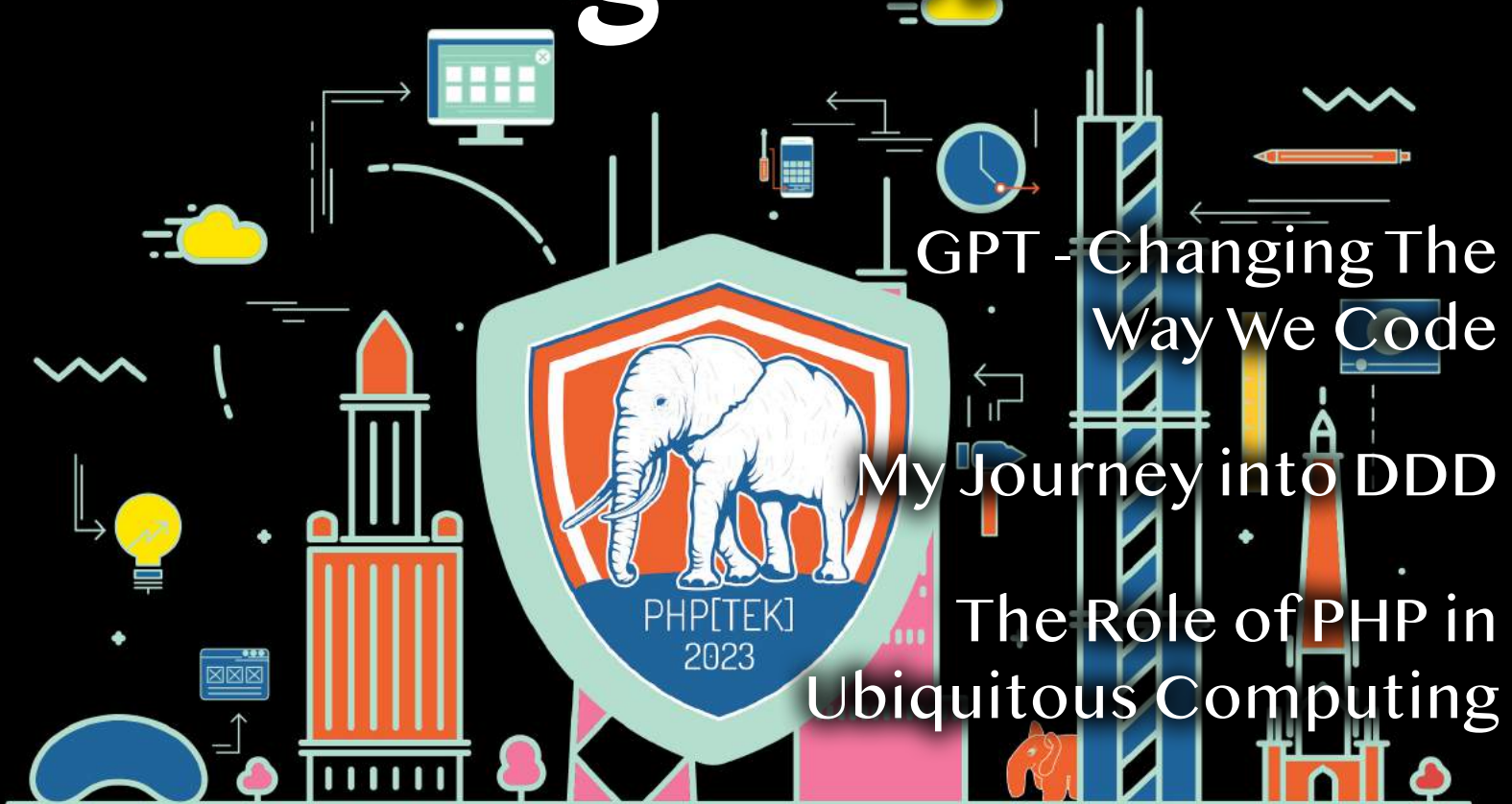


php[architect]

The Magazine For PHP Professionals

Getting TEKnical



GPT - Changing The
Way We Code

My Journey into DDD

The Role of PHP in
Ubiquitous Computing

ALSO INSIDE

The Workshop:
A Grumpy Intro to NeoVim

DDD Alley:
Impedance Mismatch

PHP Puzzles:
Maze Rats

Security Corner:
The Risks of Free Conference
Internet

Education Station:
12 Factor Applications

PSR Pickup:
PSR-18: HTTP Client

Artisan Way:
The Subtle Art of Optimal
DaaS

finally{ }:
Stop Waiting

JET
BRAINS



PhpStorm

Enjoy
productive
PHP

jetbrains.com/phpstorm

CONTENTS

APRIL 2023
Volume 22 - Issue 04

php[architect]

