

# Experiences of Using PHP in Large Websites

Aaron Crane

aaron.crane@gbdirect.co.uk

GBdirect • <http://www.gbdirect.co.uk/>

## 1 Introduction

The PHP scripting language has enjoyed an enormous growth in popularity over the past few years. It benefits from being particularly easy to pick up, and from having been designed as a language specifically for producing webpages. This means that choosing PHP as your implementation language allows you to build a dynamically-generated webpage quickly and easily.

However, it is not clear how well PHP scales for use in larger commercial websites. This paper examines the issues in trying to do so. Issues covered include:

- Separation of presentation from business logic
- Using a team of developers to build a site
- Problems that are hard to find in development, but can readily trip you up in deployed sites
- Areas where PHP's initial simplicity can actually make things more complicated

We concentrate on the last of these, and on the irony that a language which hopes to distinguish itself by its simplicity is in many cases inappropriate because of its complexity.

The conclusion arrived at is that, in some circumstances at least, PHP's tendency to create more problems than it solves makes it an inappropriate choice. However, we also recognise that there are some situations in which PHP *is* to be used. For those who find themselves in such situations, we use the experience gained from using PHP in a range of sites (including the geographic search engine Somewherenear.com <http://somerwherenear.com/> and a multi-million-dollar e-commerce site in the canning industry) to offer some guidance on how best to deal with PHP's deficiencies.

Finally, we identify a checklist of pertinent questions for determining PHP's suitability for a given project.

## 2 Separation of Presentation from Business Logic

Modern websites are expected to be able to react quickly to changing demands. For the developers, this translates into a need to be able to change the content and the look of a website with a minimum of effort. It has long been recognised that the most effective way of meeting this need is to enforce a separation of presentation (the purely æsthetic look of a site) from business logic (what the site actually does).

A separation of this sort also allows websites to be built by two teams of developers: one responsible for the business logic, and another responsible for the design. Since few capable programmers are equally capable when it comes to visual design, this is a significant advantage.

Unfortunately, PHP, far from promoting such a separation, actually promotes the inverse. The whole point of PHP is to combine the HTML that forms the site's look with the business logic — the code that produces the site's content, which in turn is presumably derived from a database. Even inexperienced programmers have little difficulty understanding of HTML markup with the code that generates the content; this in turn makes PHP an easy language to learn and to teach. And compared to the common alternative of generating HTML markup with a programming language's `print` function, this certainly represents a significant improvement in readability and writability.

However, the effect this has is precisely the opposite of that desired: the business logic and the presentation are thoroughly commingled. This makes incremental design changes particularly hard. An uncharitable observer might speculate that this is why so few PHP-driven websites receive timely improvements to their æsthetics and user interface.

How then should you cope with this disadvantage? The only option is for the developers to voluntarily impose on themselves a disciplined separation that the language cannot enforce. The basic approach is as follows:

Divide every PHP page up into two parts: the former performs database queries and does whatever else is needed to calculate the content for the page, before storing the calculated content into a series of variables which — and this is crucial — contain no markup. Then the latter part of the page can simply embed variable values into HTML markup, using loops where appropriate to traverse potentially-complex data structures containing the actual content.

Should the site's design need to change at a later date, it is likely that many alterations can be made only to the latter part of the page; the business logic (which was presumably written with significant attention to security and correctness) should remain as it is. In some cases, it may even be possible to put the two parts of the page in separate files.

This approach can be augmented with any of a wide of variety of templating systems built in PHP — or even something like XSLT, given that roll-your-own templating systems are never quite powerful enough.

Once you've made the decision to use a templating system of some sort, it's worth stopping to consider what PHP actually gains you in comparison with a templating system for some other language — given that mixing code and markup is one of the PHP's defining features.

### 3 Using a Team of Developers

PHP seems to be aimed predominantly at lone developers, rather than at teams attempting to develop serious websites. The single biggest problem with PHP *qua* team development language is its paucity of namespaces. All functions live in a single namespace, with no exceptions. This has the following consequences:

1. You can redefine a function defined by another member of the team. Developers need to have extremely good lines of communication to avoid this.
2. Even worse, you can redefine a *built-in* function used in code written by another developer.
3. File inclusion (PHP's main mechanism for code reuse) tends to exacerbate this problem: trying to use code written anywhere else can be a portability nightmare.

