Taking PHP Seriously

Keith Adams Follow
Oct 11, 2016 · 9 min read

by Keith Adams, Slack Engineering

Soyuz rocket delivered to the launchpad by train. Public domain photo by NASA

Slack uses PHP for most of its server-side application logic, which is an unusual choice these days. Why did we choose to build a new project in this language? Should you?

Most programmers who have only casually used PHP know two things about it: that it is a bad language, which they would never use if given the choice; and that some of the most extraordinarily successful projects in history use it. This is not quite a contradiction,

but it should make us curious. Did Facebook, Wikipedia, Wordpress, Etsy, Baidu, Box, and more recently Slack all succeed *in spite of* using PHP? Would they all have been better off expressing their application in Ruby? Erlang? Haskell?

Perhaps not. PHP-the-language has many flaws, which undoubtedly have slowed these efforts down, but PHP-the-environment has virtues which more than compensate for those flaws. And the options for improving on PHP's language-level flaws are pretty impressive. On the balance, PHP provides better support for building, changing, and operating a successful project than competing environments. I would start a new project in PHP today, with a reservation or two, but zero apologies.

Background

Uniquely among modern languages, **PHP was born in a web server**. Its strengths are tightly coupled to the context of request-oriented, server-side execution.

PHP originally stood for "Personal Home Page." It was first released in 1995 by Rasmus Lerdorf, with an aim of supporting small, simple dynamic web applications, like the guestbooks and hit counters that were popular in the web's early days.

From PHP's inception, it has been used for far more complicated projects than its creators anticipated. It has been through several major revisions, each of which brought new mechanisms for wrangling these more complex applications. Today, in 2016, it is a feature-rich member of the Mixed-Paradigm Developer Productivity Language (*MPDPL*) family**[1]**, which includes JavaScript, Python, Ruby, and Lua. If you last touched PHP in the early 'aughts, a contemporary PHP codebase might surprise you with traits, closures, and generators.

Virtues of PHP

PHP gets several things very deeply, and uniquely, right.

First, **state**. Every web request starts from a completely blank slate. Its namespace and globals are uninitialized, except for the standard globals, functions and classes that provide primitive functionality and life support. By starting each request from a known state, we get a kind of organic fault isolation; if request *t* encounters a software defect and fails, this bug does not directly interfere with the execution of subsequent request

t+1. State does reside in places other than the program heap, of course, and it is possible to statefully mess up a database, or memcache, or the filesystem. But PHP shares that weakness with all conceivable environments that allow persistence. Isolating request heaps from one another reduces the cost of most program defects.

Second, **concurrency**. An individual web request runs in a single PHP thread. This seems at first like a silly limitation. But since your program executes in the context of a web server, we have a natural source of concurrency available: web requests. Asynchronously curl'ing to localhost (or even another web server) provides a shared-nothing, copy-in/copy-out way of exploiting parallelism. In practice, this is safer and more resilient to error than the locks-and-shared-state approach that most other general-purpose languages provide.

Finally, the fact that PHP programs operate at a request level means that **programmer workflow** is fast and efficient, and stays fast as the application changes. Many developer productivity languages claim this, but if they do not reset state for each request, and the main event loop shares program-level state with requests, they almost invariably have some startup time. For a typical Python application server, e.g., the debugging cycle will look something like "think; edit; restart the server; send some test requests." Even if "restart the server" only takes a few seconds of wall-clock time, that takes a big cut of the 15–30 seconds our finite human brains have to hold the most delicate state in place.

I claim that PHP's simpler "think; edit; reload the page" cycle makes developers more productive. Over the course of a long and complex software project's life cycle, these productivity gains compound.

The Case Against PHP

If all of the above is true, why all the hate? When you boil the colorful hyperbole away, the most common complaints about PHP cluster around these root causes:

1. **Surprise type conversions**. Almost all languages these days let programmers compare, e.g., integers and floats with the >= operator; heck, even C allows this. It's perfectly clear what is intended. It's less clear what comparing a string and an integer with == is supposed to mean, and different languages have made different choices. PHP's choices in this department are especially perverse, leading to

surprises and undetected errors. For instance, 123 == “123foo” evaluates to true (see what it’s doing there?), but 0123 == “0123foo” is false (hmm).

2. **Inconsistency around reference, value semantics**. PHP 3 had a clear semantic that assignment, argument passing, and return are all by value, creating a logical copy of the data in question. The programmer can opt into reference semantics with a & annotation**[2]**. This clashed with the introduction of object-oriented programming facilities in PHP 4 and 5, though. Much of PHP’s OO notation is borrowed from Java, and Java has the semantic that objects are treated by reference, while primitive types are treated by value. So the current state of PHP’s semantics is that objects are passed by reference (choosing Java over, say, C++), primitive types are passed by value (where Java, C++, and PHP agree), but the older reference semantics and & notation persist, sometimes interacting with the new world in weird ways.

3. **Failure-oblivious philosophy**. PHP tries very, very hard to keep the request running, even if it has done something deeply strange. For instance, division by zero does not throw an exception, or return INF, or fatally terminate the request. By default, it warns and evaluates to the value false. Since false is silently treated as 0 in numeric contexts, many applications are deployed and run with undiagnosed divisions by zero. This particular issue is changed in PHP 7, but the design impulse to keep plowing ahead, past when it could possibly make sense, pervades libraries too.