2 How did I get here?

3 My Journey into DDD

Edward Barnard

**7 The Role of PHP in
Ubiquitous Computing**

Jack Polifka

**11 GPT: Changing The Way We
Code**

Tomas Votruba

15 12 Factor Applications

Education Station

Chris Tankersley

**19 The Risks of Free
Conference Internet**

Security Corner

Eric Mann

**21 A Grumpy Programmer's
Intro To NeoVim and PHP**

The Workshop

Chris Hartjes

25 Maze Rats

PHP Puzzles

Oscar Merida

28 Impedance Mismatch

DDD Alley

Edward Barnard

32 PSR-18: HTTP Client

PSR Pickup

Frank Wallen

**34 The Subtle Art of Optimal
DaaS**

Artisan Way

Matt Lantz

36 Stop Waiting

finally{}

Beth Tucker Long

Edited while stressing about php[tek]

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.

How did I get here?

Eric Van Johnson

When my twins were born in 1998, I was 28 and working in the water purification industry. Despite the manual labor, I enjoyed the science and math involved, made great friends, and had a stable job with great pay. I was not a professional developer.

Don't get me wrong; I've been interested in computers since I was 12 and learned to code on a RadioShack TRS-80 ColorComputer my dad bought me. I studied Computer Science in Community College and even wrote code during my time in water purification to track performance and estimate customer needs.

In 2000, I discovered PHP4, which led me to make a risky career change. Despite having two children at home, I informed my wife of my decision, and she supported me. By 2001, I began my first IT job. Though it took years, I eventually became a full-time PHP developer. I've never had a regret about my decision to change careers.

If you were lucky enough to be around for the PHP4 days, you know, despite the ease of use and quick development workflow, it also had drawbacks. It wasn't the fastest language on the market. OOP, although technically in PHP4, wasn't a good implementation and lacked some concepts other OOP languages had. And security, well, had its challenges that tarnished the reputation of PHP to this day.

We did a lot with PHP4. Social networks were built, and projects that made it easy for anyone to launch a bulletin board quickly were created, along with other great projects that are still with us today, such as WordPress and Drupal.

In 2004, PHP5 was released, bringing with it improved support for object-oriented programming and performance. Over the next decade, many more PHP applications were created and Frameworks such as Symfony, Zend got stronger and new ones like Yii, CodeIgniter, CakePHP, and Laravel emerged. The Framework Interoperability Group (FIG) collaborated to establish standards

and best practices, which resulted in PHP Standards Recommendations (PSRs) that outlined coding style and autoloading. This enabled better package management and the creation of Composer.

A lot was accomplished in the PHP 5.x releases, but by 2014 many were saying PHP had stagnated, and even 'php is dead'. Then in 2015, PHP hit hyper-drive when the core team released PHP7. A lot had changed leading up to the release of PHP7, such as workflows the internals team followed, which allowed new features to be more easily introduced. Again, the performance of PHP itself saw one of its most significant jumps in a decade. Since the release of PHP7, PHP has seen a strong and steady release cadence, and today, you can safely run a production environment of the latest release of PHP 8.2.

I would be remiss if I didn't mention that for most of PHP's life, established in 2002, php[architect] has been there to release monthly magazines focused on everything PHP. Over 20 years later, we are still doing it, and this month is no exception. This month we have three feature articles, "My Journey into Domain-Driven Design" by Edward Barnard, "The Role of PHP in Ubiquitous" by Jack Polifka, and "How GPT automation will change the way we work with code sooner than you think" by Tomas Votruba.

In our regular columns, Chris Tankersly gives us "12 Factor Applications: Parts 7-12". Frank Wallen contributes "PSR-18: HTTP Client". Eric Mann contributes a timely article, "The risks of free conference internet." Oscar Merida brings us a solution to last month's challenge and presents us with another in "Maze Rats." In Edward Barnard's regular column DDD Alley, he writes about "Impedance Mismatch." Matt Lantez provides us with "The Subtle Art of Optimal DaaS." Joe Ferguson is still taking a break from The Workshop; however, Chris Hartjes fills in his place with "A Grumpy Programmer's Introduction To NeoVim and PHP." And as usually, Beth Tucker Long has "Stop Waiting" in her finally column.

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: <https://facebook.com/phparch>

Download the Code Archive:

https://phpa.me/April2023_code

My Journey into Domain-Driven Design

Edward Barnard

Domain-Driven Design is a team sport. Here's my story of what can and cannot be accomplished with a team size of one.

One Person Getting Started

March 13, 2023, I cleaned off my desk and took my stack of Domain-Driven Design books home. This was because, unfortunately, continuing down my current path of Domain-Driven Design was making our codebase worse.

Was my code “better” than what’s in our current codebase? That’s the wrong question to be asking. “Better” misses the point.