Significant problems crop up even outside the function namespace. PHP has historically defaulted to allowing undeclared variables with no warnings; this exacerbates the single-namespace problem for developers trying to collaborate. Relying on the `register_globals` setting in the `php.ini` file (as many projects do) further compounds this problem, as we discuss in more detail below.

### 4 Deployment Problems

Many aspects of the PHP language can be configured dynamically using a `php.ini` file. Developers accustomed to traditional languages are typically surprised to discover that the `php.ini` file allows them to configure such features as whether certain global variables are created implicitly (`register_globals`) and whether incoming data is arbitrarily munged (`magic_quotes_gpc`).

The ability to change the language in this way is in one sense a good thing: some of the most awkward parts of the language can — and should — be disabled during development. For example, it's best to configure PHP to throw an error when you try to use unassigned variables; this makes PHP more similar to programming languages which require all variables must be declared before use.

However, this can lead to problems in deployment. The biggest issue with the `php.ini` file is that there can be only one per web server. For a web application that will not share hardware with other sites this isn't much of a problem, but it isn't universally the case that this can be accomplished. It's common for companies to deploy many smaller sites on one machine, to save on hardware, bandwidth, and administration costs. It's also common for sites to be deployed on servers managed by web-hosting companies. With web hosts in particular, if the hosting company configures PHP in one way, *every* site served by that hosting company will have to use that configuration. With Perl, by contrast, each individual source file specifies (for example) whether it has 'strict' variable declarations.

Another problem thrown up in deployment is that you can't in general rely on a web host having any particular version of PHP. To all intents and purposes, this requires that the developers

both know what version of PHP will be on the web host, and in addition use the same version for development. Ironically, the effect of this problem is diminished by the presence of security holes in a number of PHP versions: web hosts in particular are likely to use more recent versions of PHP because they contain fixes for vulnerabilities in earlier versions.

Related to the versioning issue is one of module availability. One of the advantages claimed for PHP over languages such as Perl is that ‘everything is built in’. Where Perl requires you to download and install, say, a library that provides access to a given database server, for PHP such code is an integral part of the language implementation. The problem, of course, is that a given web host might not provide a particular module that your code relies on. For PHP, this is essentially impossible to fix without the assistance of the web server administrator; for a language like Perl, on the other hand, add-on modules for CGI programs can readily be installed in a user’s home directory.

These issues are very hard to deal with; versioning and module availability in particular simply cannot be handled without knowledge of the deployment platform. As for the `php.ini` file, the best approach is two-pronged. Firstly, configure PHP on the development system to impose strictness where possible — for example, to warn about unassigned variables. Secondly, make sure that the code is agnostic about other configuration settings; we return to this issue in the next section.

## 5 Oversimplification Leading to Excessive Complexity

PHP was originally a collection of CGI scripts designed for building a ‘personal home page’. PHP 2 was much more akin to modern PHP: it was a simple scripting language with almost entirely the same syntax as what we now know as PHP. The fundamental design of the PHP 2 language was that it be simple to use, even for non-programmers.

The claim of this section is that this design goal has resulted in precisely the inverse of the desired effect. PHP 2 was probably a reasonable solution: it was a minimal scripting language for adding dynamic behaviour to webpages. Unfortunately, PHP has since outgrown itself. PHP is nowadays a programming language simplified to the point where the available abstractions and concepts just aren’t sufficient to express the programmer’s intent.

Furthermore, PHP suffers from a related issue: the language was accreted, rather than ever having been designed. Most programming languages that have achieved genuine long-lasting popularity have been the work of at most a small team of gifted language designers. These designers have worked as much on deciding what to leave out of their languages as on what to put in. PHP’s creator, Rasmus Lerdorf, adopted a deliberate policy of letting others contribute freely to the development of the PHP implementation — and thus implicitly to that of the language itself. Where languages like C, C++, or Perl have had a Ritchie, Stroustrup, or Wall directing their evolution, Lerdorf does not seem to have applied the same sorts of controls to PHP. The result in some cases is that PHP lacks any sort of coherent world view. This in turn only increases the complexity for those trying to actually build working solutions with PHP.

We will examine these issues in some depth, looking at some representative constructs in the language.

