



php[architect]

The Magazine For PHP Professionals

Box of PHP

A Guide to Practical PHP FFI

Problem in a Box

ALSO INSIDE

The Workshop:
What Ben Ramsey Uses

Education Station:
12 Factor Applications: Parts 1-6

PHP Puzzles:
Grade Deviations

Artisan Way:
Laravel 10: New Features & Upgrade Impacts

Security Corner:
InfoSec 102: Phishing

DDD Alley:
“Depend” Toward Stability

PSR Pickup:
PSR-17: HTTP Factories

finally{}:
Nothing Lasts Longer



The Web Developer's

SECRET WEAPON!

Exception, uptime, and cron monitoring, all in one place and easily installed in your web app. Deploy with confidence and be your team's devops hero.

Are exceptions *all* that keep you up at night?

Honeybadger gives you full confidence in the health of your production systems.

DevOps monitoring, for developers. *gasp!*

Deploying web applications at scale is easier than it has ever been, but monitoring them is hard, and it's easy to lose sight of your users. Honeybadger simplifies your production stack by combining three of the most common types of monitoring into a single, easy to use platform.

Exception Monitoring

Delight your users by proactively monitoring for and fixing errors.

Uptime Monitoring

Know when your external services go down or have other problems.

Check-In Monitoring

Know when your background jobs and services go missing or silently fail.



Start Your Free Trial Today
<https://www.honeybadger.io/>

CONTENTS

MARCH 2023
Volume 22 - Issue 03



php[architect]

- | | |
|---|--|
| <p>2 Share to Learn</p> <p>3 A Guide to Practical Usage of PHP FFI
Bohuslav Šimek</p> <p>9 Problem in a Box
Edward Barnard</p> <p>13 12 Factor Applications: Parts 1-6
Education Station
Chris Tankersley</p> <p>17 InfoSec 102: Phishing
Security Corner
Eric Mann</p> <p>19 What Ben Uses
The Workshop
Ben Ramsey</p> | <p>22 Grade Deviations
PHP Puzzles
Oscar Merida</p> <p>25 “Depend” Toward Stability
DDD Alley
Edward Barnard</p> <p>30 PSR-17: HTTP Factories
PSR Pickup
Frank Wallen</p> <p>33 New and Noteworthy</p> <p>34 Laravel 10: New Features & Upgrade Impacts
Artisan Way
Matt Lantz</p> <p>36 Nothing Lasts Longer finally}}
Beth Tucker Long</p> |
|---|--|

Edited in a problem box

php[architect] is published twelve times a year by:

PHP Architect, LLC
9245 Twin Trails Dr #720503
San Diego, CA 92129, USA

Subscriptions

Print, digital, and corporate subscriptions are available. Visit <https://www.phparch.com/magazine> to subscribe or email contact@phparch.com for more information.

Advertising

To learn about advertising and receive the full prospectus, contact us at ads@phparch.com today!

Contact Information:

General mailbox: contact@phparch.com
Editorial: editors@phparch.com

Print ISSN 1709-7169
Digital ISSN 2375-3544

Copyright © 2023—PHP Architect, LLC
All Rights Reserved

Although all possible care has been placed in assuring the accuracy of the contents of this magazine, including all associated source code, listings and figures, the publisher assumes no responsibilities with regards of use of the information contained herein or in all associated material.

php[architect], php[a], the php[architect] logo, PHP Architect, LLC and the PHP Architect, LLC logo are trademarks of PHP Architect, LLC.

Share to Learn

John Congdon

Sharing ideas has a weird side effect that people don't often discuss.

With in-person events starting back up, we finally took the opportunity to restart our local user group, SDPHP. I highly encourage everyone to find a local community to be a part of, learn from, talk to, and share ideas with.

We go to presentations, conferences, online videos, etc... to learn from people that are sharing their ideas. But what happens behind the scenes—the people sharing those ideas learned along the way. EVERY SINGLE TIME that I've ever presented from my user group, given a conference presentation, written a blog post, or shared information in any meaningful way, it has forced me to dive deeper into the topic before sharing. That deep dive is called learning... I've learned about new things just because I wanted to share information that I already knew.

My most recent example is around Event Sourcing. This technology or idea isn't new; however, the way we use it for our applications has some great benefits, and I have been taking advantage of that in some small way for a couple of years now. So while I wanted to share how I was using it, I also took the opportunity to learn more and weave that into my presentations.

Share some small piece of something you have worked on lately with co-workers, speak at a local user group, submit to talk at php[tek] next year, or write for this very magazine. Be amazed at what you learn just because you want to share.

And now this month's articles ...

We've all heard the expression, 'think outside the box', but this month, Ed Barnard is giving us a feature article all about putting the problem in a box. If you struggle with debugging your code (and who doesn't, really?), I highly recommend reading this. Ed gives us his experience of using an intuitive approach to solve coding problems.

We move on from there to our next feature article, A Guide to Practical

Usage of PHP FFI, contributed by Bohuslav Šimek. *This article breaks down the benefits of using FFI, including the added benefits of FFI vs. using PHP extensions alone. Bohuslav gives us a great example of FFI and Duckdb, pointing out some of the drawbacks of converting code.*

Lure—check; hook—check; fishing pole—check, now let's climb aboard as Eric Mann takes us into the world of phishing in his latest security column, 'InfoSec 102: Phishing.' Eric explains the common terms used in the security world and how understanding them can be helpful to you.

Over in Artisan Way, Matt Lantz takes a closer look at the new features of Laravel 10. Matt breaks down some of the potential impacts of upgrading and gives us a look at Pennant, the feature flag released in the new Laravel 10 package.

We're getting an introduction to application patterns from Chris Tankersley this month in Education Station. In his article, '12 Factor Applications: Parts 1-6', Chris discusses the history and the tenants of 12 Factor Applications—sharing how they can be applied today.

This month's PSR Pick Up column, 'PSR 17: HTTP Factories' by Frank Wallen, is taking a closer look at how to standardize the creation of an HTTP Request and Response. The factory will help to ensure that we are handling requests and various response types in a consistent manner.

Oscar Merida takes us back to school again this month with standard deviation. In his latest puzzles column, 'Grade Deviations', he explains how standard deviation is used to calculate grades.

In the column, Finally, this month, Beth Tucker Long brings us 'Nothing Lasts Longer', sharing some wisdom gained from tried and true experience. She shares how writing the 'ideal' code seems like the obvious goal; however, sometimes, the code that fixes the problem is the best choice.

Write For Us

If you would like to contribute, contact us, and one of our editors will be happy to help you hone your idea and turn it into a beautiful article for our magazine.

Visit <https://phpa.me/write> or contact our editorial team at write@phparch.com and get started!

Stay in Touch

Don't miss out on conference, book, and special announcements. Make sure you're connected with us.

- Subscribe to our list: <https://phpa.me/sub-to-updates>
- Mastodon: @editor@phparch.social
- Twitter: [@phparch](https://twitter.com/phparch)
- Facebook: <http://facebook.com/phparch>

Download the Code

Archive:

https://phpa.me/March2023_code

A Guide to Practical Usage of PHP FFI

Bohuslav Šimek

In this article, practical examples will teach you how to communicate with external libraries using PHP FFI. Together, we will navigate common challenges and explore the rising star of the database world, DuckDB.

PHP 7.4 introduced many exciting features, such as typed properties, preloading, and the foreign function interface (FFI). Some of these features have been highly anticipated, like typed properties, while others were more surprising. Regardless, it is clear that out of all the features mentioned, FFI is the least self-explanatory.

So, what is FFI? According to Wikipedia, it is “a mechanism by which a program written in one programming language can call routines or make use of services written in another.” However, this is just a book definition and doesn’t tell us what real-world problems it can solve. Therefore, a more relevant question to ask might be: What problems does FFI solve in the real world?

FFI can be helpful if you want to reuse code that has been written in a different programming language. For example, you might have a connector to a database that has not yet been ported to PHP. Another potential use of FFI is to speed up certain parts of your code. For instance, you might write an algorithm in a programming language that performs better for a specific task, such as completing complicated scientific calculations. Additionally, FFI can allow you to do things that are not supported in your language, such as communicating with hardware or directly accessing memory.

You may be wondering, “Can’t we already do all of this with PHP extensions?” The answer is yes. We can accomplish all of the tasks mentioned above using PHP extensions. However, FFI still has its advantages, such as easier usage, maintenance, deployment, and better portability across PHP versions. The main reason for this is that everything can be done in plain PHP, so there is no need to set up a compilation toolkit or change deployment procedures.

Note: All of the provided code samples require a minimum version of PHP 8.1 and an operating system of Linux or Windows with WSL. They cannot be run on standard Windows or Mac OS. The complete source code is available at <https://phpa.me/github-kambo>¹, along with a prepared Docker image.

First Steps

We can illustrate the basic usage of FFI by rewriting the `abs()` function, which returns the absolute value of a given number: (See Listing 1)

The first step is to create a proxy object between the library and PHP. One of the main challenges with FFI is mapping

Listing 1.

```
1. <?php
2.
3. $ffi = FFI::cdef(
4.     // function declaration in C language
5.     'int abs(int j);',
6.     // library from which the function will be called
7.     'libc.so.6'
8. );
9.
10. var_dump($ffi->abs(-42)); // int(42)
```

functions from one programming language to another. The authors of FFI in PHP handled this by parsing a raw C function definition. Therefore, the first argument of the `cdef()` function contains a function declaration in C language. The second parameter is the name of the library from which the function will be called. In this case, we are using functions from the standard C library.

But what if we want to use more functions? Do we have to put them all in the `cdef()` call? No, there is another way to load these definitions, which is by using the `FFI::load()` function. This function loads a file with all of the definitions and returns an instance of the FFI object. It expects just one parameter: the path to the so-called header file.

Header files are a concept that does not exist in PHP. Header files contain functions, declarations, structures, and macro definitions to be shared between multiple source files. This is important because C requires a forward declaration for all used functions, structures, or enums.

Function signatures in PHP and C are quite similar, as PHP is a C-style language. However, there are some differences:

- Variable types must always be defined.
- The return type is declared before the function name.
- Some data types are not supported in PHP and vice versa.

The function is then called through a method with the same name on the proxy object. Simple data types like `int`, `float`, and `bool` are automatically converted during the function call.

A header file can also contain macro definitions. In C, a macro is a fragment of code that has been given a name. Before compilation, the macro is evaluated and replaced by the code it represents. Macros are often used to define constants and create short, simple functions. However, FFI

¹ <https://phpa.me/github-kambo>: <https://phpa.me/github-kambo>

does not support C preprocessor directives, and they cannot be included in the header file loaded by FFI.

All of these things may seem complicated and a bit dry, so it would be better to demonstrate them using a practical example. One such example is DuckDB.

Duckdb

DuckDB is an in-process, column-based, SQL OLAP database written in C++ that has no dependencies and is distributed as a single file. It can also be described as “The SQLite for Analytics” with a PostgreSQL flavor. Its main strength is its columnar-vectorized query execution engine, which makes it particularly well-suited for performing analytical queries. However, DuckDB does not have direct support for PHP, making it an ideal use case for the PHP Foreign Function Interface (FFI).

Duckdb and FFI

If we are trying to use a new library, it is usually best to start with simple examples. This is also the case with DuckDB. In this example, we will:

1. create a database in memory,
2. insert some data and
3. query stored data.

The first step is to download a prebuilt library for Linux from the DuckDB website (<https://duckdb.org/docs/installation/>² or direct link <https://phpa.me/github-duckdb>³). The library archive will also include a header file with all function signatures and structure definitions. Unfortunately, we cannot use this header file directly, as it contains macros that PHP FFI does not currently support. We could manually evaluate these macros, but it would be more efficient to use a C++ compiler for that. Unfortunately, we still need to make a few changes. Inside the header file, there are three include directives:

```
#include <stdbool.h>
#include <stdint.h>
#include <stdlib.h>
```

All of these directives must be commented out or removed from the file. If we don't do this, the content of these files will be copied inside our header file. However, copying these files will not work, as they contain not only function signatures but also complete functions for inlining. After that, we are ready to let the C++ compiler resolve our dependencies. One of the most common compilers in the Linux world is GCC, so we'll use that for the example. There is no need to install this compiler; it is preloaded in the provided docker image.

The first step is to prefix the header file with the `#define FFI_LIB \"./libduckdb.so\"` directive, which tells PHP the location

of the library—the current folder. We can use the following command:

```
echo '#define FFI_LIB \"./libduckdb.so\"'>duckdb-ffi.h
```

Then we can use the following command for resolving the macros:

```
cpp -P -C -D"attribute(ARGS)=\" duckdb.h>duckdb-ffi.h
```

Now we are ready to call DuckDB! We will load our header file by calling the `load()` method and create two structures: database and connection—by calling the `new()` method (See Listing 2).

Listing 2.

```
1. $dbFFI = FFI::load('duckdb-ffi.h');
2.
3. $database = $dbFFI->new("duckdb_database");
4. $connection = $dbFFI->new("duckdb_connection");
5.
6. $result = $dbFFI->duckdb_open(
7.     null,
8.     FFI::addr($database)
9. );
10.
11. if ($result == $dbFFI->DuckDBError) {
12.     $dbFFI->duckdb_disconnect(FFI::addr($connection));
13.     $dbFFI->duckdb_close(FFI::addr($database));
14.     throw new Exception('Cannot open database');
15. }
```

Structures are like objects but without methods. We will discuss their specific usage in more detail later. However, we should talk about data types. Since we are calling the library with the C API, we need to use C data types. They can be roughly divided into three categories:

1. Basic data types (integer, float, etc.)
2. Extended data types (arrays, strings, and pointers)
3. User-defined types (enums, structures)

Basic data types are subject to automatic conversion and do not require any further action. Extended data types require more attention. They must be created with the `new()` method on the FFI instance. Finally, there are user-defined types, which must be defined in advance using the `cdef()` function or in the header files. We can then create them with the `new()` method, as we can already see in the case of the `duckdb_database` and `duckdb_connection` structures. Enums can be a little special, as they can be directly accessed on instances of FFI objects. In the case of DuckDB, we will encounter the `duckdb_state` enum with two possible values: success and failure.

² <https://duckdb.org/docs/installation/>

³ <https://phpa.me/github-duckdb>

```
typedef enum {
    DuckDBSuccess = 0,
    DuckDBError = 1
} duckdb_state;
```

Both of the values can be accessed on an FFI instance as dynamic properties. This will come in handy when calling DuckDB methods—we can just compare the returned value against the DuckDBError or DuckDBSuccess enum value.

Now we have all the information we need to open a connection to the database. We will do this using the `duckdb_open()` method. The method has the following signature:

```
duckdb_state duckdb_open(
    const char *path,
    duckdb_database *out_database
);
```

The method will return an enum value from `duckdb_state` representing the operation status. Fortunately, we now know how to handle this. The more interesting parameters are `path` and `database`. The first parameter, `path`, represents the file path to the database. Since we are creating our database in memory, we can safely ignore it and pass `null` as a value.

The second parameter is more interesting (See Listing 3). It clearly expects a `duckdb_database` structure, but the name of the variable is prefixed by an asterisk. This means that the value is expected as a pointer to the `duckdb_database`

Listing 3.

```
1. $result = $dbFFI->duckdb_connect(
2.     $database,
3.     FFI::addr($connection)
4. );
5.
6. if ($result == $dbFFI->DuckDBError) {
7.     $dbFFI->duckdb_disconnect(FFI::addr($connection));
8.     $dbFFI->duckdb_close(FFI::addr($database));
9.     throw new Exception('Cannot connect to database');
10. }
11.
12. $result = $dbFFI->duckdb_query(
13.     $connection,
14.     'CREATE TABLE integers(i INTEGER, j INTEGER);',
15.     null
16. );
17.
18. if ($result == $dbFFI->DuckDBError) {
19.     // Error handling, memory clean up
20. }
21.
22. $result = $dbFFI->duckdb_query(
23.     $connection,
24.     'INSERT INTO integers VALUES (3,4), (5,6), (7, NULL)',
25.     null
26. );
27.
28. if ($result == $dbFFI->DuckDBError) {
29.     // Error handling, memory clean up
30. }
```

structure. A pointer is a variable that stores the memory address of another variable located in computer memory. Unsurprisingly, PHP does not have pointers. The closest equivalent in the PHP world is a reference, but they are not the same. There is a whole page in the documentation about this: <https://phpa.me/phpManuel>.

