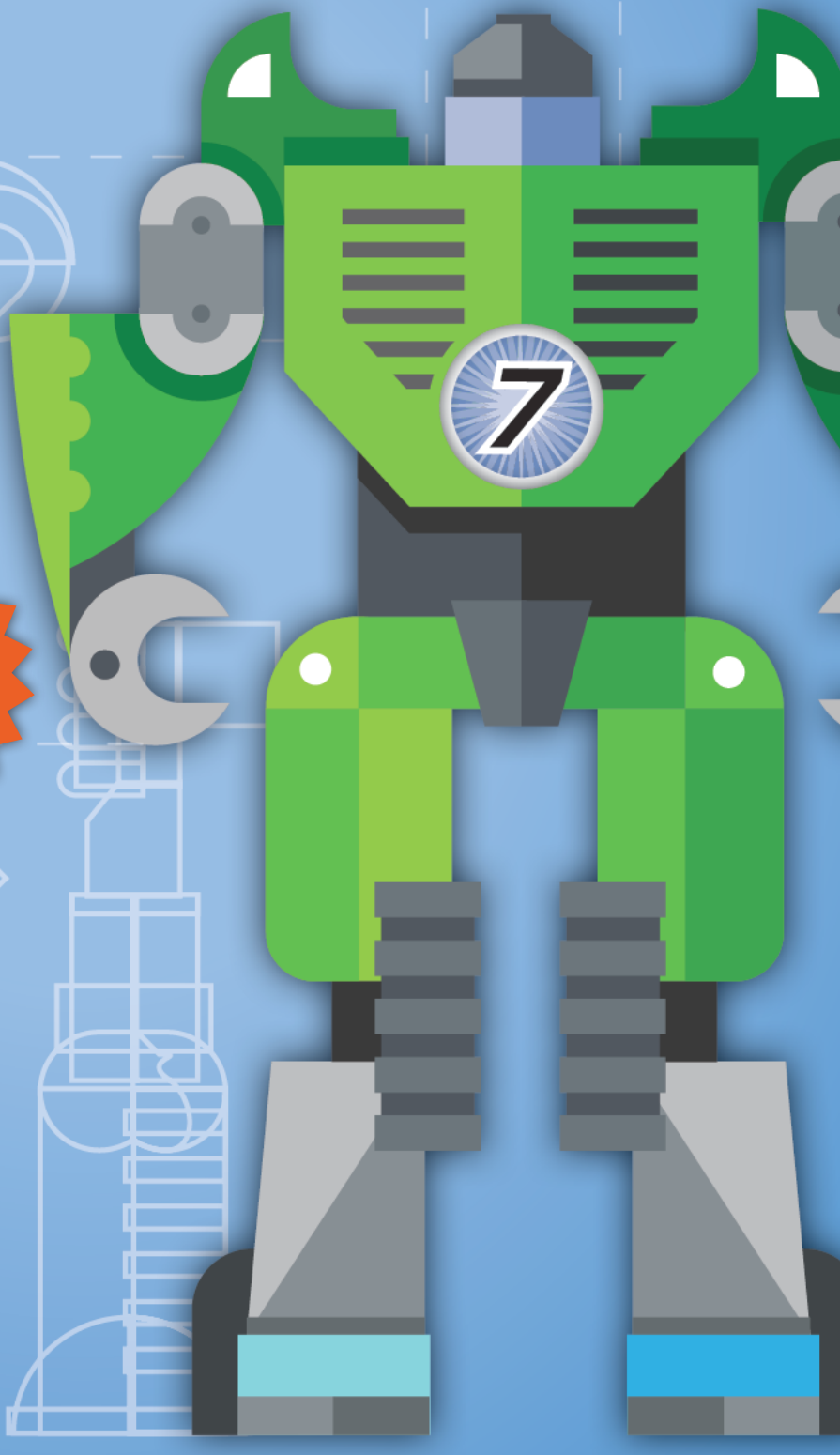
www.phparch.com

October 2015
VOLUME 14 - ISSUE 10

Integrating Extensions

Solr Power for All, Part 2
Dissecting the PHAR Format
Artificial Neural Networks with PHP

php[architect]



ALSO INSIDE

Casting Tales in PHP

Education Station:
How to Convert a WordPress Blog to Sculpin

Leveling Up:
What's Coming in PHP 7

Community Corner:
October 2015

finally{}:
"Isn't PHP Dead?"