PHP's built-in functions are a good place to start. The vast majority of PHP's functionality — database access, regular expressions, etc. — is made available through built-in functions. This constitutes a problem right from the start: there is *no* modularity. Our assumption must be that namespace mechanisms were omitted from the language because they might be hard for non-programmers to understand. This seems like a minor point, but it does have repercussions, as discussed above.

Perhaps a worse problem in this area is simply the naming of the functions themselves. Function names (and semantics) have been liberally borrowed from Unix system calls (`unlink`), the C standard library (`strcspn`), and Perl (`split`), among other places. Some function names have multiple words separated by underscores (`str_replace`); others have words squashed together (`strtoupper`). Some functions have aliases, like `disk_free_space` and `diskfreespace`. Some functions are just plain misnamed, like the `addslashes` and `stripslashes` routines which deal with *backslashes* in strings. These complaints sound trivial, but the cognitive load of dealing with such a motley collection of names is anything but. I blame this problem entirely on PHP's piecemeal evolution.

Many languages offer both a numerically-indexed array type, and at least one other type which allows data to be indexed by strings (`map<T>` in C++, or Perl's hashes for example). PHP has both, but conflates them into a single array structure. This sounds like a big advantage from the point of view of simplicity: programmers needn't worry about type to use, because there is only one choice. Regardless of the wisdom or otherwise of this approach — are programmers to be encouraged to think sloppily about their data structures? — it certainly leads to increased complexity.

On the surface, it seems straightforward enough. We can use an array as if it's numerically indexed:

```
$array = Array(10, 11, 12);  
echo $array[0];
```

or as if it's a hash:

```
$age = Array('Anne' => 32,  
            'Bob'   => 28);  
echo $age['Anne'];
```

and PHP handles everything else behind the scenes. However, you can also do things like this:

```
$a1 = Array(10, 'Anne' => 32, 11, 'Bob' => 28, 12);  
$a2 = Array(1 => 21, 2 => 22, 3 => 23);  
$a2[0] = 20;
```

Questions to consider:

- What index do we have to consult to get the value 11 out of `$a1`?
- What's the iteration order for `$a2`? Is it numerically-indexed or hash-style?

- What happens if you use PHP's `$a2[] = ...` construct to add a new element to the 'end' of the array?
- Can numerically-indexed arrays have elements missing? If so, can you still trivially iterate over the values in index order?

While these questions can be answered (by *careful* reading of the manual and not a little experimentation), the answers aren't entirely obvious. Programmers with a reasonable understanding of basic data structures would be well advised to program as if a given PHP array can be indexed by either contiguous integers or by strings, but not both.

A similar problem crops up with relational operators. PHP's simple variables can contain either a number or a string. PHP converts between numbers and strings whenever necessary, so that the programmer need not worry about the underlying type.

That's the theory, anyway; in practice, this makes it extremely difficult to compare two values of different types for equality. The standard PHP equality-testing operator is spelled `==`:

```
$n1 = 0;      $n2 = 10;
$s1 = 'foo';  $s2 = 'bar';

$n1 == $n2;   // Numeric comparison, returns false
$s1 == $s2;   // String comparison, returns false
$n1 == $s1;   // Good question
$s1 == $n1;   // Is this any different?
```

The problem is that, when the values are weakly typed, and there is only one equality-testing operator, the language cannot determine the programmer's intention. PHP has a variety of kludges to deal with this:

- Some programmers recommend using `!strcmp($n1, $s1)` to force a string interpretation.
- You can cast one or both sides to the intended type.
- PHP 4 has a `===` operator. It works the same way as `==`, except that it additionally requires both sides to have the same type. This sounds initially like the perfect solution — just use `===` wherever you would otherwise have used `==`. Unfortunately, you lose the convenience of being able to treat numbers and strings interchangeably: it is no longer the case that `3 === "3"`.

Perl has similar rules for its scalar variables: they can contain either a number or a string, and the language freely converts as necessary. However, Perl allows the programmer to express the intent: it offers both a numeric-equality operator `==` and a string-equality operator `eq`. PHP explicitly rejects this approach. The manual for PHP 2 says "Once you start having separate operators for each type you start making the language much more complex. You can't use `==`

for strings, you now would use `eq`. I don't see the point, especially for something like PHP where most of the scripts will be rather simple and in most cases written by non-programmers who want a language with a basic logical syntax that doesn't have too high a learning curve." Unfortunately, in cases like this, the complexity is rather like an air bubble under freshly hung wallpaper: pushing it from one place to another will only leave more mess on the wall.