In C, arrays and strings are always passed to functions as pointers. In the PHP implementation of FFI, pointers can be obtained only for C data types such as arrays, strings, or structures by calling the static `addr()` method. With all of this in mind, this is how the invocation of `duckdb_open()` should look:

```
$result = $duckDbFFI->duckdb_open(
    null,
    FFI::addr($database)
);
```

The next few lines are somewhat similar. After opening the database, we have to connect to it by calling the `duckdb_connect()` method with a pointer to the opened database. Now we can finally execute queries on the database with the `duckdb_query()` method. This method expects a connection structure, not a pointer, which can be rather surprising.

```
duckdb_state duckdb_query(
    duckdb_connection connection,
    const char *query,
    duckdb_result *out_result
);
```

Another interesting thing is the data type of the second parameter shown in listing 4, `const char *query`, which represents an immutable string containing a database query. Working with strings in C is not as straightforward as in PHP. Strings are passed to functions as pointers to the char type and implemented as an array of characters with a fixed size. PHP FFI will automatically convert PHP strings into C strings, but only for function parameters. Returned strings are not automatically converted.

Listing 4.

```
1. $queryResult = $dbFFI->new('duckdb_result');
2.
3. $result = $dbFFI->duckdb_query(
4.     $connection,
5.     'SELECT * FROM integers;',
6.     FFI::addr($queryResult)
7. );
8.
9. if ($result == $dbFFI->DuckDBError) {
10.     $resultAddr = FFI::addr($queryResult);
11.     $error = "Error in query: " .
12.         $dbFFI->duckdb_result_error($resultAddr);
13.     $dbFFI->duckdb_destroy_result($resultAddr);
14.     $dbFFI->duckdb_disconnect(FFI::addr($connection));
15.     $dbFFI->duckdb_close(FFI::addr($database));
16.     throw new Exception($error);
17. }
```


Thanks to this, there is no need for any further action when creating the integers table in DuckDB:

```
$result = $duckDbFFI->duckdb_query(
    $connection,
    'CREATE TABLE integers(i INTEGER, j INTEGER);',
    null
);
```

Inserting values will be pretty much the same—only the query should be changed to INSERT. So far, we have been executing queries and not caring about returned values. This will change with the next query, in which we try to select the inserted values. We will have to provide a pointer to the duckdb_result structure as the third parameter. After the execution of duckdb_query(), the duckdb_result structure will contain the result data and additional information about the result.

This brings us back to structures in PHP FFI. As we already mentioned, they are like objects but without methods (See example in listing 5). They have to be defined in advance, and their main purpose is to combine data items of different kinds. Properties of the structure can be accessed in the same way as an object property in regular PHP.

Listing 5.

```
1. typedef struct {
2.     // deprecated, use duckdb_column_count
3.     idx_t __deprecated_column_count;
4.     // deprecated, use duckdb_row_count
5.     idx_t __deprecated_row_count;
6.     // deprecated, use duckdb_rows_changed
7.     idx_t __deprecated_rows_changed;
8.     // deprecated, use duckdb_column_family of funcs
9.     duckdb_column * __deprecated_columns;
10.    // deprecated, use duckdb_result_error
11.    char * __deprecated_error_message;
12.    void * internal_data;
13. } duckdb_result;
```

To view the properties of a duckdb_result, we can refer to the header file and search for the definition of duckdb_result. Unfortunately, all the interesting properties have been deprecated in DuckDB version 0.3.2 in favor of individual functions. However, for the sake of example, we can still get the number of columns in the result by accessing the property __deprecated_column_count of the duckdb_result object: \$queryResult->__deprecated_column_count. There is no need to take any additional action when accessing this property; it will simply return the number two, as there are two columns. As mentioned before, simple data types are automatically converted.

Now, it's time to correctly retrieve the row and column counts, as we will need them eventually. The number of rows can be obtained using the function duckdb_row_count(), and the number of columns can be obtained using the function

duckdb_column_count(). Both of these functions expect a pointer to a duckdb_result object and will return integers. There is no need for any type conversion.

Another interesting function for working with the result structure is duckdb_result_error. It returns the error message in a human-readable format. We can try changing the SELECT statement into something invalid and see how DuckDB will inform us about the invalid query (See Listing 6).

Listing 6.

```
1. echo "Number of columns: " .
2.     $queryResult->__deprecated_column_count . "\n";
3.
4. $resultAddr = FFI::addr($queryResult);
5. $rowCount = $dbFFI->duckdb_row_count($resultAddr);
6. $colCount = $dbFFI->duckdb_column_count($resultAddr);
7.
8. for ($row = 0; $row < $rowCount; $row++) {
9.     for ($column = 0; $column < $colCount; $column++) {
10.        $value = $dbFFI->duckdb_value_varchar(
11.            $resultAddr,
12.            $column,
13.            $row
14.        );
15.        echo ($value != null ?
16.            FFI::string($value) :
17.            '')." ";
18.        $dbFFI->duckdb_free($value);
19.    }
20.
21.    echo "\n";
22. }
23.
24. $dbFFI->duckdb_destroy_result(FFI::addr($queryResult));
25. $dbFFI->duckdb_disconnect(FFI::addr($connection));
26. $dbFFI->duckdb_close(FFI::addr($database));
```

After populating the duckdb_result structure, we can finally print the data. However, we cannot directly traverse the result; we have to use specific functions again. We can get the data by using the duckdb_value_varchar() function. This function takes three parameters: a pointer to the result, the column position, and the row position.

The result of the duckdb_value_varchar function is a string that must be converted into a regular PHP string using the FFI::string() function and, more importantly, must be freed from memory by calling \$duckDbFFI->duckdb_free().

One of the most significant differences between PHP and C is the need to call a free function. This is the most complicated area of PHP FFI, as it is often not immediately apparent due to the different memory management models in the two languages. Please note that this article does not aim to provide a comprehensive description of the C memory model, as it is beyond the scope of this article.

PHP is a memory-safe language with a garbage collector, so we don't need to worry about how memory is allocated and freed. The only thing we usually need to consider is the memory limit defined in the PHP configuration. This is not the case with C, where we have to manage memory manually, use pointers, and manipulate memory directly.

Because FFI is just a bridge between PHP and C, we will also have to deal at least with some aspects of the C memory model. This is especially true for passing or returning arrays/strings to C functions. PHP FFI is trying, to some degree, to manage memory for us. For example, all complex data types created by FFI `new()` method are by default managed by PHP (in the FFI terminology, they are “owned”). So we do not have to care about their life cycle.

However, sometimes this is not enough. We need to be especially careful when dealing with pointers obtained using the `FFI::addr()` method, individual elements of C arrays and structures, and most data structures returned by C functions.

C pointers are always “non-owned.” This can easily lead to the creation of “dangling pointers,” which are pointers that do not point to a valid variable. This can occur when the variable being referenced is destroyed, but the pointer still points to its memory location.

One of the most complicated topics is probably handling complex data types returned by C functions. In C, values can be returned in two ways: by value or as a pointer. As previously mentioned, arrays and strings in C are always passed or returned as pointers. When working with these types, the most important question is: Who is responsible for allocating and deallocating the memory? In most cases, it is PHP FFI, but sometimes it is the called function.

The DuckDB function `duckdb_value_varchar` allocates memory without any cooperation from PHP, so after we use it, we have to free the allocated memory using the function `duckdb_free`: `$duckDbFFI->duckdb_free($value);`. Why do we have to do this? We don't know how long the returned string will be, so PHP cannot allocate the necessary memory ahead of time. This is the sole responsibility of the `duckdb_value_varchar()` function. Function `duckdb_free()` is the only reliable way to free this memory, as the authors of DuckDB provide it. Information about how memory is handled is usually described in the function documentation provided by the library author, and DuckDB is no exception. We can find information about memory handling in DuckDB documentation, specifically here: <https://phpa.me/duckdb-archive>⁴. We will create a memory leak if we forget to call this function.

After closely inspecting the documentation, we will quickly realize that the `duckdb_value_varchar()` function is not the only one that needs to be cleaned up after use. Some other steps must be taken for query results, as well as for structures representing a connection and database. Therefore, our example should end with calls to the `duckdb_destroy_result()` function to remove the result structure from memory and to



the `duckdb_disconnect()` and `duckdb_close()` functions. This concludes our short example.

Ending Notes

In this article, we have demonstrated that PHP FFI can be a powerful tool when interacting with code written in C. With just a few lines of code, we were able to connect to a database, insert data, and perform a full-fledged query. Previously, we would have had to create a C extension with a lot of boilerplate code to achieve this. However, using FFI does come with some challenges. We must be mindful of the differences in how PHP and C handle memory. We must also be aware of the similarities between the two languages, as they can create a false sense of security.

It may come as a surprise to PHP programmers, but C is extremely unforgiving. We can easily create problems for ourselves in a variety of creative ways, such as creating memory leaks, crashing the script, or bypassing most of PHP's security measures. Using FFI, especially FFI pointers, can even fundamentally change the way PHP works and lead to wildly unpredictable situations. Therefore, we must exercise caution when attempting to communicate with C code and be sure of our actions. It may be a cliché, but it is completely reasonable to say that with great power comes great responsibility.



Bohuslav Šimek works as a lead programmer at PeoplePath, where he is primarily responsible for the underlying architecture. In his free time, he is trying to apply Atwood's Law to PHP. This fruitless effort can be summarized with the following words: “Any application that can be written in PHP, will eventually be written in PHP.”
@BohuslavSimek

⁴ <https://phpa.me/duckdb-archive>: <https://phpa.me/duckdb-archive>

php[tek] 2023
**GET
YOUR
TICKETS
TODAY**

tek.phparch.com

May 16 - 18 2023

📍 Sheraton O'Hare Hotel



TIME IS RUNNING OUT

Purchase your php[tek] 2023 tickets now!

Problem in a Box

Edward Barnard

Here's an approach to software problem analysis and debugging that's based on intuition. Come up with theories about what might have gone wrong, then set out to prove or disprove each theory. This path, taken step by step, can lead us to the solution.

*How do we know when we have arrived at the solution?
That's the technique I call "fitting the problem into a box."
When the problem fits, with no loose ends hanging out,
we have almost certainly arrived at the right answer.*

What's the secret to debugging your software? Here's an approach I learned more than thirty years ago. It's an intuitive approach that serves me well.

Once More

I transferred to the Software Division of Cray Research in 1991. That's where I learned this technique. Our work group had a backlog of problem reports. By this point, the team had solved the easier of problems.

What could we do when we didn't have the answers? How do we analyze and fix something when we don't know what's wrong? In fact, we followed the advice of King Henry V as written by William Shakespeare. It's a useful approach:

*Once more unto the breach, dear friends, once more;
Or close the wall up with our English dead.*

*In peace there's nothing so becomes a man As modest
stillness and humility: But when the blast of war blows in
our ears, Then imitate the action of the tiger; Stiffen the
sinews, summon up the blood, Disguise fair nature with
hard-favour'd rage; Then lend the eye a terrible aspect;
Let pry through the portage of the head Like the brass
cannon; let the brow oerwhelm it As fearfully as doth
a galled rock O'erhang and jutty his confounded base,
Swill'd with the wild and wasteful ocean.*

*Now set the teeth and stretch the nostril wide, Hold hard
the breath and bend up every spirit To his full height. On,
on, you noblest English.*

Yes, really! The hard ones require that sort of determination.

I joined the Cray I/O Subsystem software group. The I/O Subsystem was built in-house, running a custom operating system, all written by our team. Thus, problems could be related to our hardware, software, someone else's hardware, software, or a combination of the above. We seriously (and correctly¹, as it turned out) considered cosmic rays as a possible source of failures!

The "easy ones" were long since resolved. Our problem backlog involved intermittent problems that we couldn't reproduce or otherwise analyze. The new guy (me) had just arrived. What could we do? "Once more unto the breach, dear friends, once more."

I was a fresh set of eyes. The pile of problem reports, dumps, history traces, and software listings was similar to this famous photo of Margaret Hamilton standing next to her team's Apollo Guidance Software² listings (Figure 1).

Figure 1.



1 correctly: <https://www.osti.gov/biblio/1014456-7ucdp9/>

2 Apollo Guidance Software: <https://phpa.me/DraperNewsReleases>

Loud Whiteboards

I paired with Alex who became my mentor. Together, we began attacking problems. There's always value in placing a fresh set of eyeballs on a situation!

Our most effective technique was to argue with each other. People could hear all the way down the hall that Alex and Ed were attacking another bug. We diagrammed our ideas on the whiteboard as to what the underlying problem might be.

I brought out my framed photograph of Mount Saint Helens erupting. The photo is similar to Figure 2. I put a label in front of the photo, "Backlog." It accurately portrayed the situation but perhaps not as tactful as it should have been.

Figure 2.



We picked a problem and did our best to diagram or describe the problem on a whiteboard. We listed ideas as to what each issue might be. Then we set out to prove or disprove that idea.

"But that means that..."

"No, that can't be it because..."

"In that case, let's look at this code over here. In order for that suggestion to be true, it would mean we *must* have executed

this piece of code over here. Did we?" We proceeded to find some way to prove or disprove that the code in question was, or was not, executed.

That in itself is a useful technique. Developing the sequence of events, or possible sequence of events leads to the reported problem. Match that to possible paths through the code. Attempt to prove that each particular path was, or was not, traversed.

Robert Frost describes this thought process in *The Road Not Taken*:

Two roads diverged in a yellow wood, And sorry I could not travel both And be one traveler, long I stood And looked down one as far as I could

Our analysis process turned out to be a surprisingly effective means of debugging. We put up ideas and shot them down. We were being assertive and passionate with each other but friendly. Sometimes we had to set the problem aside, still unresolved, until it happened again. Or, another possibility would occur days later, and we'd explore that possibility.

One advantage we had was complete control of the codebase. Every line had been written by our predecessors or by us. We had access to institutional knowledge from the first line of code to the present. Every problem ever fixed remained within living memory.

That complete control meant we had the luxury of adding diagnostic information to our codebase. We could add a patch to capture the necessary clue if we had a theory but could neither prove nor disprove it due to a lack of information.

Note the careful focus of this diagnostic information. We're aiming to *prove or disprove* a specific theory or prove that a specific path through the code *was or was not* taken. Robert Frost's traveler was considering which road to take. We, by contrast, seek to *identify* the road not taken. Even so, channeling Robert Frost or William Shakespeare adds fun to the often-grueling exercise.

Alex and I were using intuition rather than tools. With this sort of debugging, there just isn't any single or standard path to the solution. We got closer to the solution with each idea, sometimes proved, sometimes disproved, and sometimes left an open question.

How did we know if we actually *had* the solution? That's when Alex taught me the idea of fitting the problem into a box.

Fitting the Problem Into a Box

At some point, we came to a solution that explained *all* the symptoms. There were no remaining questions or inconsistencies. We had, in effect, built a box around the problem. We'd ask each other, "does the problem fit into the box?" That is, was there anything not explained by our solution?

From here, we searched for any possibility that we'd missed. Could we find or think of any possibility that our solution

might not fully explain? This brought another round of “What about...” or “But that would mean...” as we searched for further insight.

We often concluded, “this solution feels right.” It felt like we’d found the correct answer. “But wait,” you might ask. “Do feelings matter? It’s correct, or it isn’t.”

That’s the nature of intuition. When it “feels right” or “does not quite feel right”, that’s your intuition speaking.

If our solution (that is, our explanation of what we believe happened) explained *everything* we observed and discussed, that meant we had fit the problem into a box.

As you would expect, our next step was to fix or otherwise resolve the report. But, at this point, we were confident we were solving the correct problem. We all knew of countless cases where we’d fixed an issue only for it to turn up again with exactly the same symptoms.

We don’t get it right every time. We need to remain open to the possibility that we don’t yet have it right. “Are we there yet?” can be a tricky question to answer.

Task Main Loop

My best example of this technique comes from the same era. I’m using this example rather than a modern-era example because, in this case, once I realized what went wrong, all the symptoms fell into place. There was no question that I’d found the root cause of the problem. The failure “fit into a box” with no loose ends.

I transferred to Cray Research Software Training, intending to teach the operating system internals. I began by sitting through the two-week internals class as a student. The course materials included a system dump printout, in octal and ASCII, showing the state of the Cray operating system when it crashed. We knew this was an artificially induced crash for training purposes.