The Jargon File (version 4.4.7)¹ glossary describes my situation with their definition of Real Programmer²:

Real Programmer (noun): A particular sub-variety of hacker: one possessed of a flippant attitude toward complexity that is arrogant even when justified by experience... A Real Programmer's code can awe with its fiendish brilliance, even as its crockishness appalls. Real Programmers... terrify the crap out of other programmers—because someday, somebody else might have to try to understand their code to change it. Their successors generally consider it a Good Thing that there aren't many Real Programmers around anymore.

Our plan, since early 2020, had been for me to be transformative. That plan, we soon decided, included turning to Domain-Driven Design. That plan did not work out. What happened?

To see what was going wrong, we need to step back 30 years to 1993. That’s where my personal journey into Domain-Driven Design begins.

I contributed to a book, *Software Inspection*, by Tom Gilb and Dorothy Graham, published in 1993. I also supplied a guest chapter for that book. Tom kindly bought me dinner the next time he was in Minnesota and signed my copy of the book (Figure 1). He noted the first print run had already sold out.

Tom later told me that book had proven to be an “evergreen” technical book. He said technical books tend to have a short “tail”, meaning that after the initial release, sales drop off quickly. An “evergreen” book was one that continued to sell copies year after year. Kindle books were still 15 years into the future. Since print books, back then, were the only option, “evergreen” was good.

Figure 1.

The Authors dedicate this book to their families and their colleagues. Dorothy would like to thank her husband Roger for his continual encouragement, helpful suggestions and support, and her children Sarah and James for their patience and understanding. Tom would like to thank his family, especially his son Kai Thomas who was of so much practical help in the final stages of Tom's authorship. But authoring is also 'family'. The book is definitely a 'child' and the community of colleagues is very much our professional family.

To Ed
with 'undying'
appreciation for his
great chapter and
major advice on
making it a practical
"ready to wear" book
Tom Gilb
June 21 1994 minn
P.S. the first 5,000 are now
sold out!

This sounds like an auspicious start, right? *Software Inspection* appeared ten years before Eric Evans' groundbreaking *Domain-Driven Design: Tackling Complexity in the Heart of Software*. The irony is that my chapter's title proved to be prophetic. The title was “One Person Getting Started”.

The “one person” part was the problem then and remains the problem today. My co-worker Alex explained to me how he saw the situation at the time. He was quite willing to try *Software Inspection*. That was the title of Gilb's and Graham's book, and our own group was the topic of my case study in that book. However, if he looked around in a month or two

¹ <http://catb.org/jargon/html/index.html>

² <https://phpa.me/catb-jargon>

and nobody in the Software Division (of Cray Research) was adopting Software Inspection, he wasn't going to either.

That made sense to me. Software Inspection is a collaborative process. "One person" isn't collaborative. It can't be. We "looked around" months later. As Alex had predicted, nobody was interested. We stopped using Software Inspection as a formal methodology.

Embedded PHP

The Jargon File, linked above, provides a bit more background on the Real Programmer:

The archetypical Real Programmer likes to program on the bare metal and is very good at same, remembers the binary opcodes for every machine ever programmed, thinks that HLLs are sissy, and uses a debugger to edit his code because full-screen editors are for wimps. Real Programmers aren't satisfied with code that hasn't been tuned into a state of tenseness just short of rupture... Real Programmers can make machines do things that were never in their spec sheets; in fact, they are seldom really happy unless doing so.

Circa 2010, I had an interesting job interview. I'd been writing PHP full-time for a few years by then. I sat across the table from the hiring manager and his Principal Engineer. They were doing embedded software development for a small device that creates ID cards, access cards, chipped or embossed credit cards, etc. They had decided to create a web interface for device administration, much like the web interface for an at-home Wi-Fi router.

The web interface was an embedded server with PHP. They hired an intern to write the PHP code implementing the system administration pages. The serious-bug list was quite long. Production shipments of the device were scheduled for three months from this job interview.

The rest of the software was written in C or C++ on top of a linux-like Real Time Operating System. Nobody in the building had PHP expertise. Nobody knew how to fix these bugs. That, of course, was why we had this job interview.

As you would expect, the interview began by explaining the above situation. The Principal Engineer then said, "I'm not qualified to judge whether you can do the job. Can you?"

I answered, "Yes." That concluded the job interview!

I did get the bugs fixed with plenty of time to spare. I stayed on for the next phase of PHP development, but it was soon clear that this was not full-time use of me. I asked if I could be hired as a full-time C/C++ embedded developer. I had as much or more embedded C experience as nearly everyone in the building; however, I had zero experience with the modern microprocessors and board support packages being used.

I did hire on. That brings me to the next stage of this journey into Domain-Driven Design.

The best book I've ever found that explains Test-Driven Development is *Test-Driven Development in Embedded C*³ by James W. Grenning (2011). I was writing embedded C, but it was 2010, and that book didn't yet exist. However, a pre-release version was available from the publisher. What an "Aha!" moment that was!