4. **Inconsistencies in the standard library**. When PHP was young, its audience was most familiar with C, and many APIs used the C standard library’s design language: six-character lower case names, success and failure returned in an integer return value with “real” values returned in a callee-supplied “out” param, etc. As PHP matured, the C style of namespacing by prefixing with _ became more pervasive: mysql_…, json_…, etc. And more recently, the Java style of camelCase methods on CamelCase classes has become the most common way of introducing new functionality. So sometimes we see code snippets that interleave expressions like new DirectoryIterator($path) with if (!($f = fopen($p, ‘w+’)) … in a jarring way.

Lest I seem like an unreflective PHP apologist: **these are all serious problems that make defects more likely**. And they’re unforced errors. There’s no inherent trade-off between the Good Parts of PHP and these problems. It should be possible to build a PHP that limits these downsides while preserving the good parts.

HHVM and Hack

That successor system to PHP is called Hack**[3]**.

Hack is what programming language people call a 'gradual typing system' for PHP. The 'typing system' means that it allows the programmer to express automatically verifiable invariants about the data that flows through code: this function takes a string and an integer and returns a list of Fribbles, just like in Java or C++ or Haskell or whatever statically typed language you favor. The 'gradual' part means that some parts of your codebase can be statically typed, while other parts are still in rough-and-tumble, dynamic PHP. The ability to mix them enables gradual migration of big codebases.

Rather than spill a ton of ink here describing Hack's type system and how it works, just go play with it. I'll be here when you get back.

It's a neat system, and quite ambitious in what it allows you to express. Having the option of gradually migrating a project to Hack, in case it grows larger than you first expected, is a unique advantage of the PHP ecosystem. Hack's type checking preserves the 'think; edit; reload the page' workflow, because the type checker runs in the background, incrementally updating its model of the codebase when it sees modifications to the filesystem. The Hack project provides integrations with all the popular editors and IDEs so that the feedback about type errors comes as soon as you're done typing, just like in the web demo.

Let's evaluate the set of real risks that PHP poses in light of Hack:

1. **Surprise type conversions** become errors in Hack files. The entire class of problems boils away.
2. **Reference and value semantics** are cleaned up by simply banning old-style references in Hack, since they're unnecessary in new codebases. This leaves behind the same objects-by-reference-and-everything-else-by-value semantics as Java or C#..
3. PHP's **failure-obliviousness** is more a property of the runtime and libraries, and it is harder for a semantic checker like Hack to reach directly into these systems. However, in practice most forms of failure-obliviousness require surprise type conversions to get very far. For instance, problems that arise from propagating the

'false' returned from division by zero eventually cross a type-checked boundary**[4]**, which fails on treating a boolean numerically. These boundaries are more frequent in Hack codebases. By making it easier to write these types, Hack decreases the 'skid distance' of many buggy executions in practice.

4. Finally, **inconsistencies in the standard library** persist. The most Hack hopes to do is to make it less painful to wrap them in safer abstractions.

Hack provides an option that no other popular member of the MPDPL family has: the ability to introduce a type system after initial development, and only in the parts of the system where the value exceeds the cost.

HHVM

Hack was originally developed as part of the HipHop Virtual Machine, or HHVM, an open source JIT environment for PHP. HHVM provides another important option for the successful project: the ability to run your site faster and more economically. Facebook reports an 11.6x improvement in CPU efficiency over the PHP interpreter, and Wikipedia reports a 6x improvement.

Slack recently migrated its web environments into HHVM, and experienced significant drops in latency for all endpoints, but we lack an apples-to-apples measurement of CPU efficiency at this writing. We're also in the process of moving portions of our codebase into Hack, and will report our experience here.

Looking Ahead

We started with the apparent paradox that PHP is a really bad language that is used in a lot of successful projects. We find that its reputation as a poor language is, in isolation, pretty well deserved. The success of projects using it has more to do with properties of the PHP *environment*, and the high-cadence workflow it enables, than with PHP the language. And the advantages of that environment (reduced cost of bugs through fault isolation; safe concurrency; and high developer throughput) are more valuable than the problems that the language's flaws create.

Also, uniquely among the MPDPLs, there is a clear migration path to a higher performance, safer and more maintainable medium in the form of Hack and HHVM.

Slack is in the later stages of a transition to HHVM, and the early stages of a transition to Hack, and we are optimistic that they will let us produce better software, faster.

. . .

Slack Technologies, Inc. is looking for great technologists to join us.

. . .

Notes

1. I made up the term 'MPDPL.' While there is little direct genetic relationship among them, these languages have influenced one another heavily. Looking past syntax, they are much more similar than different. In a universe of programming languages that includes MIPS assembly, Haskell, C++, Forth, and Erlang it is hard to deny that the MPDPLs form a tight cluster in language design space. [**Back to text**]

2. Unfortunately the & was marked in the callee, not the caller. So the programmer declares a desire to receive params by reference, but actually passing them by reference is unmarked. This makes it hard to understand what might change when reading code, and complicates an efficient implementation of PHP significantly. See Figure 2 in http://dl.acm.org/citation.cfm?id=2660199 [**Back to text**]

3. Yes, Hack is a nearly unGoogleable programming language name. 'Hacklang' is sometimes used when ambiguity is possible. If Google themselves can name a popular language the still-more-unGoogleable **Go**, why not? [**Back to text**]

4. The typechecks in a Hack program are also enforced at runtime by default, because they piggy-back on PHP's "type hint" facility. This increases safety in mixed codebases where Hack and classic PHP are co-mingled. [**Back to text**]

Programming PHP Hhvm Hacking