As students, we learned that each component (called a “task”) in the operating system contains a main loop that looks for work it needs to do. The

task “checks off” each bit of work as it completes it. Each bit of work is literally a bit within a 64-bit word. When the 64-bit word becomes a binary zero, all work is complete. Binary zero means all “to do” flags have been cleared, indicating no work remains to be done.

As the class progressed, I began to suspect the cause of the crash. Intuition told me that the operating system was executing code that should not be running. That code then induced the system halt (the crash) due to detecting invalid input.

I began to suspect that the right question to be asking was, “why is the operating system executing this piece of code?” As with Robert Frost’s, *Road Less Traveled*, I wondered why we were on the road that *shouldn’t* be traveled.

I can’t emphasize strongly enough that the “road” analogy is crucial to debugging. When you recognize the road that *was* traveled, keep going. Your problem analysis is getting close to the answer. When you later realize (or figure out) *why* this path choice was made “in a yellow wood,” so to speak, continue onward. Find out where it leads.

Bilbo the Hobbit, in *The Return of the King*, takes up our process:

The Road goes ever on and on Out from the door where it began.

Now far ahead the Road has gone, Let others follow it who can!

I asked myself, “why is the operating system executing this piece of code?” That led me to the next question, “what sequence of events would bring us to this piece of code?”

That question led me to an answer. The code in question could *only* be executed as a result of the “work to be done” flag being set.

Unfortunately, the answer brought me to a standstill. The system was idle; there was no work to be done. So far as I could find, there was no reason for the “work to be done” flag to have been set. Yet here we were. The impossible had happened. That, in fact, was why the

system halted. It correctly detected the impossible set of circumstances.

I set out to prove or disprove my theory. Could I prove or disprove that we were there as a result of the “work to be done” flag being set? I believed so but still needed to prove it.

I searched out the memory location containing the “work to be done” flags. That memory location showed something odd. The dump printout showed part of the memory word had ASCII text. It shouldn’t.

Was I in the right place? I re-checked everything I’d examined up to that point, proving to myself that I was indeed in the right place—even though what I saw there looked wrong.

Have we found our solution? No! We’ve not yet fit the problem into a box. We have not explained why the “work to be done” location looks like ASCII text.

Have you heard of filesystem magic numbers³? The first few bytes of a file generally indicate the file type. In the same way, the first few bytes of the Cray operating system memory-resident tables contain the table name (as an abbreviation) as ASCII text.

This mysterious ASCII text in the dump printout looked like a partial table name. I had the list of tables in the three-ring binder in front of me. This was long before Google! One or two table names looked like possible fits, but the ASCII text was corrupt. It did not include the full name of *any* known table as it should. “As it should”, meaning if my newest theory was correct.

Corrupt ASCII text, part of a table name, where it does not belong? I had not yet fit the problem into its box unless I could prove how the corruption occurred. Indeed, I could be completely on the wrong track.

I thought about this impasse. What if the ASCII text *did* get placed there intact but then interpreted as a set of “work to be done” flags? That was just *wrong*. But is that what happened?

I wrote down the binary ones and zeroes of what the ASCII text *should*

³ <https://phpa.me/wikipedia>

have been and compared that result to what I saw in the dump printout. If my theory was correct, no zeroes would have become ones, but some ones would have become zeroes.

Do you see how my narrow focus aimed to prove or disprove my latest theory? The Task Main Loop *clears* “work to be done” flags but never *sets* those flags. I followed the Task Main Loop through the entire loop to verify that this was, in fact, the case. Therefore, *if* the ASCII text corruption resulted from the Task Main Loop flipping bits, the flips would *only* be from one to zero.

Sure enough, that *was* the case. The presumed original ASCII text had bits subtracted (flipped to become zero) but no bits added (flipped to become one). My theory was that the operating system task interpreted the ASCII text as “work to be done” flags. Each flipped bit represented work that had been done.

I now had a precise list of flipped bits because I had already “done the math” to prove or disprove my theory. Each flipped bit represented a specific code path, a specific case of the Task Main Loop doing requested work. I confirmed that each attempted work item was recorded in the event-history trace.

Yes, I was relentless, carefully proving-out every single step! That’s why we opened with Shakespeare:

*Then imitate the action of the tiger; Stiffen the sinews,
summon up the blood, Disguise fair nature with hard-fa-
vour’d rage;*

So far, everything was proving out. The problem was beginning to fit into the box. But we weren’t to the finish line yet! We had simply reached the next question: How did that ASCII text get placed where it didn’t belong?

I theorized that some task must have created the in-memory table and, for some reason, written it to the wrong place in memory. I searched through the operating system and found the section of code creating that in-memory table.

Boom! There it was!

The table was supposed to be stored at an offset from the beginning of the operating system task’s working area. Some evil genius added a line of code that cleared the offset to zero. The offset-from-zero just happened to be the other task’s “work to be done” location.

I could now explain every point of interest in the dump printout. I could explain how the ASCII text got into that memory location (due to an evil genius adding a line of operating system code). I could explain the actual crash (due to treating the ASCII text as a collection of “work to be done” flags).

Most importantly, I could explain why the ASCII text was oddly corrupted. It’s because the task main loop flipped each “flag” bit as it attempted to perform the “requested” work. The operating system was blindly following instructions until it discovered its input data were so invalid that it could not safely proceed.

The problem “fit into the box” with no loose ends.

In this case, once I found the root cause (an evil genius) and the sequence of events leading to the crash (flipping bits of ASCII text), it was easy to confirm that this was, in fact, the solution. I showed the class instructor, and he confirmed that I’d found the cause and had correctly inferred the sequence of events.



Summary

We explored some of my intuitive methods of problem analysis and debugging. I come up with theories as to what might or might not have happened. I then look for some way to prove or disprove each theory.

We focused on Robert Frost’s “road less traveled” as a way of visualizing this process. Prove or disprove the path taken through the code. If you cannot find proof one way or the other, consider: what would provide proof? Can you adjust your code base to capture and report the necessary information next time around?

How do you know when you’ve found the right solution? As your intuition develops, your solution will often “feel right” or even feel “not quite right, but close”. That’s your intuition talking to you.

I call this process as a whole, “fitting the problem into a box.” If the problem fits, with no loose ends, there’s a great chance that we’ve found the correct solution. If any aspect, no matter how minor, remains unexplained, there’s a good chance that we have *not* yet arrived at the solution.



Ed Barnard had a front-row seat when the Morris Worm took down the Internet, November 1988. He was teaching CRAY-1 supercomputer operating system internals to analysts as they were being directly hit by the Worm. It was a busy week! Ed continues to indulge his interests in computer security and teaching software concepts to others.
[@ewbarnard](#)



12 Factor Applications: Parts 1-6

Chris Tankersley

As developers, we love being able to find patterns in the things we do. We can turn these patterns into rules we invariably learn, use, and eventually modify to work for us. A few times I have talked about design patterns, which describe how different patterns in code look and should be structured. Today, let's take a look at application patterns.

One of the most interesting application design patterns I have encountered is the idea of 12-Factor Applications. This is less a pattern than a collection of patterns and ideas that, collectively, are designed to help developers build more robust architectures that are maintainable and scale well. The tenants are language agnostic, and the ideas are flexible enough to add them to applications gradually.

12 Factor Application design became a very viable set of ideas as software, especially web software, moved to more distributed deployments. As containers and adjacent tooling became more popular, monolithic application structures became harder to work with. This was not because of a failure of monolithic architecture itself, but the shift to deployment motifs like Kubernetes forced developers to change how they thought about code architecture.

Adam Wiggins presented 12 Factor Applications in 2011. Developers drafted it at Heroku¹, a Platform-as-a-Service (PaaS) hosting company. The ideas presented in 12 Factor Applications were heavily geared toward software-as-a-service applications. Heroku specialized in allowing developers to deploy code in application containers. In fact, nginx engineers released what they consider a more generalized Microservice Reference Architecture² in 2016, using 12 Factor Applications as a basis but altering it to be more general.

I agree with the nginx engineers that some ideas, viewed through the lens of someone not deploying to Heroku or

another PaaS, may not jive well with microservices in general. I think it's important to see where many ideas that we use come from, and even in 2023, 12 Factor Applications is brought up as a way to structure applications.

Let's look at these twelve factors, how they relate to PHP, and what compromises we make in 2023 to stick to these sets of ideas. Take the ideas as a whole, as many of the tenants set a basis for later ones.

1. Codebase

"One codebase tracked in revision control, many deploys"

When the 12 Factor Applications methodology was designed in 2011, it was still somewhat common to run into applications not being stored properly in version control. I remember moving many client applications to Subversion to help better track what we were delivering and make it easier for developers to share code. The world has changed a lot since then.

Your application should be stored as a single codebase in a single repository to help facilitate many of the other methodologies of 12 Factor Applications. This repository will be the basis later for what you deploy to your different environments, development, quality/staging, or production. The single repository will be a source of truth for the status of any application.

As an application becomes more complicated, this idea can be somewhat hard to deal with. When some developers take into account "the application," this can actually be various subsystems that work in tandem. Microservice

architecture is becoming increasingly common, so how do we rectify this?

If you have multiple subsystems, you treat them all as separate individual systems. If you have a microservice architecture, you keep each service in its own repository and apply the 12 Factor methodology to each system as an enclosed system. As your overall application grows and new systems are added, they are all treated the same.

This helps enforce one of the tenants of keeping things in separate repositories—multiple apps sharing the same code is a violation. You do not want one system to depend on the code inside of another. This will make it harder to refactor the shared code, and you end up coupling potentially disparate apps together for no reason other than taking a shortcut.

If you need to share code between systems, you should extract the code as a new package and treat it as an independent dependency. This new package will be a new codebase and will be treated with the same workflows as any other codebase.

Microservices and Monorepos

One wrinkle that tends to show up with this idea and modern applications is the idea of a mono repo. For those that are not familiar, a monorepo is a structural setup where you keep all your codebases in a single repository. Doing so allows teams to work on larger projects in smaller sub-projects and deploy those sub-projects separately. It is used many times in conjunction with complex microservice architectures.

While you could make an argument that monorepos do not violate the "one codebase to a repository," as all your

1 Heroku: <https://www.heroku.com>

2 Microservice Reference Architecture: <https://phpa.me/nginxMicroservices>



code is in one repository and can be deployed as such, I feel it can become a maintenance nightmare for tracking commits to deployments in an easy manner. Many monorepos also end up being tightly coupled between the various sub-projects, and you find yourself having to release multiple projects simultaneously to avoid breaking deployments. Congrats, you now have a distributed monolithic application!

This is because 12 Factor applications work on the assumption that a codebase can be tracked in the version control system, and there is a direct matching between a commit (or, more often, a tag) and a build and release. By physically separating codebases into different repositories, it makes it less likely any individual system will be hard-linked to another. It will also be harder to share code between repositories since you will have to extract said shared code and declare it as a dependency. There is a bit more thought that goes into these types of coupling.

If you do need to have separate projects working in conjunction, there are ways to take many repositories and turn them into one. `git submodules`³ can create one repository that pulls in information from other repositories. You will end up with the same basic idea of a monorepo, but you are still enforcing some separation. This might be the best option if you have an overall system comprising various languages.

Another option is a repository that stores just package dependencies and pulls them all in. For PHP, this means a repository that is essentially just a `composer.json` file hard-locked to specific package versions. When you do a new release of a sub-project, you go into this dependency repo and update `composer.json` to the new file.

2. Dependencies

“Explicitly declare and isolate dependencies”

While Dependency Injection is an important part of clean and maintainable code, for 12 Factor applications, we are talking about the third-party code your own application relies on. Applications should specify in code: what the dependencies are, what version your application is programmed against, and where to download the dependencies.

Doing so allows developers to ensure that they always download the appropriate versions of dependencies when working locally—or that your build process can be reproducible by always downloading the same versions. Some sort of package manager can also be employed to keep dependencies up-to-date.

The other important detail is that this code should not be intermingled with your own codebase. A clean physical separation of the code makes it much easier for developers to distinguish third-party code, and since we are using version control, it is easy to ignore these files. We do not need to store them if we can redownload them.

Most modern PHP applications use Composer⁴ to manage their dependencies, which pretty well satisfies this part of the methodology. It keeps everything in a `vendor/` folder, which is easily ignored in a version control system like `git`. By putting everything in a separate folder, it is easy to visually distinguish this third-party code from your application code. Composer can also keep things up-to-date when new versions of dependencies are released.

3. Config

“Store config in the environment”

Configuration should be supplied by the environment and not part of the actual application deployment. This means that things we consider mutable between installs, like database connection information, access keys, or even file paths, should be considered information that the environment supplies to us. In the ideal situation, there is no configuration file as part of the deployment either. Our application asks the environment for the configuration.

When it comes to things like routes, dependency management wiring, or something that would not change environment-to-environment, those should be part of the application code. While you may have additional routes for debugging in a development environment, you can put those behind flags to stop them from functioning in production.

In 2023, this is not so much a foreign concept anymore. We live in a world where we regularly develop locally and push to remote systems. When the 12 Factor Application design was released in 2011, it was much more common for work to be done on remote systems or situations where we were solely moving from local development to a single production system.

In PHP, you can ask for the environment variables by calling the `getenv()`⁵ function. It takes the name of the environment variable you want to access and a second boolean parameter that limits the return values to those set by the local environment (like the operating system). If you do not pass any name, it will return an array of all the available environment variables.

You can also access these values by looking at the `$_SERVER` global variable. This is an associative array with the same information that should be available via the `getenv()` function.

No matter how you access variables, the overall idea is that the environment supplies this information. If you use Apache's `httpd`, this can be done by setting environment variables in the `VHOST` config or the `.htaccess` file. You can do this by using the `SetEnv` directive:

```
<VirtualHost hostname:80>
    # Other stuff like rewrite rules and docroot

    SetEnv VARIABLE_NAME variable_value
</VirtualHost>
```

³ `git submodules`: <https://phpa.me/git-tools>

⁴ Composer: <https://getcomposer.org>

⁵ `getenv()`: <https://phpa.me/php-manual>



In nginx it is similar. You can set a `fastcgi_param` for the location settings:

```
location / {
    fastcgi_param VARIABLE_NAME variable_value;
}
```

If you are deploying to bare metal servers, your provisioning system should gather this information and set it in the appropriate place. Tools like Ansible, SaltStack, Terraform, or Puppet can push configurations and the associated system files modified with the environment variables available for applications.

If you are deploying with containers, your deployment environment should have a way to set these in the container configuration. Kubernetes supports an `env` setting for pods, and Docker Compose allows an `environment` setting in the YAML config file.

4. Backing Services

“Treat backing services as attached resources”

Now that we’ve separated our code from any configuration that might change, we need to look at the services our application talks to. This includes things like databases, caches, third-party APIs, and anything that our application needs to utilize but is not part of our code. We should treat all of these external services as if they were third-party services.

Our application should be able to swap out these external services as needed. We mentioned in the Config section that the environment should supply any mutable configuration changes, including access to our external services. Swapping between various instances of these external services should be as seamless as just changing a config option.

Let’s take a look at our database. We should always connect to things like our database services via config-supplied DSNs (Data Source Names). Overall that is not so much a revolutionary idea. We take this further by writing our code so as not to care how the database is configured from a deployment standpoint—all that it cares about is that it is a MySQL server (for instance).

Your application should not care how many instances of the database there are, how it is shared, or anything above and beyond how to connect to it. There are some caveats, like it may be a good example to define a read connection alongside a write connection. Still, switching between environments should be no more complicated than a configuration change. A local MySQL instance in a container for local development, a hosted service for QA, or a multi-zone RDS Service on AWS should not impact how you write your code.

Along those same lines, we should treat any attached service as a failure point. Our application should handle when the database server is unavailable and code defensively around it. If we are using a Mastodon instance to grab social media posts, we should prepare for when we cannot connect. If we take into account the configuration being supplied by the environment, and appropriate dependency injection,

our application should be able to swap between compatible systems or handle missing systems as best as possible.

There is a limit to this. You do not need to be as drastic as “support all databases” and support both Postgres as well as MySQL, but you should be able to swap between any MySQL instance with nothing but a DSN change.

5. Build, Release, and Run

“Strictly separate build and run stages”

We have covered where your code is stored. We covered how to get configuration into your application. Now we need to look at how to actually get your code into whatever environment you want to put it in. The proper way to do this is to have a discreet build, release, and run segments of the overall release process.