We were having quality issues as a development team. I suggested we adopt Test-Driven Development and showed why. The team lead mandated we do so. He brought in James Grenning to teach us how. (Grenning was one of the original signers of the Agile Manifesto at Snowbird in 2001.)

Test-Driven Development requires people to write code differently. Code becomes more testable, but code doesn't look like it used to. The other developers chose *not* to adopt the methodology. They'd each been writing C on bare metal for many years and saw no reason to change their actions. That

manager was replaced, and I was out the door soon thereafter.

Why was I out the door? It was because my code didn't look right—my code was broken up into small, expressive methods. That was unnecessary fluff. I was spending too much time developing automated tests that nobody else wanted. Essentially, my value was negative. I kept heading off in another direction, following some methodology nobody else needed or wanted.

Since my code was *different* it was, virtually un-maintainable by others.

My failed experiment in returning to bare-metal software development did have one good outcome. I discovered Test-Driven Development. It's been a useful but contentious technique across the past decade of writing PHP.

Big Ball of Mud

Software architecture began its own transformative leap around 30 years ago. Three points proved relevant to this story:

- Brian Foote and Joseph Yoder presented Big Ball of Mud⁴ in 1997 and published 1999;
- Seventeen people met at Snowbird ski resort in Utah and created the "Manifesto for Software Development", February 11-13, 2001⁵;
- Eric Evans published *Domain-Driven Design: Tackling Complexity in the Heart of Software* in 2003.

You've probably seen or heard of software described as a Big Ball of Mud. Foote and Yoder originally described it as:

A BIG BALL OF MUD is haphazardly structured, sprawling, sloppy, duct-tape and baling wire, spaghetti code jungle. We've all seen them. These systems show unmistakable

⁴ <http://www.laputan.org/mud/>

³ <https://phpa.me/pragprog-titles>

⁵ <https://agilemanifesto.org/history.html>

signs of unregulated growth, and repeated, expedient repair.

Information is shared promiscuously among distant elements of the system, often to the point where nearly all the important information becomes global or duplicated. The overall structure of the system may never have been well-defined. If it was, it may have eroded beyond recognition. Only those who are unconcerned about architecture, and, perhaps, are comfortable with the inertia of the day-to-day chore of patching the holes in these failing dikes, are content to work on such systems.

As of 2020—three years ago—we had a Big Ball of Mud. I have only heard “Big Ball of Mud” used as a disparaging term. It’s certainly never been a compliment or an endorsement. “It’s a big ball of mud” has always meant “It’s bad. It’s awful.”

Yet, if you read Foote and Yoder’s paper, they are actually saying the exact opposite! With the very next sentence of description, they continue:

Still, this approach endures and thrives. Why is this architecture so popular? Is it as bad as it seems, or might it serve as a way-station on the road to more enduring, elegant artifacts? What forces drive good programmers to build ugly systems? Can we avoid this? Should we? How can we make such systems better?

As of 2020, then, we had a Big Ball of Mud. Several factors pushed us to consider a full rewrite:

- Our software’s user base had grown by more than 1000X over its life cycle.
- The PHP core team was dropping support for PHP 7; our code was written to PHP 4 or PHP 5.2 best practices; we saw PHP 8.1 as forcing a near-rewrite on us.

- Our database schema needed a comprehensive restructuring to meet upcoming needs.

PHP development, traditionally (in my experience), has not required much design work. We either consider the user experience first, and the PHP code flows from that, or we design the database tables first, and the PHP code is a basic CRUD (Create, Read, Update, Delete) application. “Architecture” questions are along the lines of, “do I place this function in the Model folder, the View folder, or the Controller folder?”

Foote and Yoder agree:

Scale: Managing a large project is a qualitatively different problem from managing a small one, just as leading a division of infantry into battle is different from commanding a small special forces team. Obviously, “divide and conquer” is, in general, an insufficient answer to the problems posed by scale. Alan Kay, during an invited talk at OOSPS-LA ’86 observed that “good ideas don’t always scale.” That observation prompted Henry Lieberman to inquire “so what do we do, just scale the bad ones?”

Protective Boundaries

Domain-Driven Design (DDD) has two areas, “strategic” DDD and the “tactical” DDD patterns. The tactical patterns are concrete and relatively easily understood, and many sample implementations exist in many programming languages. However, Evans and many others assure us, DDD’s power is in its “strategic” aspects. I found no examples of translating those latter concepts into a modern PHP application within a typical modern PHP ecosystem.

I decided it would be worth my while (and my company’s while) to figure this out and share this knowledge with others.

I developed the idea of “protective boundaries” as the most important of all, in the code. Collaboration and

knowledge-crunching were far more important, but that came before writing the code. More to the point, collaboration was outside my control, whereas writing code was something I could do on my own. My “protective boundaries” are the DDD concepts of “bounded context” and the consistency boundaries within the Aggregate pattern.