This is a clear case of PHP trying to simplify the problem beyond the limits of possibility. Languages like C++ can manage with only one equality testing operator because they are statically typed. Perl avoids static typing distinctions for numbers and strings, but uses additional operators to determine the programmer's intent. PHP follows neither path, with the result that comparing values is unnecessarily complex.

Variable scoping in user-defined functions is extremely peculiar. PHP almost entirely follows the common convention of using block scoping. PHP variables are created by assigning to them. Once a variable has been assigned, it is available within inferior scopes (nested blocks) as most programmers would expect:

```
$message = 'Hello, world!';
if (!$quiet) {
    echo $message, "<br />";
}
```

However, function bodies don't have access to variables from higher scopes:

```
$message = 'Hello, world!';

function print_message ()
{
    // Tries to print a nonexistent local variable
    echo $message, "<br />";
}
```

Accessing a global variable from within a function requires the use of the `global` statement:

```
$message = 'Hello, world!';

function print_message ()
{
    // Bring the variable into this function's scope
    global $message;
    echo $message, "<br />";
}
```

This scoping rule is sufficiently unusual that many PHP programmers — even skilled programmers with a great deal of PHP experience — report spending lots of time tracking down bugs

caused by forgetting to use `global`. The problem is particularly severe for PHP's built-in variables. For example PHP provides a `$HTTP_USER_AGENT` variable, taken from the web server's equivalent status variable. What happens if you use this in a PHP function without declaring it `global`? PHP doesn't let you access the global variable that it created for you. Nor does it by default give you any error message. Instead, it assumes that you wanted a completely random, uninitialised variable.

The rationale for designing the language in this way was that it would prevent inadvertent changing of global variables. This strikes me as an extremely good example of how designing the language for surface simplicity in fact leads to a great deal of underlying complexity.

One of the most abject examples of PHP's tendency to pander to non-programmers is the feature called `magic_quotes_gpc`. The idea behind this is as follows. Many PHP pages need to take data from HTML forms and use that data in database queries. It's common to see code roughly like this:

```
// $max_price is taken directly from the CGI parameters
db_query("SELECT id, name, description, price
        FROM item
        WHERE price <= $max_price");
```

This makes use of PHP's convenient variable-interpolation feature to build a dynamic query from user-supplied data. However, consider what the unpleasant consequences of this could be. Unexpected characters in the user's data could confuse the query parser; this would be awkward, but not terrible. Or worse, a user might be able to construct malformed data that would persuade the database server to run an additional query of the attacker's choosing.

Given that fact, languages with convenient database interfaces make it easy to have user-supplied data quoted appropriately. For example, some languages let you write the query with 'placeholders' for user-supplied data. The programmer can then supply the actual data in a separate step. As long as the database API knows how to quote data for the DBMS in use, this is guaranteed to avoid errors.

PHP's approach is rather different. Instead of making sure that database queries are valid, PHP chooses to ensure that the input can't be invalid. When you enable the `magic_quotes_gpc` setting, PHP alters its handling of the CGI parameters, and *actually inserts backslashes before 'dangerous' characters in the incoming data*. (An astute reader might stop to wonder why this feature isn't called `magic_backslashes_gpc`, but that's rather a side issue.)

If you weren't expecting this behaviour, the symptom is that PHP randomly throws backslashes into perfectly valid data — all in the name of *not* corrupting data! Lerdorf attempted to justify this in a recent interview as follows: "the worst that would happen is that someone would see an extra `\` on the screen when they output the data directly instead of sticking it into the database." I for one get rather concerned when I encounter such flagrant disregard for data integrity.

How to deal with this? The goal is to work transparently, regardless of the setting of `magic_quotes_gpc`. It's quite hard, though, because the damage has already been done by the time your PHP code runs. You must examine the configuration settings, and apply `stripslashes` to each CGI parameter before running the rest of the program:

```
if (isset($param1) && get_magic_quotes_gpc())
    $param1 = stripslashes($param1);
```

Then you can write the rest of your program as if `magic_quotes_gpc` didn't exist.