The concept of a “build” process may seem a bit odd to many PHP developers. We never have to compile our software, but if we look at what we store in our version control system, we lack things like dependencies. We also tend to have very rich front ends for many applications, so we need to compile front-end assets like CSS files or JavaScript components.

Your build process should check out the source code from the version control system, download any dependencies, and compile any needed assets. The outcome of this is called a Build Artifact and has everything that you would need to deploy to a system. This Build Artifact can be stored somewhere, like an S3 Bucket, and kept for some time. Doing so allows anyone to grab an artifact at a later date for debugging.

If you are deploying to containers, your Build Artifact may be a container. After you have grabbed all your dependencies and built all your assets, the final build step may be to run something like a `docker build` command to build a container. In this case, your Build Artifact would be the resulting image; you can store that in your registry.

After you have built your application, the Release step will take the artifact that was generated and deploy it. If you are deploying to bare metal, you may use something like Capistrano⁶ or Deployer⁷ to move code to the remote server. A good deployment tool will allow you to keep at least one previous deployment available to roll back to a known good state if there are problems.

If you are using a system like Kubernetes, you can use its Deployment process to roll out updates to pods. As long as you do not remove previous images, you can roll back the pod configuration to a previous good container image and be back up and running if there are any problems.

Most PHP applications do not have a true “run” step, which is the final process of this specific idea. Unless there have been configuration changes, things like httpd or nginx do not need to be restarted, and most containers are set up to run the needed command when they start. If you have long-running

⁶ Capistrano: <https://capistranorb.com>

⁷ Deployer: <https://deployer.org>



processes like ReactPHP async daemons or subsystems written in other languages, you may need to restart these.

Where this practice really shines is in the ability to triage bugs. Since a release can be traced back to a build, it allows a developer to pull down the build artifact locally and run the exact same code that is in production. This becomes even more important if the application can have different versions running concurrently, like a self-hosted application used by many end-users or different installations per customer.

6. Processes

“Execute the app as one or more stateless processes”

In tenant #1, I and 12 Factor Applications argue that codebases should be stored as separate repositories. Especially as we move to a microservice architecture, we want to treat each service as its own thing. This means we treat each service as a process that can be scaled separately from other processes, and since our services are geared toward the web, we want to keep everything stateless.

If you are already doing microservices, congratulations! You are currently taking this factor into account... assuming you are building a proper microservice and not just a distributed monolith. If you want more information about microservices and monolithic applications, check out the January 2023 and February 2023 editions of *php[architect]*, where I go into more detail about building microservices.

For 12 Factor Applications, they do take into account simple deployments by just suggesting launching your app as a script all the way up to using container orchestration tools like Kubernetes. For PHP developers, we can somewhat skip this idea as we tend to run our applications either inside a web server, like Apache httpd, or as a FastCGI process. We let those managers handle requests.

If you are using long-running daemons like something based in Swoole or ReactPHP, you may need a script that starts these applications. Applications like these normally handle their connections, bypassing the normal web server or FastCGI setup. These types of PHP applications will have additional considerations to their internal architecture, as they function differently than a procedural PHP application.

If you are building an application using something like ReactPHP or Swoole, you will want to think about any state your application keeps and how things will react if you run multiple instances of these applications. We want our application to be horizontally scalable, so if two users hit two different instances of our application, how do we want that? What happens if one node goes down, and one of those users now has to access the other instance?

Things that we take for granted, like sessions, need special attention as we have to move them to a distributed storage mechanism like a database. The same goes for file storage. If a user uploads something like an avatar image to one instance, how do you make that available to any other instances?

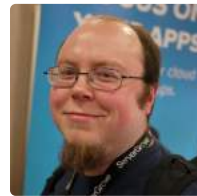
Factors 7-12

There are six more factors for us to review. Factor 6 started to venture away from the technical details around building and deploying and gives us a glimpse at some of the business logic and code architectural ideas we need to consider with modern distributed applications.

I hope you look forward to next month, when we continue to go through the methodologies and how they may still apply in 2023!

Related Reading

- *Education Station: Background Queues* by Chris Tankersley, January 2022.
<https://phpa.me/background-queues>
- *Education Station: Async is a Lie* by Chris Tankersley, February 2022.
<http://phpa.me/2022-02-education>



Chris Tankersley is a husband, father, author, speaker, podcast host, and PHP developer. Chris has worked with many different frameworks and languages throughout his twelve years of programming but spends most of his day working in PHP and Python. He is the author of Docker for Developers and works with companies and developers for integrating containers into their workflows. @dragonmantank



InfoSec 102: Phishing

Eric Mann

Continuing on last month's trend, we want to spend some time defining and explaining some of the terms and jargon frequently used by practitioners in the security community. Fortunately, this month's term is likely one you've already come across in business: phishing.

On the Hook

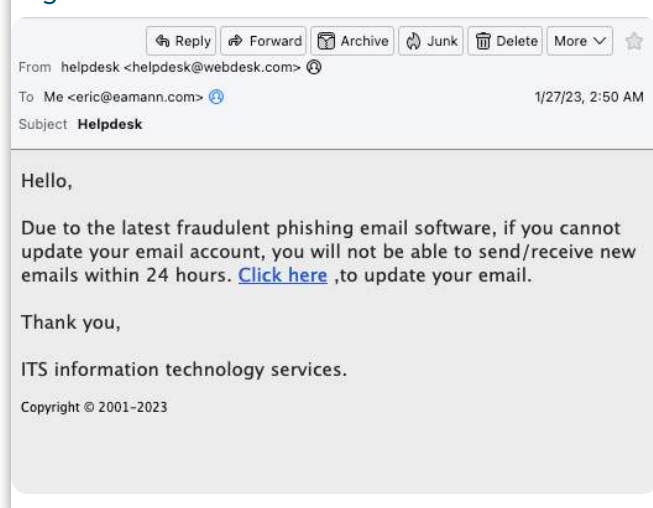
One of my favorite summertime activities is fishing. I enjoy my time on the pier, sitting back and waiting for a fish to strike my hook. On a day when I *actually* want to catch something, I might try a lure or a fly—anything to actively trick a fish into thinking my bait is food so I can take something home for my kids.

In technology, the attackers see all of us as fish. They're working hard to craft legitimate emails and websites but are nothing more than my lure to a fish—bait. The goal is to get us to click a link, download a file, or fill out a form with sensitive information they can use to enact further harm on our teams. All of these attacks are forms of "phishing"—a wonderful, web 2.0 way to write "fishing" but feel cool for using a phonetic "ph" instead.

Each phishing attack leverages social engineering to trick you into aiding the criminals in their attack against your team. Some attacks are remarkably sophisticated, but most are novice-level attacks hoping to gain a foothold in whatever system they can. In any event, a phishing attack will be just the beginning of an assault on your team, should it succeed.

Every organization will have to deal with phishing at some point in time, so it's important to understand the different ways the attackers will try to get your team on the hook. The most basic and frequent means is with an "email full" or "email suspended" notice. The attacker impersonates your own IT department and tries to trick you into providing your email credentials to avoid your account being shut down.

Figure 1.



Most corporate spam filters will capture and remove phishing messages automatically. However, more sophisticated attacks intentionally work around these filters and might land in your inbox regardless.

"Spear Phishing"

Most phishing attacks are scattershot. The attackers build out a campaign and blast it randomly to as many targets as possible, hoping one or more unsuspecting victims fall for the plot. These attacks are not very sophisticated and yield primarily low-value targets for the malicious parties running them.

In other scenarios, the attackers will identify a more specific target. This might be a particular executive or a key department. Focusing on these high-value targets often requires more finesse, as these individuals are usually on the lookout for phishing-type attacks.

In late 2022, Uber was breached not by a random phishing attack but by a focused breach against a particular external contractor¹. A hacker group gained access to this contractor's credentials on the dark web, then began logging into their account repeatedly, prompting a barrage of two-factor login prompts to the attacker.

The hacker group then posed as Uber's own IT support team and successfully phished the contractor's two-factor authentication token. From there, they had a full run of Uber's systems, including *complete administrative control* over the key IT infrastructure.

Recent Evolutions

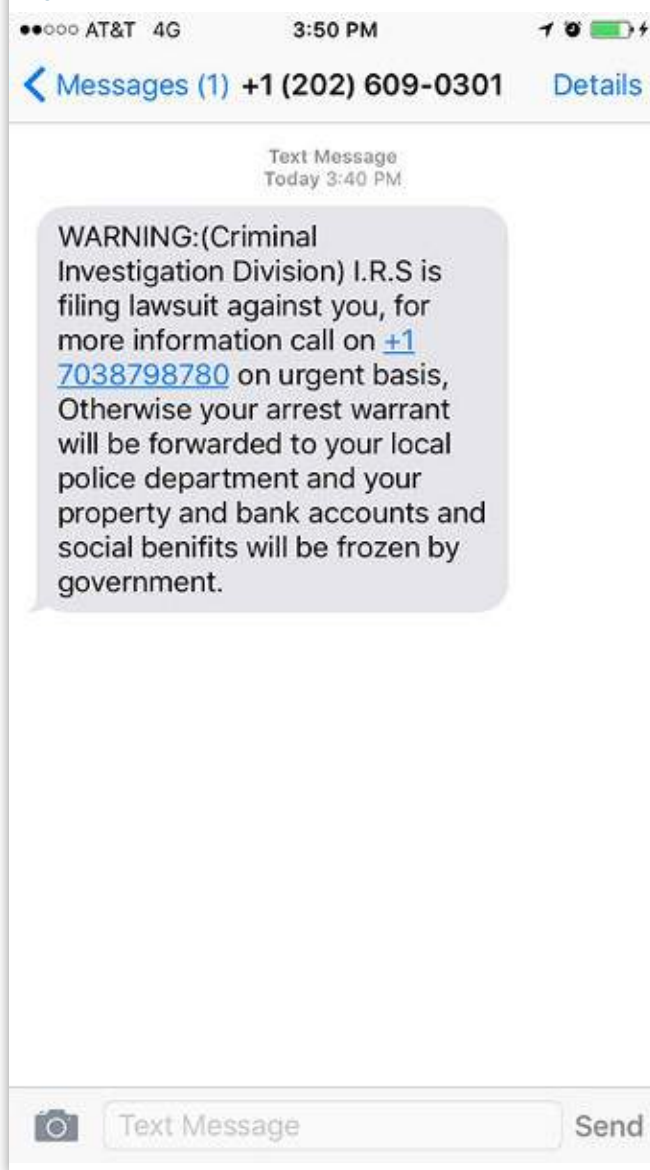
The most frequent phishing attacks come over email, thanks in part to the difficulty most organizations face when securing their email systems. However, newer techniques have begun leveraging additional channels to attack targets.

When attackers move to SMS rather than email, the attack is called "smishing." In these scenarios, the attacker impersonates a business, government entity, or team leader trying to get the victim to act on their behalf. Common requests might be to call a number to provide information, to pass along a 2FA token or password, or even to purchase gift cards on the attacker's behalf.

¹ focused breach: <https://phpa.me/darkreading-attacks>



Figure 2.



Even more sophisticated attacks leverage voice. Imagine you get a late-night voice message from your boss. Things are going south quickly at work and you need to urgently wire money to a vendor to prevent the loss of a major contract. You don't recognize the phone number, but your boss explains that she's lost her phone and is using a friend's. Since it's not email or a text, but a voice you know and trust, you're more likely to act without verifying.

Traditional “vishing” attacks include any voice-based social engineering attack. For example, someone calling claiming to be from your bank and asking to verify a charge—when you don't recognize the charge they then ask for sensitive details to help contest it and return the money. In reality, they're a scammer hoping to trick you into providing details similar to a standard phishing or smishing attack.

This kind of attack is a form of “vishing” or voice-based phishing. Thanks to advances in artificial intelligence and media creation, it often leverages a known party's *actual voice* to record a fictitious message aimed at defrauding victims. It's utterly terrifying to experience and incredibly difficult to protect against. For now, it's also among the rarest means of phishing; this could easily change as voice spoofing technology becomes more readily available.

Don't Get Phished

There are three things to look out for in any unexpected communication:

1. The sender doesn't know you personally—“Dear account holder”
2. There is an artificial sense of urgency—“please respond immediately”
3. A threat of negative repercussions—“your account will be closed”

If the email is unexpected but appears to come from someone you know, reach out to them directly to confirm its validity. For a text message, call your boss. For a voicemail, call them back on a known number or check with another member of the team. You won't lose your job for practicing diligence and validating the request.

Remember to never *reply* to a phishing email. Your new message might inadvertently lead someone else to click a link or download a malicious attachment. Don't become a “confused deputy”—take a few minutes to review last month's discussion² instead.

Everyone who has an email will eventually become the recipient of a phishing attack. Know what to look out for and stay alert so you can avoid becoming a *victim* when the attack comes through. Know also how to avoid smishing, vishing, and any other form of “ishing” that comes up as technology continues to evolve around us. Your discipline and vigilance will help you stay one step ahead of those who want to destroy your applications.



Eric is a seasoned web developer experienced with multiple languages and platforms. He's been working with PHP for more than a decade and focuses his time on helping developers get started and learn new skills with their tech of choice. You can reach out to him directly via Twitter: @EricMann


² last month's discussion: <https://phpa.me/security-jan-2023>



What Ben Uses

Ben Ramsey

A popular trend among developers these days is sharing the specs of one's local development environment and listing their most often-used tools. In this month's Workshop, guest columnist Ben Ramsey stops by to share what's running on his machine and the tools he finds most helpful in his day-to-day work.

 *Editor Note: Joe Ferguson, the regular author of this column, had to step away for a few months. A monthly release would not be the same without a Workshop in it, so we reached out to a few of our friends to cover until Joe gets back.*

This month is Ben Ramsey. Ben has been passionate about PHP and the PHP Community for years. He is a PHP Release Manager for 8.1 and 8.2, and this month he shares with us the tools he uses every day. Enjoy—Eric

I've been programming for over a quarter of a century, and this qualifies me to say that the only good development environment is *your* development environment. Not some Twitch streamers. Not some podcasters. Not some Internet-famous thought leaders. Not some release managers of a major open source project. *Yours*. You're the only one who knows what works best for you.

Still, there is value in learning about others' development environments and tooling. We might learn of tools we didn't know existed or see things configured in ways we've never considered. Adopting these and making them our own might lead to more efficient development habits, or they could cause a lot of frustration and pain as we try to integrate something that's just not working well for us.

So, take this advice with a healthy dash of salt and know that if something's not working for you, there's no shame in dropping it. As Mary Schmich once wrote, "Advice is a form of nostalgia.

Dispensing it is a way of fishing the past from the disposal, wiping it off, painting over the ugly parts and recycling it for more than it's worth."¹ Besides, this article will be out-of-date tomorrow because I'm always changing my mind.

Hardware

I've used Apple hardware exclusively since 2005. Prior to that, I ran Fedora Core on a laptop since its first release in 2003, and before that, I used Windows XP (and ME, 98, and 95). My transition to Apple resulted from their successful marketing campaign promoting OS X as a true Unix. I wanted to run a *nix, but I grew tired of compiling the kernel to support my wireless card's proprietary drivers (it was a Proxim ORiNOCO PC card), among other things. With Apple, the hardware just worked, as Steve Jobs was known to say.

I'm writing this article on a 14-inch, 2021 MacBook Pro with an M1 Max processor, 64 GB of RAM, and a 1 TB hard drive. This is my primary development machine. I recently switched from a 2019 MacBook Pro, and the difference is remarkable. I like to run my laptop in closed-clamshell mode.² It sits in a vertical laptop stand at the back of my desk, behind my monitors, to help with heat dissipation, but with the M1 Max, I rarely hear the fans.

The monitors are two 27-inch LG UltraFine 5K displays (the 27MD5KA model). These displays are gorgeous, but after five years of constant use, they're beginning to show their age. There are screen burn-in artifacts. The burn-in isn't permanent, but I'm in the market for replacements. I'm unsure what's next, though—these monitors spoiled me. I'm open to recommendations.

My keyboard is the Keychron K3. I switched to it a few years ago because I wanted to understand the draw of mechanical keyboards—it seemed like all the *cool kids* were using them. After several years of use, I'm thinking about switching back to the Apple Magic Keyboard. I keep pressing the wrong keys. My fingers often slip or get caught between (or even under!) keys. I'm not sure what the problem is, but my experience with a mechanical keyboard hasn't been great. I'm also open to recommendations for keyboards, though I love the keyboard on this MacBook Pro, so I might stick with the Apple keyboards.