FREE Article!

Casting Tales in PHP

Zachary Drillings

As a dynamically typed language, working with data types in PHP can sometimes seem like a dark art. This article will discuss some of the trade-offs inherent to PHP and delve into some of the strangest behaviors my team has debugged while developing a high-volume PHP application.

DisplayInfo()

Requirements

- PHP: 5.3+

Other Software

- MySQL 5.0+
- Apache 2

Related URLS

- PHP Manual—https://php.net/manual/
- MySQL—http://www.mysql.com
- PHP History—https://php.net/history.php
- Scripting: higher level programming for the 21st Century—http://phpa.me/ieee-scripting-21st
- Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages—http://phpa.me/lopes-types-pdf
- Floating point numbers—https://php.net/language.types.float
- PHP type comparison tables—https://php.net/types.comparisons
- PHP RFC: Scalar Type Declarations—https://wiki.php.net/rfc/scalar_type_hints_v5

Introduction

One of the core language features of PHP is dynamic typing. In a statically typed language, a developer must manually cast a variable between cases, as in these examples from Java:

```
//Casting an Integer to a Float
int x = 1;
float y = (float) x;
//Casting a String to an Integer
String s = "10";
int z = int foo = Integer.parseInt(s);
```

Theoretically, dynamic typing provides for more concise code and increased ease of use as the interpreter picks up the slack, assigning and reassigning types as indicated by the programmer. In most cases, this system works entirely as expected; however, when it does not, it can lead to bizarre situations.

PHP was originally intended to process HTML forms on the Internet. Although this has branched out over the years, it continues to be one of the most common-use cases. An extension of this, and a very popular subset of use cases in its own right, is to use PHP to power a RESTful API. Using HTTP as the communication mechanism, RESTful APIs are designed

to take user input just like web forms; very often they are also designed to persist this to some sort of datastore on the back end. This back-end datastore is traditionally a relational database management system (RDBMS), and typically RDBMSs are strictly typed.

In the course of developing a RESTful API in PHP with MySQL (in this case, our chosen back-end RDBMS), we have come across many interesting use cases and unexpected behaviors. The combination of arbitrary user input, dynamically typed PHP, and a RDBMS has provided many hours of intense debugging and has required exceptional care to sanitize and validate input.

To set the stage for this typing adventure, this article will go through some background on typing, why my team uses PHP, and some of the most interesting bugs that we have encountered. Finally, there will be a discussion of some conclusions that can be drawn, and a discussion of strategies for going forward.

Dynamic Versus Static Typing

There is continual debate in computer science and software development over the benefits and deficits to using statically or dynamically typed languages. Of course, each has its own place with positives and negatives. In the end, the debate comes down to personal preference and use case.

Static Typing

Statically typed languages, such as C or Java, require all variables to be initialized with a typed value; a compile-time error will result if an attempt is made to assign a value of the wrong type, or, more significantly, to pass a variable of one type into a function expecting another. Four strong benefits of this, as described in Erik Meijer and Peter Drayton's paper *Static Typing Where Possible, Dynamic Typing When Needed: The End of the Cold War Between Programming Languages* (see Related URLs), are:

1. It makes it somewhat harder to make certain errors, in which the developer misuses variables. This can prevent long debugging sessions from occurring late in application development and ultimately save time.
2. Another aspect of improving the development experience is clearer self-documentation, achieved by providing more information in function/method declarations. Instead of relying on the previous developer to make the type-of-input parameters clear, it is a required portion of the signature.
3. Static typing also provides ways to improve the program's run speed. It does this in two ways:
   - It allows the compiler to make optimizations that are not otherwise possible. For instance, if the type signature is not known, the compiler is unable to use a direct method call. Although virtual method calls do not require an exceptional amount of extra time, this additional overhead can add up.
   - It reduces type-determination baggage. Since assigning new values to variables does not require runtime determination of variable type, this extra overhead is not introduced. This aspect also shrinks the program size, since none of this extra type baggage is required. (An example of this extra information is PHP's Zval.)
4. No runtime type exceptions will occur. This is related to the first point in regard to catching errors earlier, but is important in its own right.