The last area I will cover is PHP's handling of CGI parameters. Traditionally, PHP takes all the GET and POST parameters, any values set in HTTP cookies, any environment variables, and all the web server status variables, and made each of them available to PHP programs as a global variable. This is certainly a helpful feature for beginners: no effort need be expended to find the CGI parameters. This feature is enabled by the `register_globals` configuration setting.

However, this approach has three major problems:

1. It leads to hideous pollution of the global variable namespace, with all the concomitant security vulnerabilities and problems for working in larger teams.
2. Within your functions, you have to remember to `globalise` the variables you're interested in.
3. It's just too plain hard to distinguish cookies, CGI parameters, and server variables. There's a further configuration setting, `gpc_order`, which specifies which family of variables take precedence, but relying on yet another `.ini`-file setting would be unwise.

For these reasons, the PHP maintainers have long counselled against enabling `register_globals`, and the most recent releases default to turning that setting off. Instead, the language now has some special 'autoglobal' or 'superglobal' variables to hold information like this. These autoglobal variables are like ordinary globals, with the exception that they don't need to be explicitly `globalised`. Amongst the available autoglobals are:

- `$_GET`, an array of CGI GET parameters
- `$_POST`, an array of CGI POST parameters
- `$_COOKIE`, an array of incoming HTTP cookies
- `$_REQUEST`, an array of all the above

There are two major problems with the autoglobals:

1. They aren't available in older versions of PHP
2. You still have CGI parameters conflated with cookies

In many sites, all the programmer wants is to be able to access the CGI parameters appropriate for the request method that the client used, and to keep those separate. The only reliable way to do this in a way that will work for older PHP versions as well as current ones is to check both `$HTTP_GET_VARS` and `$HTTP_POST_VARS` for CGI parameters, and remember to `globalise` when necessary.

It's merely an additional annoyance that PHP won't let you create your own autoglobal variables. If you could, you create a `$_CGI` autoglobal that did the right thing.

## 6 When to Use PHP

We have identified a number of problems that arise when attempting to use PHP on large websites. However, we do not claim that PHP should never be used, only that these issues make its use inadvisable in certain circumstances. The following checklist suggests questions that should be considered before making a decision to use PHP for a given project.

- How much control will you have over the deployment platform? PHP's one-size-fits-all approach to the `php.ini` file makes it hard to share servers with sites that were developed with different settings.
- How many people will work on the site, now and in the future? PHP as a language lacks the features necessary to promote effective teamwork; the bigger your team, the greater the problems you'll have.
- How big will the site be, in terms of numbers of distinct pages? This is related to the previous item: the bigger the site, the greater your need will be for language features that promote teamwork.
- How long will the site be expected to last? The longer it lasts, the more likely it is that significant design changes will be needed. If you use PHP in the obvious manner, major design changes are difficult. If you extend PHP with a templating system, whether *ad hoc* or carefully enforced, using PHP buys you little if anything.
- How experienced are the developers; and how complex will the site need to be? Experienced developers will find themselves hindered rather than helped by the language's simplicity. Inexperienced developers will find the simplicity a significant boon — but if you have inexperienced developers trying to develop a complicated dynamic site, you will soon run into other problems.

## 7 Conclusions

PHP is a convenient language for rapidly prototyping simple dynamic websites. Websites thus built can in many cases be deployed indefinitely, without spending time and money on refactoring code in a different language. PHP's simplicity makes it a good language for inexperienced programmers, such as those moving from a pure page-design rôle to a site development one.

For more experienced developers, though, the language's simplicity rapidly turns into complexity, slowing down the development process. These developers are the ones who have the skills needed to build large and/or complex websites; using PHP for such sites therefore tends to be a net loss. This tendency is reinforced by PHP's lack of the linguistic features needed to promote working on large software projects. If your project is at all large or complex, it may be better to look elsewhere when choosing an implementation language.

In cases where PHP has been determined to be inappropriate, what language should be used? There is considerable choice here; few languages are as bad as PHP for doing serious development work. The author and his colleagues have had good results with Perl, and believe that languages such as C++, Java, and Python should serve equally well.