My pointing device is the Apple Magic Trackpad. I love gestures and use them often to switch between spaces, windows, and a lot more. I don't feel comfortable using a regular mouse anymore.

I also have an iPhone 13 and an Apple Watch, which are both a part of the hardware I use daily for my work. I know I sound like an Apple fanboy, but I do like their products—I also understand if you don't. Use the hardware you like and are most comfortable with.

¹ Schmich, Mary (June 1, 1997). "Advice, like youth, probably just wasted on the young." *Chicago Tribune*.

² Closed-clamshell mode (also known as closed-display mode) is when you run your laptop with the lid (or top) closed. (My son was very confused by my calling it a "lid.")



Software

I use a lot of software for my daily work, including open source development.

Editors

First, let's talk about the tools I use to write code. I've tried a myriad of editors and IDEs over the years. All of them are awful; nothing is perfect.

For most of my career, I used Vim within a terminal. I made numerous attempts at IDEs, but nothing stuck. Nothing felt comfortable—until five years ago. Maybe it's my aging mind, or maybe I finally got over some mental hump. I saw the value in using an IDE after making sense of an open source Java project because I could navigate the symbols. The effect was like a light bulb turning on. Later, I changed jobs and decided to use PhpStorm, which made onboarding and ramping up much easier, especially on an older and sprawling legacy codebase. I've been using it ever since.

In addition to PhpStorm, I still use Vim. In fact, I use the Vim keybindings in PhpStorm. JetBrains have done a good job with the implementation, and much of the `~/.vimrc` configuration will work in `~/.ideavimrc`, which is what the IdeaVim plugin for JetBrains IDEs uses. I also use TextMate, though I primarily use it as a copy-paste scratchpad. I'm trying Fleet as I write this article, the new lightweight text editor from JetBrains that's meant to compete with Visual Studio Code; I'm not yet sure how I feel about it.

Rounding out the code editors, I really like the JetBrains products. In addition to PhpStorm and Fleet, I have installed IntelliJ, WebStorm, CLion, and GoLand, and I use them each to varying degrees. CLion has helped me feel much more comfortable with the PHP core source. The *All Products Pack* from JetBrains is worth it. (No, JetBrains are not paying me to say this.)

Command Line Tools

The next collection of software that plays a major role in my daily work is related to the terminal. The terminal emulator I use most is iTerm2, though I have been playing with Wez Furlong's WezTerm.

I use Z shell (Zsh) with Oh My Zsh, but I keep the number of plugins low, so I don't slow down the shell. I use Homebrew for most tools, but I also use asdf for multiple versions of certain tools, like NodeJS and PHP.

Here's an important point regarding Homebrew: I try installing *all* software through Homebrew, command line and GUI applications alike. This makes it easy to keep up with everything installed on my system. For example:

```
# Install command line tools with Homebrew:
brew install asdf
brew install starship
```

```
# Install GUI tools with Homebrew:
brew install --cask iterm2
brew install --cask jetbrains-toolbox
brew install --cask firefox
```

Taking this one step further, I use mas-cli (Mac App Store command line interface) to install programs that I can only get through the Mac App Store.

```
# Install MCG (a.k.a. "Mac callgrind")
mas install 799178412
```

By combining this approach with Homebrew Bundle, I'm able to create a Brewfile that captures most of the software I've installed on my system. I keep this file up-to-date, and in version control, so I can use it to quickly rebuild my system from scratch, should the need arise.

Colors and Themes

Colors and themes are important. We stare at our screens all day, so we should pick a color palette and theme that is comfortable and engaging, something that's enjoyable to work in for many hours each day.

For my system and most applications, I use dark mode.

I like to use the same theme across all my editors and terminal windows. I used Solarized Dark with the Menlo font family for a long time. In recent years, I've switched to Dracula and the JetBrains Mono font family, particularly for the increased contrast.

I use Starship for a fun and colorful command line prompt.

Productivity

I use a lot more software every day, and I'm grouping it under this productivity heading. Much of this is banal, but maybe you'll see something you've never heard or considered, so I hope it helps.

To keep my taskbar clean, I use Bartender. To keep my workspace organized, I use Magnet. To launch programs and search for files, I use Spotlight.

For web browsing, my browser of choice is Safari, though I use Firefox at times, and if I need a Chromium-based browser, I'll use Microsoft Edge.

My password manager is 1Password.

For mail, I like Apple Mail. GPGTools has excellent integration with Apple Mail, so I'm able to sign or encrypt any messages with my GPG keys. Sometimes, I will use Thunderbird to read and send messages to the PHP newsgroups (i.e., mailing lists).

For taking notes, I use Obsidian. It's the latest in a very long list of note-taking tools I've used over the years. I've still not found the one tool that feels right.

For diagramming, I use OmniGraffle, Miro, and StarUML. MySQL Workbench is still one of the best tools available for database diagramming.

To manage MySQL and MariaDB databases, I use Sequel Ace. Sequel Ace forked from Sequel Pro a few years ago when development on the latter stagnated. The new tool is under active development, and the developers have given it much care and attention. It's a joy to work with.

I do a lot of chatting for both open source and work, especially since our company is fully remote. I connect to Matrix



using Element and IRCcloud to connect to the Libera Chat IRC network. Of course, I'm also on Discord and a million different Slacks.

I use Keynote to create presentations and ScreenFlow to record screencasts for presentations. Pixelmator Pro and Affinity Designer are the tools I use for manipulating and creating graphics, which I do very poorly.

Cool Tools

One of the interesting things about human beings is our proclivity for creating and using tools. We extend and augment ourselves with tools. When we drive a car, we aren't simply *using* an automobile; it becomes a part of us through the steering wheel and pedals. In the same way, the tools we use to create software become a part of us through their

interfaces. Become proficient enough with them, and you'll never *think* about using them—they'll become second nature, a part of your brain.

That's the ultimate goal with any tool. They should make our work easier and more enjoyable. If we have to constantly think about the tool in order to use it, then it's getting in the way. Assess it and either endeavor to dig in and really learn it or drop it and find something else that works better for you.

I don't think there's anything special about my collection of tools. Most of them work well for me, and I've tried to highlight some of the ones that aren't working to show how this is a journey and not a destination. I began this article by saying the only good development environment is *your* development environment. Find the tools that work well for you, and make them your own.

My Set Up

Software

- 1Password - <https://1password.com>
- asdf - <https://asdf-vm.com>
- Bartender - <https://macbartender.com>
- CLion - <https://jetbrains.com/clion>
- Element - <https://element.io/download>
- GPGTools - <https://gpgtools.com>
- Homebrew - <https://brew.sh>
- Homebrew Bundle
- <https://github.com/Homebrew/homebrew-bundle>
- IdeaVim - <https://github.com/JetBrains/ideavim>
- IRCcloud - <https://irccloud.com>
- iTerm2 - <https://iterm2.com>
- Magnet - <https://magnet.crowdcafe.com>
- mas-cli - <https://github.com/mas-cli/mas>
- Miro - <https://miro.com>
- MySQL Workbench - <https://wb.mysql.com>
- Obsidian - <https://obsidian.md>
- Oh My Zsh - <https://ohmyz.sh>
- OmniGraffle - <https://omnigroup.com/omnigraffle>
- PhpStorm - <https://jetbrains.com/phpstorm>
- Sequel Ace - <https://sequel-ace.com>
- Starship - <https://starship.rs>
- StarUML - <https://staruml.io>
- Vim - <https://www.vim.org>
- WezTerm - <https://wezfurlong.org/wezterm/>

Hardware

- Apple Magic Trackpad
- Bose QuietComfort 35 wireless headphones
- Keychron K3 keyboard - <https://phpa.me/keychron>
- LG UltraFine 5K monitors - <http://lgeus.to/nlqZnq>
- MacBook Pro - <https://apple.com/macbook-pro>

Themes

- Dracula - <https://draculatheme.com>
- Solarized - <https://ethanschoonover.com/solarized/>



Ben Ramsey is a Senior Staff Engineer at Skillshare and an open source developer. An advocate for open source software and community, he is author of the *ramsey/uuid* library for PHP, a release manager for PHP 8.1 and 8.2, co-organizer of the Nashville PHP user group, author of several books on PHP development, and a popular speaker at PHP conferences. He lives with his family in Nashville, and you can follow him on the Fediverse at <https://phpc.social/@ramsey>.



Grade Deviations

Oscar Merida

We're continuing with grades and statistics by calculating statistics for Grade Point Averages. In the educational system in the United States, it's a way to boil down the grades across the multiple courses taken by a student. It involves translating a letter grade back to a numeric value and then calculating a weighted average. At the end of all this, we're left with a singular number we can use for a rough comparison between students.

Recap

Consider the following grades for a set of students. Each is a final grade in one of 6 subjects (Labeled S1-S6). Based on the table below, calculate each student's final Grade Point Average (GPA), the range of GPAs, and the standard deviation of the GPAs.

You can download a Tab-separated values gist of the grades¹.

ID #	S 1	S 2	S 3	S 4	S 5	S 6
222301	86	76	70	86	90	79
222302	85	90	81	91	90	95
222303	93	80	86	80	81	85
222304	80	93	77	93	82	84
222305	92	84	86	86	88	75
222306	69	86	85	80	86	98
222307	102	99	86	99	88	94
222308	73	74	95	88	70	87
222309	80	78	85	82	79	82
222310	86	88	102	87	71	79
222311	101	86	91	88	89	81
222312	80	92	82	92	101	86
222313	87	79	84	78	88	95
222314	87	94	83	79	94	90
222315	72	85	72	88	89	86
222316	82	79	82	88	79	78
222317	88	86	78	88	78	80
222318	95	75	72	74	83	81
222319	86	80	89	89	78	79
222320	81	86	87	93	84	91

¹ the grades: <https://phpa.me/gist-omerida>

² League/CSV: <https://csv.thephpleague.com>

³ SplFileObject: <https://php.net/class.splfileobject>

Use the following table to calculate the GPA.

Letter	%	GPA	Letter	%	GPA
A+	97+	4.3	C+	77-79%	2.3
A	93-96%	4.0	C	73-76%	2.0
A-	90-92%	3.7	C-	70-72%	1.7
B+	87-89%	3.3	D+	67-69%	1.3
B	83-86%	3.0	D	65-66%	1.0
B-	80-82%	2.7	E/F	< 65%	0.0

Data Ingest

When I can't use a library like League/CSV² to read and parse a CSV file, I prefer to use SplFileObject³ to read files. It provides an object-oriented interface for reading and writing files, including one for parsing CSV files. By specifying the tab character (\t), we can read each row and manipulate it (see listing 1).

Listing 1.

```

1. <?php
2.
3. $file = new SplFileObject(
4.     'https://gist.githubusercontent.com/'
5.     . 'omerida/b7a7bbb4c137e189ffc7d448cfe0baaf/raw/'
6.     . '58914d8e1f1f8ecae6dd653e416235a6f05bb05a/'
7.     . 'Puzzles-2023-02.tsv'
8. );
9.
10. $grades = [];
11. while (!$file->eof()) {
12.     $line = $file->fgetcsv("\t");
13.     // calculate the average for this person's grade
14.     $courses = array_slice($line, 1, 6);
15.     // use the first col as the key
16.     $grades[$line[0]] = array_sum($courses) / 6;
17. }
18.
19. // throw away the first one w/column names
20. array_shift($grades);

```

I chose to calculate the average across a student's courses since we don't need the individual grades to calculate the overall GPA...because math.

GPA Lookup

We have each student's average; we need to convert that into a value on the 4-point GPA scale. If we have a function that gives us the GPA given the final average, we can use `array_map()`⁴ to do the lookups to generate a `$gpas` array as in list 2.

Listing 2.

```
1. <?php
2.
3. function getGPA(float $grade): float
4. {
5.     switch (true) {
6.         case ($grade >= 97): return 4.3;
7.         case ($grade >= 93): return 4.0;
8.         case ($grade >= 90): return 3.7;
9.         case ($grade >= 87): return 3.3;
10.        case ($grade >= 83): return 3.0;
11.        case ($grade >= 80): return 2.7;
12.        case ($grade >= 77): return 2.3;
13.        case ($grade >= 73): return 2.0;
14.        case ($grade >= 70): return 1.7;
15.        case ($grade >= 67): return 1.3;
16.        case ($grade >= 65): return 1.0;
17.        default: return 0.0;
18.    }
19. }
20.
21. $gpas = array_map('getGPA', $grades);
```

Once I had this solution, I realized that it made sense to do the same lookup when reading the TSV file. Such a performance tweak will make a difference if we're working with hundreds of thousands of rows. Instead of looping through our dataset twice, we could only use one loop. I'll leave that optimization to you, dear reader.

Range

The range is the difference between a data set's highest and lowest values. Instead of writing loops to find the minimum and maximum values in `$gpas`, PHP provides two built-in functions for use: `min()`⁵ and `max()`⁶

```
$min = min($gpas);
$max = max($gpas);
$range = $max - $min;
echo "Max: " . $max . PHP_EOL;
echo "Min: " . $min . PHP_EOL;
echo "Range is " . $range . PHP_EOL;
```

Standard Deviation

Let's look at the formula for standard deviation, as this one might be trickier. (See Figure 1)

To start, we need the mean GPA, the number of GPAs, and the individual GPAs. For each GPA, we sum the square of the difference of each GPA and the mean GPA, divide that by the number of GPAs, and take the square root.

We can turn that formula into code now that we understand what steps we have to take. Listing 3 shows the steps to take. To make this reusable, we should make it a function that takes an array of floats or ints and returns the standard deviation.

Figure 1.

$$\sigma = \sqrt{\frac{\sum (x_i - \mu)^2}{N}}$$

σ = population standard deviation

N = the size of the population

x_i = each value from the population

μ = the population mean

Listing 3.

```
1. <?php
2.
3. // Calculate the standard deviation
4. $meanGPA = array_sum($gpas) / count($gpas);
5.
6. // Difference of each from mean
7. $devs = array_map(function($gpa) use ($meanGPA) {
8.     return $gpa - $meanGPA;
9. }, $gpas);
10. // Square all the differences
11. $devs = array_map(function($gpa) {
12.     return $gpa * $gpa;
13. }, $devs);
14. // Sum all the difference
15. $sumDevs = array_sum($devs);
16. // Divide by number of items
17. $variance = $sumDevs / count($gpas);
18.
19. $stdev = sqrt($variance);
20.
21. echo "Standard deviation is: " . $stdev . PHP_EOL;
```

4 `array_map()`: https://php.net/array_map
5 `min()`: <https://php.net/min>
6 `max()`: <https://php.net/max>



Maze Rats

Next month, we'll revisit a topic from the earliest of columns in two parts. In this first part:

The great wizard Archlin has hired you to build a maze to trap treklins that will be traveling to his tower to steal his treasure. To ensure you're up to the task, he's asked you to indicate in his spell book, the PHP-o-nomicon, how you plan to represent the design of the maze.

The maze will be a square grid. Each square in the grid has four walls (north, south, east, and west) and each wall can either be solid or an opening.

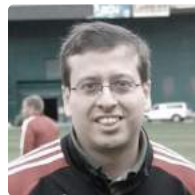
Related Reading

- PHP Puzzles: Grid Mapping by Sherri Wheeler, December 2020.
<https://phpa.me/puzzles-dec20>
- PHP Puzzles: Staircase Path by Sherri Wheeler, January 2021.
<https://phpa.me/puzzles-jan-21>
- PHP Puzzles: Environmental Noise by Sherri Wheeler, February 2021.
<https://phpa.me/puzzles-feb-21>

7 PsySH: <https://psysh.org>

Some Guidelines And Tips

- The puzzles can be solved with pure PHP. No frameworks or libraries are required.
- Each solution is encapsulated in a function or a class, and I'll give sample output to test your solution against.
- You're encouraged to make your first attempt at solving the problem without using the web for clues.
- Refactoring is encouraged.
- I'm not looking for speed, cleverness, or elegance in the solutions. I'm looking for solutions that work.
- Go ahead and try many solutions if you like.
- PHP's interactive shell (php -a at the command line) or 3rd party tools like PsySH⁷ can be helpful when working on your solution.
- To keep solutions brief, we'll omit the handling of out-of-range inputs or exception checking.



