querying FileMaker Pro for XML data, getting the data back over HTTP, and parsing the data into some usable form. This integration of PHP

and FileMaker Pro is a great example of the power of open source tools: FX.php is freely available for download and use. Thanks,Chris!

This all sounds good, but how do we use it? Well, to start with, you need to download the files and put them in a place where they can be easily referenced. You then include the main FX.php file in your own code, and start using its features. Here's an example, boiled down from the samples included with FX. The following code runs a search against a FileMaker Pro database called Book_List.fp5, and returns any results as a formatted HTML table:

```
<?
include_once("FX.php");

$serverIP = "12.248.71.14";

$BookQuery = new FX($serverIP);
$BookQuery->SetDBData("Book_List.fp5", "Book_List");

if ($currentSort != '') {
    $BookQuery->AddSortParam($currentSort);
}
if ($currentQuery == 'Search Book List!') {
    $arrayName = 'HTTP_' . $HTTP_SERVER_VARS["REQUEST_METHOD"] . '_VARS';

    foreach ($$arrayName as $key => $value) {

        if ($key != 'currentSort') {
            $BookQuery->AddDBParam($key, $value);
        }
        $currentSearch .= '&' . "$key=" . urlencode($value);
    }
    $BookData = $BookQuery->FMFind();
}
else {
    $BookData = $BookQuery->FMFindAll();
}

$counter = 1;
if ($BookData['foundCount'] > 0) {
    foreach ($BookData['data'] as $key => $value) {
        $recordID = strtok($key, '.');

        if ($counter % 2 == 0) {
            echo "<tr bgColor=#CCCCCC>\n";
        }
        else {
            echo "<tr>\n";
        }
    }
}
```

```

    echo "<td align=\"left\" valign=\"top\">";
    echo $value['author'][0];
    echo "</td>\n";

    echo "<td align=\"left\" valign=\"top\">";
    echo "<a href=\"detail.php?ID=$recordID&query=" .
urlencode($currentSearch) . "\">";
    echo $value['title'][0];
    echo "</a></td>\n";
    echo "<td align=\"right\" valign=\"top\">";
    echo $value['number_of_pages'][0];
    echo "&nbsp;&nbsp;&nbsp;</td>\n";
    echo "</tr>\n";
    ++$counter;
  }
}
else {
}
?>

```

What does all this code mean? Well, the first thing to know is that the FX code uses PHP's object-oriented extensions. FX is coded as a class, and all of its functionality is accessed through that class' methods (functions).

Using FX in a page generally follows a fairly simple pattern. First you create a new instance of the FX class and point it to a particular instance of FileMaker Unlimited. Then you set up all necessary query parameters, and execute the query. Finally, you do something with the data you got back.

For example, the code above creates a new instance of the FX class and points it at the server at 12.248.71.14. Next, the FX object is instructed to look at the database Books.fp5, using the Book_List layout. From there, additional query parameters are established.

For example, if the variable \$sortOrder is set, its contents are added to the query as a sort field using the AddSortParam() function. The code then tries to decide what kind of search to perform. If a search is being performed on fields the user specified, the above code will loop through all of those user-specified fields and add each of them to the current search using the AddDBParam() function. (The way this is accomplished is a little arcane if you're not used to PHP – the code decides whether the search page was accessed through a form submission or through a URL, and extracts all of the parameters that were specified in the form or URL and treats them as search parameters).

At this point, if a user-specified search is in process, the code will use the FMFind() function. Otherwise it will simply look for all records using the FMFindAll() command.

The last step is to actually do something with the returned data. FX database result sets are very verbose, meaning that a lot of useful information comes back, structured as a multi-layer array. Getting to the actual data you want can seem a little involved at first, but it's simple once you're used to it. The result set contains information about the found

count for the query, the names of the fields returned, the actual field data, the Web Companion URL that was used to issue the request, any error code, and also information about any value lists that were in use on the queried layout (this lets you pull value lists from FileMaker Pro and use them on the web).

That's more or less all there is to using the FX class. It's pleasingly simple: point FX to a FileMaker Web Companion, format a request, submit the request, look at the result (if any). One final note: since FX is ultimately driving the Web Companion, it's helpful to be familiar with the Web Companion's available command set. (The Web Companion's capabilities are documented as part of the FileMaker Developer Edition software).

Other Useful Tricks

One other useful technique in combining PHP with FileMaker Pro, if you're using sessions on your web site, is to write a custom handler to save your session data to a FileMaker Pro database. To recap what we said above about sessions, by default PHP will simply take all the data associated with a session and write it to a file somewhere on the server. But what if you want each new session to become a database entry? Perhaps you want to run reports on session activity on your site, or otherwise maintain a permanent record of your sessions. The usual way to do this is to write what's called a session handler. PHP provides six functions that control session management. If you, the programmer, redefine these functions, it's easy to have PHP save your sessions to a database rather than to a disk file. Session handlers already exist for popular open source databases such as PostgreSQL and MySQL. Writing your own session handler for FileMaker Pro is not terribly difficult, but happily, you don't have to. An open source FileMaker Pro session handler is available for download at <http://www.moyergroup.com/opensource>.