Dynamic Typing

On the other hand, dynamic typing is generally intended to be faster and more expressive, and lines up better with a mental image of program structure. In addition to PHP, other major dynamically typed languages include JavaScript, Python, and Ruby. Benefits, described in John Ousterhout's paper *Scripting: Higher Level Programming for the 21st Century* (see Related URLs), include:

1. Faster development by not requiring the additional typing out of data types.
2. No restrictions on how a variable may be used allow the developer to build in hacks and use cases that would not be possible with static typing. This is especially beneficial during the rapid prototyping phase of a project when ideas (and types along with them) may change often and results are paramount.
3. Because of the lack of type declarations, code is more concise. With proper naming conventions it should still be abundantly clear what a variable is intended to be, and the programmer can more easily see all business logic.
4. Dynamic typing allows for easier function overloading—particularly overloading that is far less verbose. The programmer is able to use one function, possibly calling a single function containing the actual business logic; as opposed to using several different functions, each with their own signature.
   - This is even more true in PHP, specifically, which allows for parameters with default values.

Relative Merits

The one area in which statically typed languages are generally able to declare a true victory is execution speed. Because of the diminished code baggage and possible optimizations, the win is clear. There are few arenas in modern application design in which these small gains are truly going to affect the end user, but if you plan to make micro-optimizations on this level, then your choice is likely set.

The argument is typically made that it is faster to write code in a dynamically typed language. However, due to the added testing time and potential issues in production, this is certainly not always the case. For quick prototyping, a dynamic language definitely shows benefits; however, for writing a full system, the trade-offs are less clear.

Aside from the realm of ultra-high-speed applications, programmer preference takes a much larger role. Do you prefer to write excellent tests (a must anyhow) and risk debugging strange runtime errors, or prefer to deal with a larger variety of compile-time errors? Is it preferable to have concise and clear code, or superior self-documentation?

How We Wound Up Here

As an organization we believe it is extremely important to work with clients and provide for the greatest ease of use possible. To this end, we prioritized two things:

1. A RESTful API as a first-class citizen of the ecosystem.
2. Great ease of use for customers, especially not throwing errors if their intent is clear. (PHP allows working with integers as strings; why shouldn't we?)

With many great libraries, a highly active community, and its specialty in processing forms, PHP was an ideal choice for the need. MySQL is similarly well adapted and the PHP/MySQL working relationship is well designed, well documented, and well understood. Add to these the interest in rapid development and prototyping, and the benefits of this system design are clear. Unfortunately, as we have grown and the system has become increasingly complex, we have run into some issues.

We Live in a Stringy World

As discussed earlier, PHP was originally designed to process web form input. In our API this is JSON input, but that's not entirely relevant. What is more important is that this input arrives from the POST input stream as a **string** (`file_get_contents("php://input")`)! As we continue to dive through casting in PHP, it is imperative to realize that we *must* cast in order to arrive with data in any sort of usable format. PHP's dynamic typing makes the manipulation smoother than it might otherwise be (one more reason to use it), but without use of casts we simply could never work with our user input.

Why User Input + PHP + MySQL is a Dangerous Combination

Working toward the goal of being flexible for our clients, we have traditionally made our best attempt to cast before rejecting an input, as opposed to being fundamentalist about checking input types. For instance, if a user passed in an `id` value as a string, but we expected it to be an integer (`"id":"19"` instead of `"id":19`), we would still like to accept this. Our code, then, wound up looking somewhat like Listing 1.