Oscar Merida has been working with PHP since version 4 was released and is constantly learning new things about it and still remembers installing an early version on Apache. When not coding or writing, he enjoys RPGs, soccer, and drawing. [@omerida](#)

**STAY UP-TO-DATE
WITH THE LATEST PHP TRENDS
AND TECHNOLOGIES**

SUBSCRIBE

www.youtube.com/@Phparch



PSR-17: HTTP Factories

Frank Wallen

In PHPArchitect Issue 7 of Volume 21 (way back in July 2022), I wrote about *PSR-7: Message Interfaces*¹, where Request and Response interfaces were defined to ensure that requests and responses were properly structured and handled. PSR-17² was created to support PSR-7 by defining interfaces for factories to create HTTP objects. The factory design pattern is an excellent way to introduce standards in components not specifically implementing PSR-7 interfaces. They will produce the request and response objects following the recommended standards, and the implementing code will not require a major refactor. PSR-7 was written before the release of PHP 7; therefore, following these recommended standards is important for security and stability.

Let's review the interfaces. Here's the Request factory:

```
namespace Psr\Http\Message;

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\UriInterface;

interface RequestFactoryInterface
{
    public function createRequest(
        string $method,
        UriInterface $uri
    ): RequestInterface;
}
```

The `$uri` parameter does not specifically typehint the `UriInterface`, meaning that `$uri` can also be a string. As described in the Meta document³:

...creation of the URI instance is secondary. Requiring a UriInterface means users would either need to also have access to a UriFactoryInterface, or the RequestFactoryInterface would have a hard requirement on a UriFactoryInterface. The first complicates usage for consumers of the factory, the second complicates usage for either developers of the factory, or those creating the factory instance.

Essentially, this means that if the implementor wants to use `UriInterface`, it can be done in the body of the `createRequest` method and even use a `UriFactory`.

The Response factory:

```
namespace Psr\Http\Message;

use Psr\Http\Message\ResponseInterface;

interface ResponseFactoryInterface
{
    public function createResponse(
        int $code = 200,
        string $reasonPhrase = ''
    ): ResponseInterface;
}
```

The Meta document⁴ for PSR-17 responds to the potential question about `$reasonPhrase` being optional. This is to support PSR-7's `ResponseInterface::withStatus()`, where the status code and reason phrase are related and necessary.

The Stream factory:

```
namespace Psr\Http\Message;

use Psr\Http\Message\StreamInterface;

interface StreamFactoryInterface
{
    public function createStream(
        string $content = ''
    ): StreamInterface;

    public function createStreamFromFile(
        string $filename,
        string $mode = 'r'
    ): StreamInterface;

    public function createStreamFromResource(
        $resource
    ): StreamInterface;
}
```

¹ PSR-7: Message Interfaces: <https://www.php-fig.org/psr/psr-7/>

² PSR-17: <https://www.php-fig.org/psr/psr-17/>

³ Meta document: <https://phpa.me/php-psr17>

⁴ Meta document: <https://phpa.me/psr17>



The Stream factory needs to support creating Stream responses from several sources: content, file, and resource. If created from a string, it is highly recommended that the stream be temporary, such as using `fopen('php://temp', 'r+')`.

The UploadedFile factory:

```
namespace Psr\Http\Message;

use Psr\Http\Message\StreamInterface;
use Psr\Http\Message\UploadedFileInterface;

interface UploadedFileFactoryInterface
{
    public function createUploadedFile(
        StreamInterface $stream,
        int $size = null,
        int $error = UPLOAD_ERR_OK, //value is 0
        string $clientFilename = null,
        string $clientMediaType = null
    ): UploadedFileInterface;
}
```

Note that the uploaded file *should* come from a stream created by the Stream factory. This will ensure consistency and comply with PSR-7 recommendations.

The Uri factory:

```
namespace Psr\Http\Message;

use Psr\Http\Message\UriInterface;

interface UriFactoryInterface
{
    public function createUri(
        string $uri = ''
    ): UriInterface;
}
```

The URI factory must be able to create a Uri for both client and server requests.

The ServerRequest factory:

```
namespace Psr\Http\Message;

use Psr\Http\Message\ServerRequestInterface;
use Psr\Http\Message\UriInterface;

interface ServerRequestFactoryInterface
{
    public function createServerRequest(
        string $method,
        UriInterface $uri,
        array $serverParams = []
    ): ServerRequestInterface;
}
```

The optional `$serverParams` is important here as the `ServerRequestInterface` does not have a mutator method for server

parameters and they must be provided during creation. One may also notice that there is no method for creating a server request from superglobals. This should be implementation-specific, as there are scenarios where not all the data will be available. In an asynchronous environment (like Swoole⁵ and ReactPHP⁶ or a CLI SAPI (like a console command), one may not have data in `$_GET`, `$_POST`, `$_COOKIES`, and `$_FILES` or will receive different information in `$_SERVER`. Following are some examples of how some packages have implemented PSR-17 and use superglobals to create a `ServerRequest`:

Guzzle

See the repository [here](#)⁷.

While not having a specific `ServerRequestFactory`, the `ServerRequest` object contains several methods to create objects. Listing 1 is the one for creating a `ServerRequest` from superglobals:

Listing 1.

```
1. public static function fromGlobals(): ServerRequestInterface
2. {
3.     $method = $_SERVER['REQUEST_METHOD'] ?? 'GET';
4.     $headers = getallheaders();
5.     $uri = self::getUriFromGlobals();
6.     $body = new CachingStream(
7.         new LazyOpenStream('php://input', 'r+')
8.     );
9.     $protocol = isset($_SERVER['SERVER_PROTOCOL'])
10.         ? $this->getProtocolVersion()
11.         : '1.1';
12.
13.     $serverRequest = new ServerRequest(
14.         $method,
15.         $uri,
16.         $headers,
17.         $body,
18.         $protocol,
19.         $_SERVER);
20.
21.     return $serverRequest
22.         ->withCookieParams($_COOKIE)
23.         ->withQueryParams($_GET)
24.         ->withParsedBody($_POST)
25.         ->withUploadedFiles(
26.             self::normalizeFiles($_FILES)
27.         );
28. }
29.
30. public function getProtocolVersion(): string
31. {
32.     return str_replace(
33.         'HTTP/',
34.         '',
35.         $_SERVER['SERVER_PROTOCOL']
36.     );
37. }
```

⁵ Swoole: <https://openswoole.com>

⁶ ReactPHP: <https://reactphp.org>

⁷ here: <https://github.com/guzzle/guzzle>



Here the `fromGlobals` method does not expect any arguments and will determine and collect the data itself.

Diactoros

See the repository here⁸.

This package *does* have a `ServerRequestFactory`, and the `fromGlobals` method (See Listing 2) has expectations of optional arguments to provide the data. If those values are not provided, then several steps are taken to extract the data from *existing* superglobals to fill in where possible (see `UriFactory::createFromSapi()`, `marshalMethodFromSapi()`, and `marshalProtocolVersionFromSapi()`).

Listing 2.

```
1. public static function fromGlobals(  
2.     ?array $server = null,  
3.     ?array $query = null,  
4.     ?array $body = null,  
5.     ?array $cookies = null,  
6.     ?array $files = null,  
7.     ?FilterServerRequestInterface $requestFilter = null  
8. ): ServerRequest {  
9.     $requestFilter = $requestFilter  
10.    ?: FilterUsingXForwardedHeaders::trustReservedSubnets();  
11.  
12.    $server = normalizeServer(  
13.        $server ?: $_SERVER,  
14.        is_callable(self::$apacheRequestHeaders)  
15.            ? self::$apacheRequestHeaders  
16.            : null  
17.    );  
18.  
19.    $files = normalizeUploadedFiles($files ?: $_FILES);  
20.    $headers = marshalHeadersFromSapi($server);  
21.  
22.    if (null === $cookies &&  
23.        array_key_exists('cookie', $headers)  
24.    ) {  
25.        $cookies = parseCookieHeader($headers['cookie']);  
26.    }  
27.  
28.    return $requestFilter(new ServerRequest(  
29.        $server,  
30.        $files,  
31.        UriFactory::createFromSapi($server, $headers),  
32.        marshalMethodFromSapi($server),  
33.        'php://input',  
34.        $headers,  
35.        $cookies ?: $_COOKIE,  
36.        $query ?: $_GET,  
37.        $body ?: $_POST,  
38.        marshalProtocolVersionFromSapi($server)  
39.    ));  
40. }
```



Related Reading

- *PSR Pickup: PSR 14: Event Dispatcher* by Frank Wallen, January 2023.
<https://phpa.me/psr-jan-23>
- *PSR Pickup: PSR-15: HTTP Server Request Handlers* by Frank Wallen, February 2023.
<https://phpa.me/psr-feb-23>

Conclusion

Although the factory interfaces have been explained to be presented as singular classes, it is important to note that some implementations may choose to put them in a single class. While this approach may help in simplifying code management, it may also have some drawbacks. For instance, the code may be more difficult to read and understand, especially if multiple interfaces are combined in a single class. Therefore, despite the potential benefits, it is usually better to keep the factory interfaces separated. This will make the code more modular and easier to maintain in the long run. Furthermore, by separating the factory interfaces, one can create more specific and specialized classes that can be reused in other parts of the codebase, leading to a more efficient and scalable code structure.



Frank Wallen is a PHP developer, musician, and tabletop gaming geek (really a gamer geek in general). He is also the proud father of two amazing young men and grandfather to two beautiful children who light up his life. He is from Southern and Baja California and dreams of having his own Rancharito where he can live with his family, have a dog, and, of course, a cat. [@frank_wallen](https://twitter.com/frank_wallen)

⁸ here: <https://github.com/laminas/laminas-diactoros>



“Depend” Toward Stability

Edward Barnard

Something was wrong in the sense that we missed our project deadlines repeatedly. What was wrong? This one took a while to figure out. We’ll look to Seymour Cray for part of the answer, and to James Grenning, one of the original signers of The Agile Manifesto, for the rest of that answer.

Software development can get crazy when you have too many things happening simultaneously. Domain-Driven Design, in an Agile context, should help with this—but I missed a point. I’ll show you what happened, but first, let’s visit what Seymour Cray had to say on the subject.

Short Circuit

Seymour R. Cray, Jr. designed large-scale computer equipment throughout the last half of the 20th Century. He made a famous statement that relates to our problem of the moment. I’ve never seen Cray’s statement in print. Nor can I find it online:

“Only change one variable at a time.”—Seymour Cray

Do you see the problem here? How can that statement be famous if it’s probably not in print or easily found online? The answer is that his perspective was well-known to us, his employees. It was a fairly common saying within Cray Research, and thus I’ve always thought of it as “famous”—even though I can’t find it anywhere online.

The first lesson within “DDD Alley”, is to figure out what your customers and stakeholders *really* mean!

Martin Fowler explains¹:

Ubiquitous language is the term Eric Evans uses in Domain-Driven Design for the practice of building up a common, rigorous language between developers and users. This language should be based on the Domain Model used in the software—hence the need to be rigorous, since software doesn’t cope well with ambiguity.

Evans makes clear that using the ubiquitous language in conversations with domain experts is an important part of testing it, and hence the domain model. He also stresses that the language (and model) should evolve as the team’s understanding of the domain grows.

Thus, Domain-Driven Design guides us to the answer to our first problem. The word “famous” was probably the wrong word for me to use, even though it’s the word that comes to my mind as the (hypothetical) subject-matter expert. We clarified that I actually meant “well-known among employees

of Cray Research during the 1980s”. What I called “famous” was actually an obscure reference from 40 years ago!

Fowler knows this phenomenon all too well. He continues the above explanation by quoting Eric Evans:

Domain experts should object to terms or structures that are awkward or inadequate to convey domain understanding; developers should watch for ambiguity or inconsistency that will trip up design.

In this particular example, we could have ended with a confrontation. Confrontational software development is not a good thing!

I could say, “it’s a famous statement” and you could rightly respond with, “No, it’s not famous”. We got at what I *actually* meant with additional questions. This is something like the five whys² technique for getting at the root cause of a problem. When designing software, words need to mean the same thing to each of us!

This “famous” statement is quite important to the issue I’ll describe below, but what did Cray actually mean? What’s the context? What was Cray talking about? Once again, Domain-Driven Design reminds us that when developing a ubiquitous language, its scope is limited to a specific bounded context³. We all know that specific phrases can have different meanings to different people and in different contexts. It’s important to tune your hearing, so to speak, to listen for context and potential ambiguity.

We know Cray earned a master of science degree in applied mathematics⁴ from the University of Minnesota. He might well be referring to “variable” in the mathematical sense. But Cray also earned his Electrical Engineering degree the year before, and I did place his “famous” statement in the context of large-scale computer design.

Indeed, hardware design does provide a good picture of Cray’s meaning. Figure 1 is from Cray’s first U.S. patent.

Suppose you were an electrical engineer. When experimenting with an electrical circuit, no doubt you could vary the input voltage and observe results directly, or perhaps with

¹ Martin Fowler explains: <https://phpa.me/mfowler-ubiq-lang>

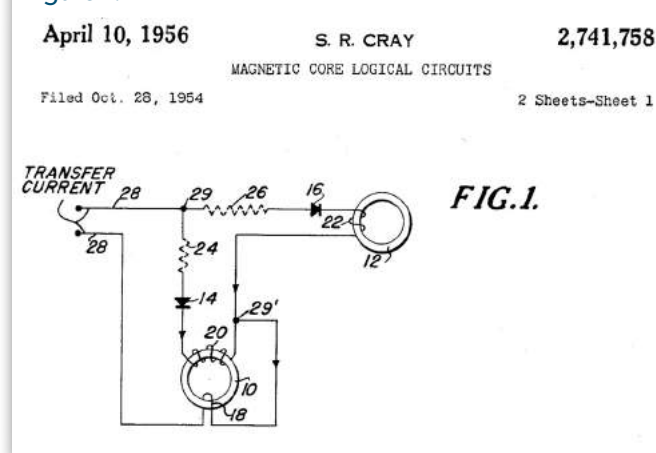
² five whys: https://en.wikipedia.org/wiki/Five_whys

³ bounded context: <https://phpa.me/mfowler-bounded-context>

⁴ earned a master of science degree in applied mathematics: <https://phpa.me/americanhistory>



Figure 1.

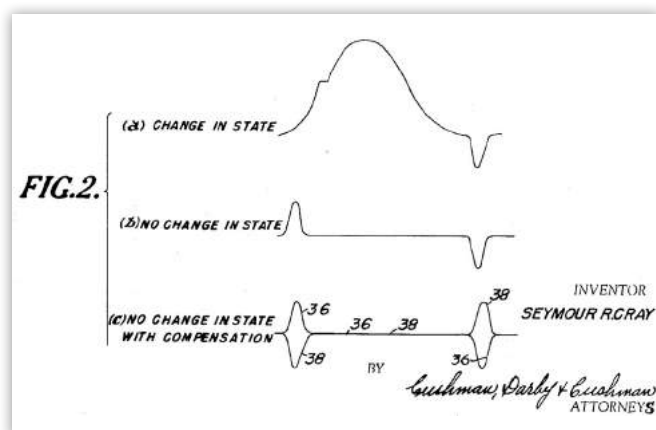


some sort of test equipment like an oscilloscope. You might also vary signal phase and frequency to, again, observe results. You might also vary amperage, resistive load, whatever. We’ve just described several different variables, that is, things that can be varied.

Let’s assume those variations relate to the point labeled “transfer current” at the upper left of Figure 1.

Figure 2, from the same patent, shows three lines, that is, wave forms:

- Change in state
- No change in state
- No change in state with compensation



Can you imagine the experimentation involved in getting from line (a) and line (b) to line (c)? Cray undoubtedly changed one variable at a time, observing and recording results. “Turning all the knobs” at once would have been pointless. Something would have changed, but why? What’s the optimum course?

Take a close look at line (c) in Figure 2.

- At the left edge of the line, signal “38” compensates signal “36”. When 36 goes high, as shown in line (b), Cray designed signal “38” to go low, with (I presume) a net result of zero or steady state.

- Toward the right, when signal “36” goes low, “38” must go high.

The middle of line (c) shows both “36” and “38” at the same level.

Suppose two different people (or two teams) were simultaneously working with the transfer current. One team is experimenting with *delivering* the transfer current. The first team might be tasked with discovering how fast the transfer current can change state—from high to low or low to high.

The second team might be focused on the magnetic core (labeled “10” in Figure 1). Specifically, how many turns of wire (labeled “20” in Figure 1) might be optimum.