A year later, in late 2021, our rewrite failed.

2022 brought the next rewrite effort. I still had not taught or explained my new software-development approach to the rest of the team. We were too busy trying to hurry. I continued with the new architecture/approach. I did explain that unless we take time for me to train the team in this methodology, there’s no point. That time never happened. This, of course, is a completely common occurrence. That’s why I mention it.

Our second rewrite also failed. However, this time, some causes were becoming clear. Failed efforts *can* have value if you can act on what you’ve learned.

Full Circle

As of March 2023, I remained in the position of a wilderness explorer leading a party through the wilderness. When I look back and see that nobody’s following, that’s a problem! This fact brings me, personally, full circle.

Software Inspection includes a formal “gatekeeping” process. The requested Inspection cannot proceed unless the Entry Criteria are met (page 63):

The purpose of having entry criteria to the Inspection process is to ensure that the time spent in Inspecting the product and associated documents is not wasted, but is well spent. If the product document is of obviously very poor quality, then the checkers will find a large number of defects which can be found very easily by one individual alone, even the author. Inspections are cost-effective when the combined skills of a number of checkers are concentrated on a

document which seems superficially to be all right, but actually has a number of major defects in it which will only be found by the intensive inspection process, not by the author checking the product.

Software Inspection, by design, *only* pays off when collaboration is absolutely necessary. Don't take the Software Inspection path if a single expert can handle the task.

Domain-Driven Design carries the same principle but from the opposite direction. Eric Evans (*Domain-Driven Design* page 15) describes the ideal path:

The interaction between team members changes as all members crunch the model together. The constant refinement of the domain model forces the developers to learn the important principles of the business they are assisting... The domain experts often refine their own understanding by being forced to distill what they know to essentials, and they come to understand the conceptual rigor that software projects require.

All this makes the team members more competent knowledge crunchers. They winnow out the extraneous. They recast the model into an ever more useful form.

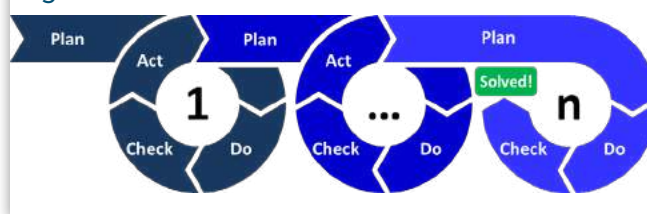
In other words, Domain-Driven Design carries the basic assumption of being a collaborative process that continuously includes *both* developers and domain experts. Collaboration is absolutely necessary because of the tension between development expertise and business-domain expertise.

Gilb and Graham wrote 30 years ago that the final phase of a Software Inspection is Process Improvement. They described the Plan-Do-Study-Act⁶ cycle promoted by W. Edwards Deming, the father of modern quality control. (See Figure 2, by Christoph Roser at AllAboutLean.com⁷ via Wikimedia Commons.)

⁶ <https://en.wikipedia.org/wiki/PDCA>

⁷ <http://allaboutlean.com>

Figure 2.



Deming called it the “Shewhart Cycle”, referring to Walter A. Shewhart⁸, the father of statistical quality control. We now, in the Agile context, call this a Retrospective⁹:

Identify how to improve teamwork by reflecting on what worked, what didn't, and why. We recommend running a Retrospective with your team every couple of weeks or at the end of a project milestone.

My own necessary process improvement, obviously, is collaboration. I've reached as far as I can go as “One Person Getting Started”. My personal journey into Domain-Driven Design is complete. It's a group effort from here.

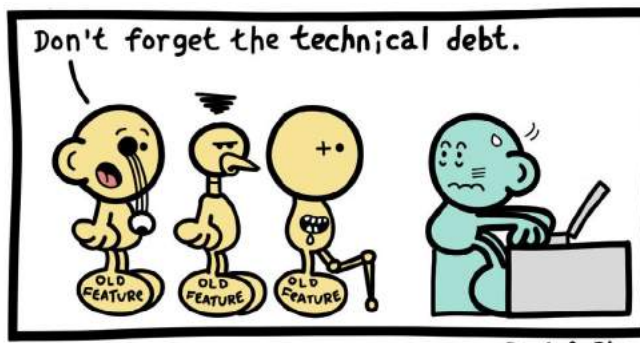
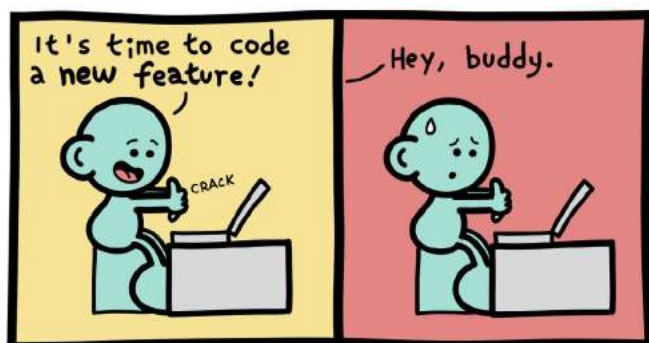