You may be looking at the line `if ($field != $jsonIn->id)` and questioning the validity of using only the double equals comparison operator as opposed to the triple equals operator, which additionally compares for type. With the cast to an integer that has already been performed, using triple equals would allow us to catch only the situation in which our user input an integer. If this were our intention, then we would only

LISTING 1

```
// Extract POST body
$input = file_get_contents("php://input");
$jsonIn = json_decode($input);
if (!$jsonIn) {
   throw new Exception("Bad Input - Not Json");
}

// Assuming we wish to accept an integer field
$field = (int) $jsonIn->id;
if ($field != $jsonIn->id) {
   throw new Exception("Bad Input - Not an int");
}
// Validate $field in any other ways
$adodb_mysqli->execute(
   "Insert into example ('id') VALUES ({$jsonIn->id})"
);
```

need to check the input with `is_int($jsonIn->id)`. However, in order to allow greater flexibility, as previously stated, we wish for `'12' == 12` to be accepted in order to better accommodate the stringy nature of input.

Some Specific Issues We Have Seen

We have referred to the *potential* for strange, unexpected, and possibly dangerous behaviors related to casting in PHP. To drive these issues home, though, we will dive into some specific examples in this section: first, one of the more complex bugs we had to debug related to taking scientific notation as input; second, a recurring issue I have seen in many applications with evaluating strings as boolean values; and third, an additional gotcha to watch for.

Scientific Notation

Let's assume that we have a system which is expecting to take an integer as input and eventually pass this integer down into a database table. Perhaps this is an *id* acting as a foreign key from some other table. Although we may expect a user to pass in an integer (and they may in 99% of cases), what happens if they do something strange, such as pass in scientific notation?

Consider the following code example:

```
$scientificNotationVariable ='5e2';
$varPlusZero = $scientificNotationVariable + 0;
$castedVar = (int) $scientificNotationVariable;

echo $varPlusZero . "\n";
echo $castedVar . "\n";
```

OUTPUT 1

```
500
5
```

You might expect that our output would be the same on both lines. Our initial variable is clearly just "500", expressed in scientific notation. However, this is the Output Panel 1.

Experienced PHP developers will immediately realize that this occurs when doing arithmetic because our string is cast to a float and not to an integer. Floats, in PHP, can be expressed properly in scientific notation, as described in the PHP manual's *Floating Point Numbers* section (see Related URLs). However, when *not* doing arithmetic, the casting mechanism does not cast to a float before converting to an integer. Instead, all characters are grabbed for casting until the first non-numerical character is reached. In this case, `e` is the first non-numerical character, so we're left with only `5`.

If our intention is to use this—or another similar piece of code—as a validation (say to confirm that our user input is in the correct format, or to verify that the id exists in our database), we can very easily shoot ourselves in the foot and ruin data integrity! The value 5 certainly *looks* like valid input, but it is not what our user expects! It may even present a security hole.

OUTPUT 2

```
mysql > desc example;
+-------+------------------+------+-----+---------+-------+
| Field | Type             | Null | Key | Default | Extra |
+-------+------------------+------+-----+---------+-------+
| id    | int(10) unsigned | NO   |     | NULL    |       |
+-------+------------------+------+-----+---------+-------+
mysql > insert into example (`id`) VALUES (5e2);
mysql > insert into example (`id`) VALUES ('5e2');
mysql > select * from example;
+-----+
| id  |
+-----+
| 500 |
| 500 |
+-----+
```

But wait—this gets scarier! What happens if you try to push data into your datastore? Let's take the MySQL in Output Panel 2.

Now, of course, if we had been vigilant about inserting our *casted* data, then we simply wind up with data integrity issues and angry users (we would be

inserting 5). As we can see from the example above, though, if we perform validations on our casted data, and then (confident that it is correct) insert original data, we expose ourselves to a complex vulnerability—validating against 5, but inserting 500!

If, instead of arbitrary ids, these values represent cash amounts that a user may transfer, we may be relaxing confident that we allowed our user to request $5 when we have really given them the much larger sum of $500!