Summary

PHP is one of the most popular and powerful web scripting languages currently in use. It combines a clear and simple syntax with a huge number of high-quality code libraries and support for dozens of databases. As an open source product, it is both free of charge, and supported by an extremely knowledgeable Internet community. Thanks to the power of the open source process, it is now extremely easy to use PHP to access your FileMaker Pro data to drive dynamic web sites. Whether you're a FileMaker developer just getting started with web programming, or a seasoned web programmer in a search of a new tool, PHP can be an extremely important tool in your arsenal.

References

Web Sites

<http://www.php.net>

The official PHP web site. Code, documentation, tutorials, and a whole range of useful resources.

<http://www.phpbuilder.com>

Many useful in-depth technical articles on PHP.

<http://www.iviking.org>

Official site for Chris Hansen's FX class for PHP -> FileMaker Pro connectivity.

<http://www.moyergroup.com/opensource>

Freely downloadable PHP/FileMaker session handler.

<http://www.entropy.ch/software/macosx/>

Marc Liyanage's collection of downloadable software for Mac OS X. Includes precompiled software for PHP (Apache module), and the popular open source databases PostgreSQL and MySQL.

<http://www.opensource.org>

Great collection of references and resources on the open source movement.

Books

Professional PHP 4. Luis Argerich et al. Wrox Press. 1861006918.

Programming PHP. Rasmus Lerdorf (the principal author of PHP). O'Reilly and Associates. 1565926102

Professional PHP4 XML. Luis Argerich et al. Wrox Press. 1861007213

Newsgroups

The definitive newsgroup is now comp.lang.php.

¹ See, for example, a document made available on the web site of BlueWorld Communications, Inc., titled "Lasso Professional 5 vs. PHP 4.1.x", by Duncan Cameron and Lee McNeil, which states that open source software (in particular PHP) suffers from a lack of "commercial legitimacy". (Lasso is a commercial middleware product used to web-enable FileMaker Pro, 4D, and SQL-based databases.)

² Here's the relevant text from the license for one of the world's foremost commercial database products:

[Name of Vendor] DOES NOT GUARANTEE THAT THE PROGRAMS WILL PERFORM ERROR-FREE OR UNINTERRUPTED OR THAT [Name of Vendor] WILL CORRECT ALL PROGRAM ERRORS. TO THE EXTENT PERMITTED BY LAW, THESE WARRANTIES ARE EXCLUSIVE AND THERE ARE NO OTHER EXPRESS OR IMPLIED WARRANTIES OR CONDITIONS, INCLUDING WARRANTIES OR CONDITIONS OF MERCHANTABILITY, SATISFACTORY QUALITY, AND FITNESS FOR A PARTICULAR PURPOSE.

The enterprise version of this software costs about \$40,000 per CPU. Money spent on commercial software may buy many things, but accountability, at least in the legal sense, is not one of them: so in this particular respect, open source software does not really suffer by comparison.

³ The press release for the Cameron/McNeill white paper suggests that PHP is a relatively verbose language, a claim the white paper attempts to support by comparing the coding of different tasks in both PHP and Lasso. In general the examples do not offer much support for this contention. Regular coding tasks, in fact, frequently require more keystrokes in Lasso. Consider the following simple code snippets, which both check whether a variable called "count" has been set, initialize it to zero if not, and increment it if so.

PHP:

```
if ( !isset($count)) {  
    $count = 0;  
} else {  
    $count++;  
}
```

(44 characters excluding spaces)

Lasso:

```
[If: ! (Variable_Defined:'count') ]  
    [Var: 'count' = 0]  
[Else]  
    [Var:'count' += 1]  
[/If]
```

(74 characters excluding spaces)

The Lasso example can be made slightly more concise, at the expense of consistency in the variable references. But the PHP example can in fact be made much more concise, as follows:

```
$count += isset($count)?$1:0; // 27 characters excluding spaces
```

⁴ This function was posted to the code libraries on codewalker.com by Matt (no last name given).

⁵ Though many middleware tools might claim to be "object-oriented", that claim should be regarded as generally suspect. Middleware based around Java can obviously make that claim. PHP also does a fairly creditable job of implementing two of the three pillars of an OO language (the three are inheritance, encapsulation and polymorphism – PHP does not really have much support for encapsulation), Likewise, Lasso's Custom Types provide some support for inheritance and polymorphism, but no apparent support for encapsulation. These tools, as well as others such as Javascript, could be called "object-aware", but certainly fall far short of what would be considered a genuine OO implementation. Again, if your application's needs are such that you need the power of a true OO language, Java Server Pages may be your best bet. PHP's object support is scheduled for significant further improvements in the upcoming 4.3 release.

⁶ We are puzzled by the statement in the Cameron/McNeil white paper (p. 11) that PHP "totally lacks" a database abstraction layer. It is true that there is no single abstraction layer that covers all twenty-five or so databases that PHP supports, and indeed this would be a considerable engineering feat to create – PHP instead has several abstraction layers, each supporting a different group of databases.

⁷ This is even true in areas where you might not expect it to be so, such as with the FileMaker JDBC driver. The JDBC driver actually turns all of the requested database operations into instructions that it then passes to the Web Companion.