Ed Barnard had a front-row seat when the Morris Worm took down the Internet, November 1988. He was teaching CRAY-1 supercomputer operating system internals to analysts as they were being directly hit by the Worm. It was a busy week! Ed continues to indulge his interests in computer security and teaching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

[@ewbarnard](https://twitter.com/ewbarnard)

⁸ <https://phpa.me/wikipedia-WalterShewhart>

⁹ <https://phpa.me/atlassian>



Daniel Stori

Shared with permission from the artist Daniel Stori, turnoff.us

The Role of PHP in Ubiquitous Computing

Jack Polifka

The future of the Internet and computing will likely shift away from devices we sit down with and use on a daily basis. Technologies like augmented reality, virtual reality, and the Internet of Things will increasingly require users to physically interact with them instead. As a programming language primarily designed for web development, PHP will soon need to evolve into a more universal language that extends beyond the limits of the web. What is the role of PHP in these future technologies, and how can we, as developers, prepare ourselves for working with them?

PHP is a general-purpose server-side programming language primarily used for web development. It was initially developed by Rasmus Lerdorf in 1994 to maintain their personal homepage, hence the name PHP, before releasing it in 1995¹. Since that time, PHP has gone on to power some of the most popular web systems on the internet, including content management systems like WordPress² and Drupal³, technology giants like Facebook⁴, and much more. According to at least one source⁵ at the time of writing this article, PHP is the primary server-side programming language for about 77.7% of the internet.

With the arrival of PHP 8.x⁶ and its numerous improvements, the programming language is well-situated for current and future usage. In addition, using the internet via the World Wide Web has become a norm for humans worldwide. With so many people accessing the internet daily, PHP is on track to be utilized for a significant amount of time into the future. But what would happen if, one day, the number of people who routinely visited the World Wide Web declined? What would that mean for PHP? Would PHP see a decline in its use as a server-side programming language, as its primary client is the web browser? Possibly, but not necessarily. If such a scenario were ever to happen, one change that would most likely happen is that developers would need to start using PHP more as a universal language and less as just for web development. Considering the current state of technology, this scenario might be more likely than not to happen, as described in the next paragraph.

One thing that is certain in today's world is the ever-growing number of devices and technologies humans have access to. A quick glance at the last 20 years shows that there is an ever-increasing number of technologies for humans to interact with and use. Focusing primarily on major technologies, examples of these include: ecommerce (e.g., eBay, Amazon, etc.), social media (e.g., Facebook, Twitter,

Instagram, TikTok, etc.), smartphones (e.g., iOS, Android, etc.), augmented reality (AR), virtual reality (VR), and the Internet of Things (IoT) among others. Technologies associated with the first three examples have become a facet of daily life for many people in the world. PHP can be found in many of these. This is likely based on their associations with the web browser as a main access point for many of them. The latter three technologies (i.e., AR, VR, and IoT) have yet to find similar success. This appears to be only a matter of time, though. Not including the success of the Metaverse, or the lack thereof, multiple businesses are investing large sums of money into technologies related to AR, VR, and IoT. Examples of these businesses include Apple⁷, Alphabet⁸ (Google), Meta⁹ (Facebook), Microsoft¹⁰, and more (see The Metaverse: And How It Will Revolutionize Everything¹¹ for further details about what businesses benefit from these technologies being successful). When these technologies find their way into daily life, will PHP power them if the web browser is not their primary access point? Yes. Instead of being used for part logic and part templating though, PHP will be used purely for logic via Application Programming Interfaces (APIs).

Based on the need to adapt to increasingly new technologies in the near future, this article advocates for more developers to use PHP to develop more API-first systems. These types of systems appear to adapt to the introduction of new systems much more easily as compared to systems that allow for cross-platform development, such as Ionic¹². This is due to not being able to fully predict the methods of how future client technologies will be implemented and render visual content. Instead, it assumes future technologies will continue to support methods of passing information in the form of JSON or XML, along with the ability to connect to remote web servers. Doing so would allow one central system to be responsible for the business logic of a whole application and for that application to be rendered using multiple different

1 <https://www.php.net/manual/en/history.php.php>

2 <https://wordpress.com>

3 <https://www.drupal.org>

4 <https://phpa.me/engineering-fb>

5 <https://phpa.me/php-usage-0321>

6 <https://www.php.net/releases/8.0/en.php>

7 <https://www.apple.com/augmented-reality>

8 <https://arvr.google.com>

9 <https://www.meta.com/quest>

10 <https://www.microsoft.com/en-us/hololens>

11 <https://www.matthewball.vc/metaversebook>

12 <https://ionicframework.com>

clients (including those that have yet to exist). This article will share an overview of the technical components of an API and how they can be used for multiple clients, including future ones—all to advocate for more APIs designed using PHP.