Can you imagine both teams working on the same physical device...simultaneously? Neither team could get anything done because the other team continuously changed the environment. The final result, no doubt, would be a hot mess with actual smoke rising in the air.

Clean Sheet of Paper

Deadlines are important. In Cray’s case, “deadlines” generally meant the life or death of the entire company. How did he respond?

In Seymour Cray’s interview for the National Museum of American History at the Smithsonian⁵, the interviewer (David Allison) asked:

We were talking a while ago about designing at Control Data Corporation when you first got started. We know your goal was to design fast scientific computers. Was it scary in terms of being able to meet an objective that would deliver and keep the company in business?

Cray answered:

It was scary but not at all what you expect. And so I’ll enjoy telling you this part.

The scary part was that we were trying to start a company with virtually no money and accomplished the construction of the world’s biggest [meaning “fastest”] scientific computer and do it quickly. And so the environment was the threat.

There was a large newspaper in Minneapolis and probably still is. The Minneapolis Tribune I guess it is. [The Minneapolis Tribune at the time, became the current Star Tribune in 1987.] And so we were in the paper warehouse where they store these large rolls of newsprint. They were stacked up about four rolls high in this big warehouse.

I did my engineering work in one corner and at night I could hear the rolls slipping every once in a while. There was a block of wood under the end roll and so my fear

⁵ interview for the National Museum of American History at the Smithsonian: <https://phpa.me/americanhistorycray>



was that the block would slip some night and all those rolls would come down on me and I be squashed against the far wall.

This was a period of time for probably one year, but I shall always remember the sound of those rolls of newsprint slipping every once in a while, and it would sort of echo through the empty building as I was busy working on my computer in the corner.

Allison:

So were you also alone at night?

Cray:

Yes. Very much.

Allison:

It's always said of you that you liked to start with a clean piece of paper and draft.

Cray:

I had a good supply there.

Knowledge Crunching

With that unlimited supply of clean paper, Cray explained his approach to the “knowledge crunching” process of Domain-Driven Design:

Well, you have to understand that the blank sheet of paper is not a blank mind. I wanted to take advantage of all the things that I remembered and all the inputs I had gotten from people over a period of a few years to help me decide what to make.

But by the blank piece of paper I mean that I liked to start over with the technical details, review all the things that the world offers at this point in time rather than to reuse things that were just used. I would rather try new ingredients in building the computer than old ingredients. That is what I meant by the expression “a blank piece of paper” for each design.

This of course gets one in trouble because it increases risk. Every time you take a new approach, new ingredients, you increase risk. But it was my feeling that the rewards would come often enough so that taking those kind of risks would have long term benefit. I think they did during my career.

Cray has one other observation that I’ve always considered super-important for myself. James Higginbotham, in *Principles of Web API Design: Delivering Value with APIs and Microservices*, page 45, quotes Alberto Brandolini, the inventor of Event Storming:

The real story is that software developers are spending a relevant amount of their time learning, in order to do things they don't have a clue about. Differently from other professions, we're doing things for the first time, most of the time (even if it looks like the same old typing from the outside).

Brandolini is describing the continuous need for knowledge crunching. It's not a matter of gaining more “coding” expertise; it's a matter of delivering business capabilities, which means first learning and figuring out how to do so.

Cray, in the Smithsonian interview, put it this way in describing his first position after graduating from the University of Minnesota in 1951:

It was the blind leading the blind. There were people all hired within a year because the facility was only that old. There were perhaps a dozen Navy people who had some idea about the purpose of the whole project but were mostly very young people at that point in my time and knowing nothing about computing equipment I spent a good deal of time in the library reading what material there was, almost all from universities of an academic nature as to what computing should be and how it works. I quickly discovered in a few months that there wasn't much there.

It was on to inventing from that point. It was a wonderful opportunity to get started without a lot of peer pressure in the sense of there wasn't much of anybody else there.

Risk and Security

Cray was eager to take technical risks, but only against a background of stability. In founding Cray Research in 1972, he explained,

In terms of people, I needed the security that I didn't need in a technical area. I was willing to take a lot of risks in a technical sense. I wanted the security of having people I knew, when starting a new company. I think you can understand that.

You focus your risk on particular areas. I wanted the people that I'd worked with most of my lifetime to be a part of this project because I liked the relationships. I wouldn't have to spend my time working on the human



aspects of the relationship involved because they were already established; therefore, I could concentrate all my efforts in the technical area. One brings one's friends on these adventures.

I believe this is what Seymour Cray meant by “Only change one variable at a time.” When you’re changing a variable, you’re creating risk. The more innovative the change, the greater the risk—and the greater the potential reward, but the risk is still there. Your risk needs to be in the context of a solid, stable surface rather than on quicksand, so to speak.

Clean Architecture

Robert C. Martin, in *Clean Architecture: A Craftsman's Guide to Software Structure and Design*, hides a brilliant chapter by James Grenning, “Clean Embedded Architecture”. Why would a PHP person need to know about embedded architecture? But here’s the point I missed.

On Page 261, “The Target-Hardware Bottleneck”, Grenning describes a problem we often face. You may be surprised:

Most of the time the hardware is concurrently developed with the software and firmware. As an engineer developing code for this kind of system, you may have no place to run the code. If that's not bad enough, once you get the hardware, it is likely that the hardware will have its own defects, making software development progress even slower than usual...

When embedded code is structured without applying clean architecture principles and practices, you will often face the scenario in which you can test your code only on the target hardware. If the target is the only place where testing is possible, the target-hardware bottleneck will slow you down.

Our Bottleneck

In my own project, we are in the midst of rewriting our legacy codebase to use a PHP framework and also be PHP 8.1-compliant. A primary reason for undertaking the rewrite is the need for major structural changes to our database design.

In essence, the rapidly-evolving database schema is our “target hardware”. By design, the PHP back-end code is not closely coupled to the database schema—or so I thought. If the database changes, only “repository” code should need changing to compensate. This wasn’t the case!

As Grenning explains (page 262), think of “hardware” as “MySQL database”:

The separation between hardware and the rest of the system is a given—at least, once the hardware is defined. Here is where the problems often begin... There is nothing

that keeps hardware knowledge from polluting all the code. If you are not careful about where you put things and what one module is allowed to know about another module, the code will be very hard to change. I'm not just talking about when the hardware changes, but when the user asks for a change, or when a bug needs to be fixed.

Software and firmware intermingling is an anti-pattern. Code exhibiting this anti-pattern will resist changes. In addition, changes will be dangerous, often leading to unintended consequences. Full regression tests of the whole system will be needed for minor changes. If you have not created externally instrumented tests, expect to get bored with manual tests—and then you can expect new bug reports.

What did I do? One of my favorite features of CakePHP is the code generator. I generally set up tooling so the Model classes can be regenerated at will. Nothing resides in the Model classes except the generated code. In this way, my PHP application sits independent of the actual database schema.

I made a mistake. I forgot what Grenning taught me...in person...years ago!

Since CakePHP generated “Entity” classes for me, and they closely match the Domain-Driven Design “Entity” pattern, I used them throughout the PHP codebase. This, unfortunately, meant that I allowed specific knowledge of the MySQL table schema to be depended on throughout that code.

We were rapidly, intentionally evolving table designs as we were simultaneously developing PHP code. The problem came when any aspect of a table or field changed that was already being used by some part of our PHP code. The nasty result was just as Grenning describes. Fortunately, I *had* developed a thorough set of integration (behat) tests, but the rework caused massive slowdowns, and we weren’t quite sure why. We certainly had not budgeted time for this issue!

Grenning calls this “getting your code addicted to the operating environment” (page 269):

To give your embedded code a good chance at a long life, you have to treat the operating system [in our case, the database schema] as a detail and protect against OS dependencies.

On page 271, Grenning rubs it in:

If you have ever moved your software from one Real-Time Operating System to another, you know it is painful. If your software depended on an OSAL [Operating System Abstraction Layer], instead of the OS directly, you would largely be writing a new OSAL that is compatible with the old OSAL.

Which would you rather do: modify a bunch of complex existing code, or write new code to a defined interface and behavior? This is not a trick question. I choose the latter.



Summary

Sometimes it's not enough to identify what went wrong. Learning from other peoples' experiences can also be helpful in learning how to “do it right”. We took a close look at Seymour Cray's perspective on computer design.

As it turned out, my own issue of the day was similar to problems seen with embedded software development. The root cause was allowing knowledge of “the target hardware”, our rapidly-evolving database schema, to leak into (and throughout) our PHP codebase. The result was missed deadlines.

Looking Ahead

You will recall that Seymour Cray described going to the library and soaking up all the knowledge that he could. I took a similar approach. Knowing we had a problem but not recognizing it, I discovered a second common problem. Sam Newman describes our second potential problem in *Monolith to Microservices: Evolutionary Patterns to Transform Your Monolith*, page 4. It's something of a word salad as-is:

Changes in functionality are primarily about changes in business functionality. But our business functionality is in effect spread across all three tiers, increasing the chance that a change in functionality will cross layers. This is an architecture in which we have high cohesion of related technology, but low cohesion of business functionality.

If we want to make it easier to make changes, instead we need to change how we group code—we choose cohesion of business functionality, rather than technology.

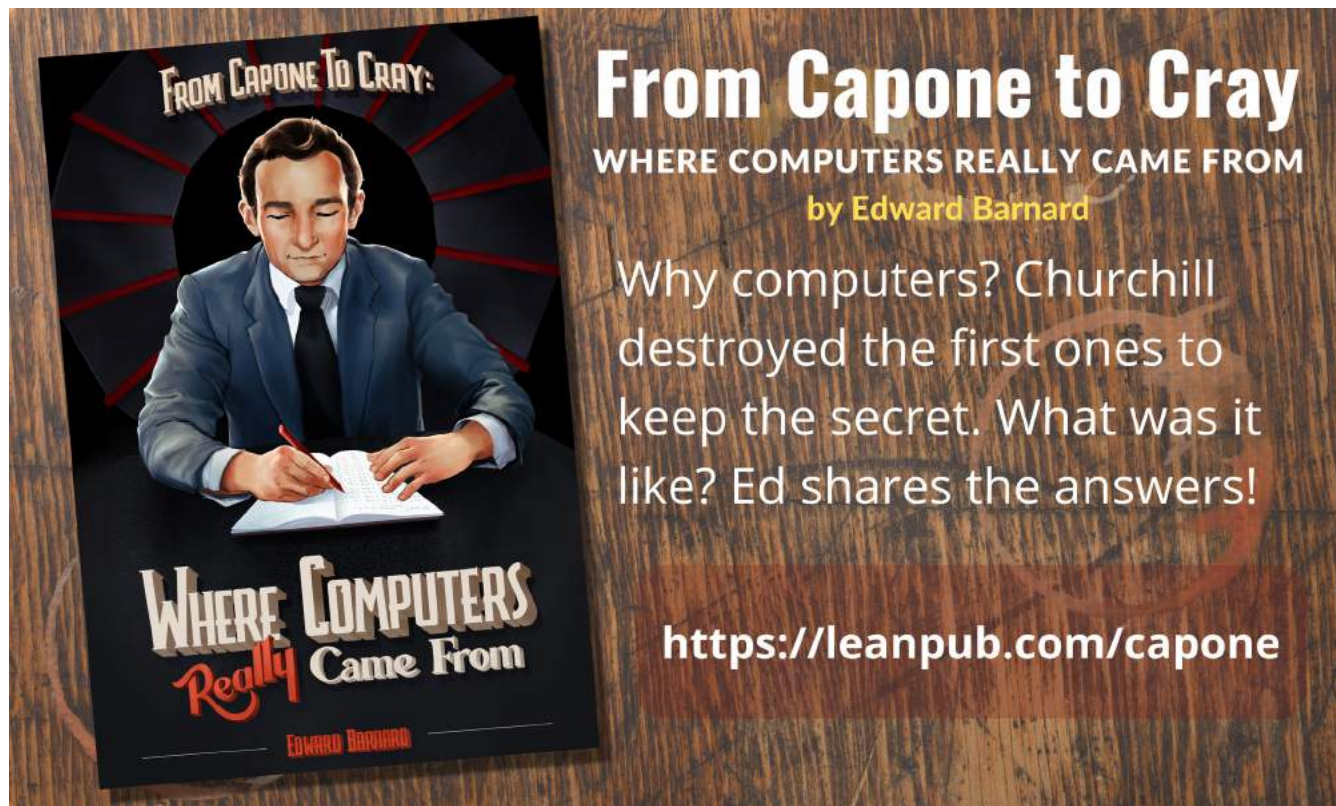
We drew the situation on our whiteboard, producing an “Aha!” moment. That's our next topic: to share the “Aha!” with you.

We found we have an “impedance mismatch” at the core of our PHP software architecture.

Next month's article will be titled “DDD Alley: Impedance Mismatch,” and I'll get it to you quick as I can!



Ed Barnard had a front-row seat when the Morris Worm took down the Internet, November 1988. He was teaching CRAY-1 supercomputer operating system internals to analysts as they were being directly hit by the Worm. It was a busy week! Ed continues to indulge his interests in computer security and teaching software concepts to others.
[@ewbarnard](https://twitter.com/ewbarnard)



From Capone to Cray

WHERE COMPUTERS REALLY CAME FROM

by Edward Barnard

Why computers? Churchill destroyed the first ones to keep the secret. What was it like? Ed shares the answers!

<https://leanpub.com/capone>

New and Noteworthy

PHP Releases

PHP 8.2.4 Released:

<https://www.php.net/archive/2023.php#2023-03-16-2>

PHP 8.1.17 Released:

<https://www.php.net/archive/2023.php#2023-03-16-1>

News

Platetscale: Mysql Video Course, Mysql for Developers (free)

This 64 video course is perfect for developers who work with the database and want to strengthen their MySQL skills in a way that's relevant to their every day work.

<https://phpa.me/schema-intro>

Phpstorm 2023.1 Rc

3v4l.org integration, New UI (Beta), Performance improvements, Data Flow Analysis in the PHP debugger, Custom Reg-exp based inspections, Full IDE Zoom, Clickable paths and class references for `var_dump()` / `dd()` output in the terminal, Improvements for Docker support:

<https://phpa.me/storm2023-1-rc>

Phpunit 10

The PHPUnit development team is pleased to announce the immediate availability of PHPUnit 10. This release adds new features, modifies and removes existing functionality, and fixes bugs.

<https://phpunit.de/announcements/phpunit-10.html>

Secure Your Php Code with Taint Analysis by Qodana

Qodana – a code quality platform by JetBrains. The platform is designed to bring server-side static analysis to your preferred CI tool.

<https://phpa.me/qodana-analysis>

Phproundtable: The Education of Development

A panel discussing how development is being taught and learned.

<https://phpa.me/phprt-86>

Open Collective: Php Core Roundup #10

The PHP Foundation currently supports six part-time PHP contributors who work on maintenance and new features for PHP. Maintenance is not limited to fixing bugs, but also includes work to reduce technical debt, making life easier for everyone working on PHP. The contributors funded by the PHP Foundation collaborate with other contributors on code, documentation, and discussions.

<https://phpa.me/core-roundup-10>

Laravel V10 Released

Laravel 10 continues the improvements made in Laravel 9 by introducing argument and return types to all application skeleton methods, as well as all stub files used to generate classes throughout the framework. In addition, a new, developer-friendly abstraction layer has been introduced for starting and interacting with external processes. Further, Laravel Pennant has been introduced to provide a wonderful approach to managing your application's "feature flags".

<https://blog.laravel.com/laravel-v10-released>

Data Modeling with Records – 5 Examples for Busy Developers

If you talk about the syntax, [Records](#)¹ seem to be a concise way of creating a new data type to store your immutable data. But what about its semantics? In other words, what does the term 'Record' imply? What are its features and limitations? What are the best places to use records, or the worst places to even think about them?

<https://phpa.me/intellij-busy-devs>

YouTube: Automated Testing Using Phpunit

In this video, Scott discusses how to create automated tests using PHPUnit, along with how to install PHPUnit and the basics for getting it working.