Strings to Booleans

In a similar vein to our previous discussion on casting to an integer, it is very common to expect a boolean value and intend to insert this into our database. Unfortunately, once again we find ourselves trying to support our clients and shooting ourselves in the foot as a result. In MySQL, booleans can be represented as `tinyint(1)`. This makes a lot of sense from a storage perspective, but what is the best way to solicit client input when you intend to store your value as 1 or 0?

Theoretically, you could imagine a user expecting to be able to input `1` or `true` in the best case, but what about the string `"true"`? In much the same way as we accepted our id as a string above, it is clear what the user's intention is here. Unfortunately, PHP behaves in an odd way in this respect as well.

Let's take a look at a few possible inputs and how they are interpreted:

```
if (1) echo "true";             // true
if (14) echo "true";            // true
if ('true') echo "true";        // true
if ('hello') echo "true";       // true
if ('false') echo "true";       // true
if (true) echo "true";          // true
if (false) echo "true";         // false, no output
if (0) echo "true";             // false, no output
if ('0') echo "true";           // false, no output
```

Several of these make a lot of sense—but things start to go off the rails when we try to check `14` or `"hello"`. Essentially, anything that is *not empty* is going to get evaluated to `true`. Our validations work well for false values, since there is only a small set of possible empty values (note: things get odd again if you try to check on `null`). However, aside from this limited set of empty values, anything and everything will evaluate to true. And to make matters worse the following is also true!

```
if ((bool) "hello" == true)
```

Although likely not as terrifying as our previous example, the ease with which we wind up storing data differently than our user expects is disconcerting. (We are unable to predict what any client would intend when passing `"hello"` in for a boolean.) The PHP manual possesses a complete list under *PHP type comparison tables* (see Related URLs).

The major difference in this situation from the prior example is that here, MySQL at least acts as our safety net, since attempting to insert

anything aside from a one byte integer into a tinyint column results in an error. In the scientific notation example, inserting the *cast* value into MySQL resulted in data inconsistent with user expectation, but was the safer of our alternatives. Here, inserting the *uncast* value saves us an error, but inserting the *cast* value would leave us with inconsistent data!

One More Gotcha–PHP is Very Active

PHP is also, as it turns out, very active from line to line with its casting. Although in many languages, a type hint is considered a powerful declaration from the programmer, PHP has no such respect.

Let us take another short code snippet, building off an earlier example:

```
$var = "5e4";
$var = (int) $var;
$var = $var + .2;
echo $var . "\n";
```

What would you desire this to output? What would you expect it to output? `5.2` is the output, of course. The interpreter accepts that we wish to cast our initial value to an integer, but this has no bearing going forward. On the very next line, it is glad to juggle `$var` into a float!

This is probably a bit less frightening than our previous two examples, but it's yet another important message to the PHP developer that you must remain ever vigilant when dealing with casting.

Conclusions

Here we have discovered some interesting dangers in how the PHP interpreter handles type casting. Although these are all easy enough to mitigate in theory, it demonstrates how easy it is to find yourself in a strange situation when you put too much trust in your tools and fail to validate your inputs completely. The biggest challenge in application development is typically integration, since connecting users through PHP to MySQL exposes the biggest concerns.

So What Can We Do?

One of our key lessons learned has been that being flexible and doing our best to format input is not advisable. Although we believe strongly in what we were trying to achieve, we jeopardized data integrity (at least in certain edge cases) by making our best attempts.

Our takeaway here is that we must engage with our users as partners. Instead of working as hard as we had to intuit their intentions and accept input whenever we can, it is much safer to be vigilant with the types of our input. We are better served by spending our efforts in writing effective error messages and documentation so that it is clear why our application is behaving as it does, rather than spending that same time attempting to guess what our users mean.