Application Programming Interface

An API is the interface two computer-related entities use to exchange information. This exchange is akin to a conversation by people where each entity involved sends messages explaining what they want to be done or were able to do. In the case of web and software development, an API is how two systems communicate in order to share data and evoke logic. Typically, these types of APIs use Representational State Transfer (REST) architecture¹³ via the Hypertext Transfer Protocol (HTTP)¹⁴ to represent the data they are sharing or to tell a system which logic should be evoked. The different elements of the HTTP protocol used during this exchange are equivalent to the different words people use when communicating together. The varying combinations of HTTP elements have different meanings, similar to how the varying combination of spoken words has different meanings. The elements that make up an HTTP message include the following:

- Method¹⁵ (e.g., GET, POST, PUT, DELETE)
- Version (e.g., HTTP/1.1)
- Path (e.g., /article/ as in www.example.com/article/)
- Headers¹⁶ (e.g., Content-Type: multipart/form-data)
- Body (e.g., HTML, JSON, XML)
- Status code¹⁷ (e.g., 200, 400, 500)

The *method* element is a sort of “verb” that explains what actions should be performed as part of a message when received. These actions are explained more in the later CRUD SQL Commands section. Each *version* of the HTTP protocol has more features and compatibilities than the preceding ones for the purpose of sending information. The current supported versions of HTTP include 1.1, 2, and 3. The *path* element indicates which resource should be sent back to whoever sent the message or what logic should be evoked by the respondent of the message. *Headers* are the metadata of an HTTP message which can include: authentication settings, options for caching, the user-agent who sent the message, and more. The *body* represents the resource being asked to be processed by a message or the resource requested in a previous message. The *status code* represents whether an action was successfully completed or failed, along with the reason in the form of numbers. Numbers between 200 – 299 indicate the message was processed correctly. Numbers between 400 – 499 indicate an invalid message was sent and how to correct

it. Numbers between 500 – 599 indicate the receiver of the message had an error while processing the message.

Client-server Model

While the HTTP protocol has multiple elements, not all are used for each API command. What elements are used can be explained by using the client-server model (Figure 1). A *request* is the HTTP message a client sends to a server when it wants data or to evoke logic. An example of this is when a web browser requests the content of a webpage from a server, such as its HTML, CSS, JavaScript, etc. Each file required to render the page fully needs its own request. This results in multiple requests being needed for one webpage. A *response* is what a server returns after processing a client request.

Figure 1: The client-server model

As a request and a response serve different functions, they each use specific HTTP elements. These elements can be seen in Table 1. The element missing from an HTTP request object is the status code. This makes sense as requests do not need to indicate their status of being able to reach a server or not. The only detail needed to describe the status of a request is whether a server receives said request or not. Elements commonly missing from HTTP response objects are the method or path. The combination of the method and

Figure 1.

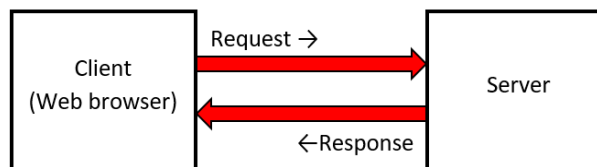


Figure 1: The client-server model

path elements tells a system the specific data that needs to be returned or what logic to evoke. Clients are the ones who ask servers to do this, so there's no need to include these in a response.

Table 1: Which HTTP elements are associated with request and response messages

| HTTP Protocol Elements | HTTP Request | HTTP Response |
|------------------------|--------------|---------------|
| Method | X | |
| Version | X | X |
| Path | X | |
| Headers | X | X |
| Body | X | X |
| Status Code | | X |

¹³ <https://phpa.me/wikip-rest-state>

¹⁴ <https://phpa.me/developer-mozilla>

¹⁵ <https://phpa.me/developer-mozilla/Methods>

¹⁶ <https://phpa.me/developer-mozilla/Headers>

¹⁷ <https://phpa.me/developer-mozilla/Status>

Crud SQL Commands

Often in simple APIs, request methods are associated with database functions. These functions are commonly known by an acronym called CRUD. This acronym stands for Create, Read, Update, and Delete. If these words were to be mapped to the different functions of a database system (e.g., MySQL), they would correspond to the INSERT, SELECT, UPDATE, and DELETE commands, respectively. In many instances, not every function is necessary to implement. Which methods and their associated mapping is often a design choice developers make when creating or modifying an API. Resources related to this topic are shared at the end of this article.