<https://www.youtube.com/live/zzMyOrPF4So>

¹ Records: <https://blog.jetbrains.com/idea/2021/03/java-16-and-intellij-idea/#Whyuserecords>



Laravel 10: New Features & Upgrade Impacts

Matt Lantz

Upgrading a Laravel application has rarely been considered a “large process”, but a version of Laravel is released every now and then that carries with it some “bigger” changes. Laravel 10, like the last couple of versions, has been an exercise in an elegant release structure that enables developers to upgrade and deploy an app, usually in less than 10 minutes. I’ve been able to get a couple applications upgraded recently, and it took less than 5 minutes in each case. There are a few small quirks in version 10’s upgrade process, which are outlined below. However, the overall upgrade is smooth and, in this version’s case, simply adds more features and a minor reduction in dependencies.

Before diving into the various new features, let’s look at a few of the small quirks. Laravel 10 now requires **PHP ^8.1**—this enables developers to use a handful of the latest features in PHP, such as Enums, Readonly properties, and First-Class Callable Syntax. PHP 8.1 also includes some notable performance improvements. Laravel has also set a requirement of Composer ^2.2, which should only impact developers or their workflows if you have yet to actively update composer on your work machine or your projects’ servers. It can be a pesky issue to hit when trying to deploy via Forge before your first set of team meetings. Other than those, there are only a few major dependency updates: **laravel/sanctum**, **spatie/laravel-ignition**, and if you want to use PHPUnit 10: **nunomaduro/collision**, and lastly **phpunit/phpunit**.

The most notable change I had to make in my applications was removing `$this->registerPolicies();` in the `AuthProvider` file. You should also be firmly aware that `$dates` was marked as deprecated in favor of `$casts` in Laravel 9, which means that for Laravel 10, it’s been removed from the core, so it will no longer do anything in your models unless you have something in place to handle it. Similarly, the `MockApplicationServices` trait has been removed as well. And according to the documentation, any use of methods like `expectsEvents` should be moved to `Event::fake()`. One more minor change is the Schema Builder change method. When we have to make changes to our database columns, we have always had to pull in **doctrine/dbal**. This is no longer true,

Hafez Divandari (@hafezdivandari on GitHub) created the pull-request to drop dbal altogether and offer native support for column modifying. In cases where you have **dbal** installed, you can add the following method to your `AppServiceProvider` in the `boot` method: `Schema::useNativeSchemaOperationsIfPossible()`. What’s evident in the PR is that Hafez did his homework and put together a well-constructed PR with only a few comments from Otwell. Though other minor changes have a low probability of impact, Laravel’s upgrade guide is well-documented as usual, and quick to read through. For applications with more than 30 routes, you may wish to explore using Laravel Shift, which automates

the process and provides a clear GIT comparison with the upgrade branch for a very reasonable cost. As per the Laravel Framework lifecycle with Laravel 10’s release, we can expect bug support until Aug 6, 2024, and security fixes until Feb 14, 2025.

As with most upgrades, we’re often far more interested in the new features vs. the improvements or adjusted requirements. Though Laravel 10 offered only a few new features, the Laravel team was able to release some new packages as well, making for a more significant upgrade for the Laravel ecosystem.

Overall, a couple of notable new features in Laravel 10 won’t impact your workflow but are worth mentioning. The first is the default invocable Validation rules. Unless you spend a fair amount of time in Terminal making requests for creating Validation rules, this change simply defaults that when running `php artisan make:rule Uppercase --invokable` flag from Laravel 9 is now simply not needed. The other significant difference is the removal of docblocks in favor of native types. For example, in Laravel 9, an edit method looked similar to the following:

```
/**
 * Show the form for editing the Note.
 *
 * @param \App\Models\Note $note
 * @return \Illuminate\View\View
 */
public function edit(Note $note)
{
    return view('notes.edit')->withNote($note);
}
```

Now, Laravel 10 looks like this:

```
/**
 * Show the form for editing the Note.
 */
public function edit(Note $note): View
{
    return view('notes.edit')->withNote($note);
}
```



Many applications have been following this pattern as of PHP 8. So, from what I can tell, there will be little impact on most developers' workflows. More notable new features in Laravel 10 include the new password generator and the Process Service. Within Laravel's Str helper class, we can now call `Str::password()`, which will, by default, produce a 32 random character password. According to Stephen Rees-Carter, the method internally uses `random_int()` to securely build the password, enabling elements like numbers, symbols, and spaces, making it cryptographically secure.

```
Str::password(
    length: 32,
    letters: true,
    numbers: false,
    symbols: true,
    spaces: false
);
```

The Laravel team also added the new Process Service to Laravel 10, which provides an elegant wrapper around Symfony's Process tooling for handling external processes that need to be run on application servers. Overall the Process Facade returns an easy-to-manage instance of a running process. Below are a few methods; however, there are far more options per the documentation, such as `throw()` and `throwIf($condition)`.

```
$result = Process::run('ls -la');

$result->successful();
$result->failed();
$result->exitCode();
$result->output();
$result->errorOutput();
```

Lastly, we should note the new addition for Laravel tests, adding a profiling option `--profile`, which can clearly outline how long each test is taking to help you identify those pesky slow tests in your CI pipelines.

One more thing.

Laravel 10's release came with a new package from the Laravel team called Pennant. It's an elegant feature flag management system. Feature flags themselves are nothing new, and there are other packages out there that similarly handle them. However, having the Laravel team create and maintain a package for handling them, in my mind, is a significant boost to the Laravel ecosystem. Feature flags can enable elegant feature release controls and work with basic A/B testing patterns. Within the `AppServiceProvider` or a more aptly named say, `FeatureServiceProvider`, which could then be added to the `config/app.php` file. In its `boot()` method you can add the following:

```
use Laravel\Pennant\Feature;
use Illuminate\Support\Lottery;

Feature::define('new-onboarding-flow', function () {
    return Lottery::odds(1, 10);
});
```

Then in a very basic (similar to Gates) manner, we can cross check in places in our application as to whether or not the feature is enabled:

```
if (Feature::active('new-onboarding-flow')) {
    //
}

@feature('new-onboarding-flow')
...
@endfeature
```

There are more means of checking features, such as within middleware. When using class based features via the cli command, `php artisan pennant:feature NewApi`, you can specify details such as `$user` specific identifiers for when the feature should be enabled. Such as in the example in Laravel's documentation:

```
/**
 * Resolve the feature's initial value.
 */
public function resolve(User $user): mixed
{
    return match (true) {
        $user->isInternalTeamMember() => true,
        $user->isHighTrafficCustomer() => false,
        default => Lottery::odds(1 / 100),
    };
}
```

Whether you've got an application that's a few versions behind or recently started one in Laravel 9 a few weeks ago, this upgrade should be a smooth transition. I plan on doing some extensive exploring of Pennant. Still, between the new Process service and better native Schema support, Laravel continues to make the developer experience as elegant as possible.



Matt has been developing software for over 13 years. He started his career as a developer working for a small marketing firm, but has since worked for a few Fortune 500 companies, lead a couple teams of developers and is currently working as a Cloud Architect. He's contributed to the open source community on projects such as Cordova and Laravel. He also made numerous packages and helped maintain a few. He's worked with Start Ups and sub-teams of big teams within large divisions of companies. He's a passionate developer who often fills his weekend with extra freelance projects, and code experiments. [@MattLantz](#)



Nothing Lasts Longer

Beth Tucker Long

The system needs a new feature, but we will need to do a lot of refactoring or rewriting to get it done. Someone proposes a quick workaround to get it working while we wait to get the rest of the proper work done. “What? We can’t do that”, someone responds. “We have to do this the right way.”

The old programming adage is that nothing lasts longer than a temporary solution. I believed in this wholeheartedly for decades. I stayed focused on fixing things the right way. I scheduled the time needed to implement proper fixes, which would deserve to live permanently in our codebase. I strove for code that I would be proud for others to see in a few years. I did this for way too long.

I’ve changed my mind.

Temporary solutions do tend to stick around, but this is because the problem has been solved. There is so much to do that we cannot waste time reworking a section of code that is functioning because we want to make it more “correct”. Ugly code that works is a salient solution. With this code functioning, we can focus on the myriad of other actually broken things to fix or infinite new features we need to implement.

In real life, what lasts much longer than the colloquial temporary solution, is the planning phase of the perfect implementation. Architecting the perfect solution to a problem requires too much philosophical debate, too much research, and too much time. By the time the perfect solution has been created, the codebase has changed enough that it is no longer perfect anymore. New technologies have been developed. New features are needed. New bugs need to be accounted for. Nothing stays static enough to survive waiting for the perfect solution.

Above all else, though, the opinions on what is “the perfect way to do things” will have changed. The perfect solution uses object-orientation. The perfect solution uses a framework. The perfect solution uses a micro-framework. The perfect solution uses functional programming. The perfect solution uses async workers. The perfect solution uses BDD. The perfect solution uses TDD. And, so it goes, on and on.

We are a constantly evolving industry. Our job is to be problem solvers, always thinking of new and better ways to solve a problem. This doesn’t mean that we should not attempt a “good” solution, but we need to remember:

All solutions are temporary.

Everything in our industry is just a small step away from needing a new feature, a new technology, a new usability theory, or a new accessibility requirement. Do not waste time on a “perfect” temporary solution when a quick temporary solution can do the job—both will be replaced. We have more important things to be doing than spending so much stress and planning on something that is more perfect but, ultimately, just as temporary.

So go ahead, implement that quick and dirty solution. If it lasts a long time, it means it’s working, and you have better things to do.



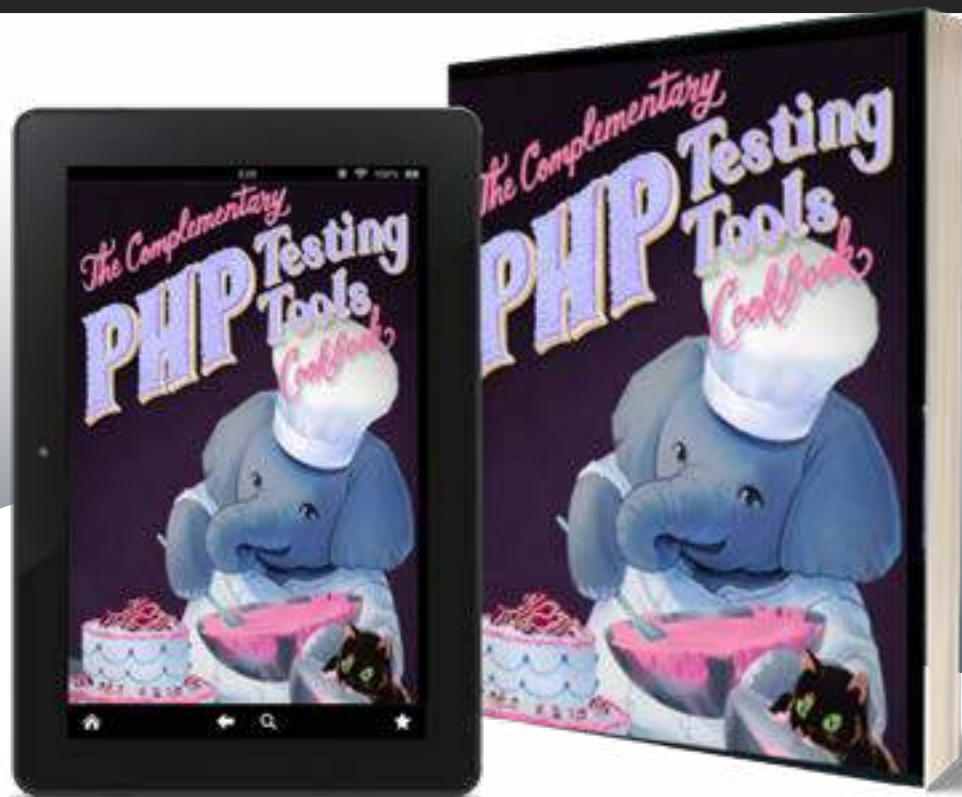
Beth Tucker Long is a developer and owner at Treeline Design¹, a web development company, and runs Exploricon², a gaming convention, along with her husband, Chris. She leads the Madison Web Design & Development³ and Full Stack Madison⁴ user groups. You can find her on her blog (<http://www.alittleofboth.com>) or on Twitter [@e3BethT](https://twitter.com/e3BethT)

1 Treeline Design: <http://www.treelinedesign.com>

2 Exploricon: <http://www.exploricon.com>

3 Madison Web Design & Development: <http://madwebdev.com>

4 Full Stack Madison: <http://www.fullstackmadison.com>

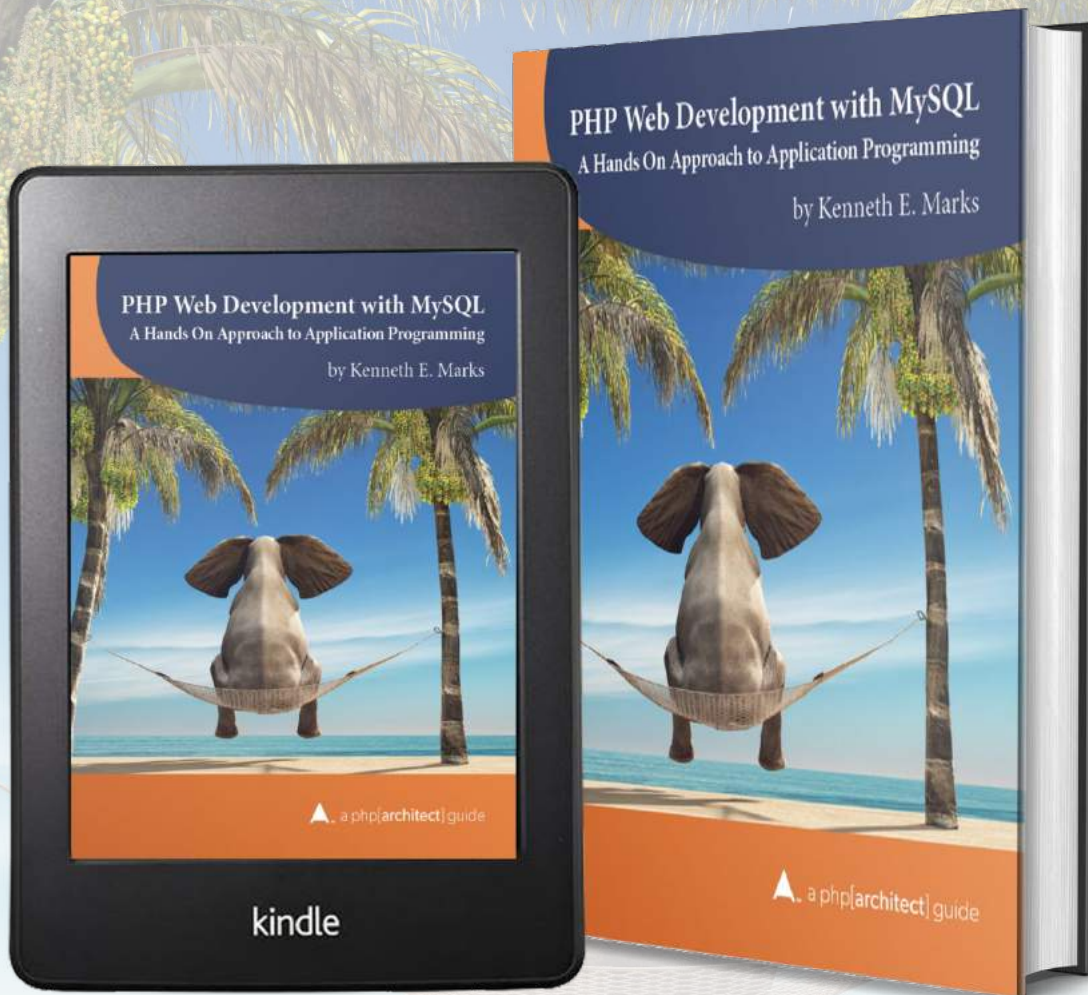


Learn how a Grumpy Programmer approaches improving his own codebase, including all of the tools used and why.

The Complementary PHP Testing Tools Cookbook is Chris Hartjes' way to try and provide additional tools to PHP programmers who already have experience writing tests but want to improve. He believes that by learning the skills (both technical and core) surrounding testing you will be able to write tests using almost any testing framework and almost any PHP application.

Available in Print+Digital and Digital Editions.

Order Your Copy
phpa.me/grumpy-cookbook



Learn how to build dynamic and secure websites.

The book also walks you through building a typical Create-Read-Update-Delete (CRUD) application. Along the way, you'll get solid, practical advice on how to add authentication, handle file uploads, safely store passwords, application security, and more.

Available in Print+Digital and Digital Editions.

Purchase Your Copy
<https://phpa.me/php-development-book>