A small change that we made was to alter our input logic to become:

```
//Assuming we wish to accept an integer field
$field = (int) $jsonIn->id;
if ($field !== $jsonIn->id) {
   throw new Exception("Bad Input - Not an int");
}
//Validate $field in any other ways
```

You will notice the change from `!=` to `!==`. Although we previously discussed our very clear intention to use the double equals comparison logic, a new focus on engaging with our users as partners and choosing to be vigilant about data types mandates that we now reject client input of the wrong type.

Onwards to PHP 7

In the upcoming PHP 7, a powerful new tool for working with types has been implemented—scalar type hinting. The developer will now be able to declare scalar types (`int`, `float`, `string`, and `bool`) for function parameters and return values, and optionally choose whether violations of these type declarations will result in an error or an implicit cast. In the below example, we have declared `strict_types=1`, meaning that a fatal error will result if a value that is not an integer is passed in as a parameter (or returned, though this is not possible in this example). Listing 2 shows the syntax for using strict type hinting.

LISTING 2

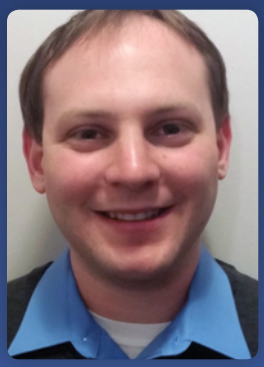
```
<?php
declare(strict_types=1);

function adder(int $a, int $b): int {
   return $a + $b;
}

class Adder {
   function add(int $a, int $b): int {
      return $a + $b;
   }
}
```

If, alternately, `strict_types=0` were declared, then an implicit cast of the input parameter to an integer would result. A complete RFC of the implemented system can be found under *PHP RFC: Scalar Type Declarations* (see Related URLs). Although there is some debate about whether these changes bring PHP too much towards a statically typed language, they definitely enable greater flexibility for developers. The largest value of these changes will likely be seen when integrating with other developers' interfaces and libraries, but it will surely have benefits for use cases involving input validation (as we have discussed).

In the end, working with any language and any system requires extensive knowledge of how best to work with the tools. The casting implementations within PHP that enable its powerful and dynamic typing engine have particular gotchas to watch out for. By being aware of them, and through exhaustive testing, it is possible to write great, stable applications, without slowing down.



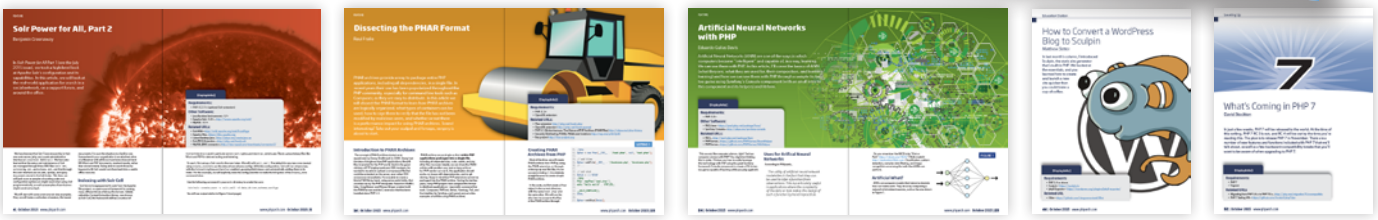
ZACH is an experienced Web Application Developer with a passion for architecting scalable RESTful APIs. He enjoys exploring the full-stack, from following advancements in transistor technology and writing toy assembler programs to discussing developments in JavaScript libraries. Zach has spent the last 3 years at AppNexus, Inc helping to scale a PHP-based API from 12,000 to 128,000 users. He is currently an Engineering Manager, where he leads a team of ten extremely talented full stack engineers to solve interesting advertising technology challenges by developing well-tested, secure applications.

Twitter: @zdrills

Want more articles like this one?

Keep your skills current and stay on top of the latest PHP news and best practices by reading each new issue of php[architect], jam-packed with articles.

Learn more every month about frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.



php[architect]

magazine

books

conferences

training

www.phparch.com

Get the complete issue for only $6!

We also offer digital and print+digital subscriptions starting at $49/year.