Restful API

The combination of HTTP elements, client-server model, and CRUD SQL commands allows for establishing a final API design (see Table 2). Typically, a single API is designed around representing a specific entity (e.g., nouns such as people, places, or things) or a collection of entities, along with the different functionalities associated with said entity. If an individual API contains multiple entities, proper design is necessary as it helps guarantee that the API is not responsible for too much.

Table 2: The basics of a RESTful API

| HTTP Request Path 1,2 | HTTP Request Method | SQL Command |
|-----------------------|---------------------|---------------|
| /noun/ | GET | SELECT * FROM |
| /noun/:id/ | GET | SELECT * FROM |
| /noun/ | POST | INSERT INTO |
| /noun/:id/ | PUT | UPDATE |
| /noun/:id/ | DELETE | DELETE FROM |

1 *noun* is the item represented by API and commonly modeled in the database. 2 *:id* is the unique identifier for a specific row of data in the database. Dynamic routing parameters like these are a common feature of routing libraries like Symfony Routing¹⁸.

Using an API with Multiple Clients, Including Future Ones

In order to illustrate how the use of an API can lend itself to future clients, this article will use the API in Table 3. This API is a subset of functionality for a fictional music streaming service similar to Spotify and Pandora. At a minimum, the API has the following actions associated with it:

- View all of the songs available through the service

- View the details of a song by ID
- View the details of a playlist by ID
- Add a new song to a playlist by using both playlist and song ID
- Delete a song from a playlist by using both playlist and song ID

Whenever the API is evoked, it affects the underlying database structure displayed in Figure 2.

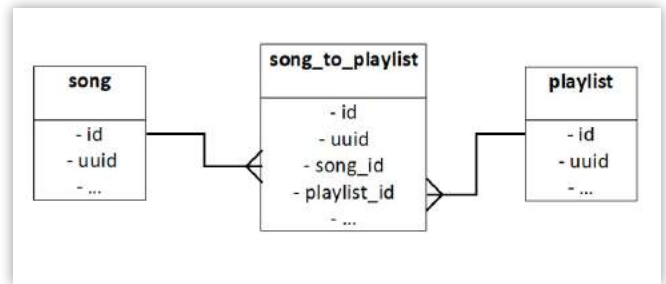


Table 3: A simple API designed to model the functionality for a fictional music streaming service

| HTTP Request Path | HTTP Request Method | SQL Command |
|---|---------------------|---|
| /song/ | GET | SELECT * FROM song |
| /song/:songId/ | GET | SELECT * FROM song WHERE id = ? |
| /playlist/:playlistId/ | GET | SELECT * FROM playlist WHERE id = ? |
| /playlist/:playlistId/addSong/:songId/ | POST | INSERT INTO song_to_playlist (song_id, playlist_id, ...) VALUES (?, ?, ...) |
| /playlist/:playlistId/removeSong/:songId/ | DELETE | DELETE FROM song_to_playlist WHERE song_id = ? AND playlist_id = ? |

For websites to evoke the API, there are a number of methods to do so. The initial method would be to simply enter the URL of the API into the address field of the browser (e.g., www.example.com/song/). Using the address field only lends itself to HTTP requests with the GET method. To use other methods such as POST, PUT, and DELETE, JavaScript can be used to send HTTP requests programmatically. Two common methods include using XMLHttpRequest objects¹⁹ and the newer fetch API²⁰. These allow users to define the method, path, headers, and body for a request before sending

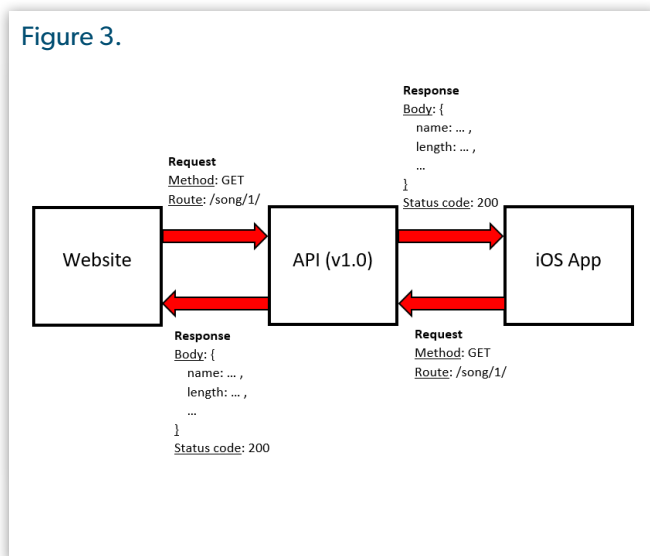
18 Symfony Routing: <https://symfony.com/doc/current/routing.html>

19 XMLHttpRequest objects: <https://phpa.me/developer>

20 fetch API: <https://phpa.me/developer-mozilla-web>

them and acting upon the elements of the response object. Most programming languages have different options for sending HTTP requests (e.g., PHP has cURL²¹, Guzzle²², etc.). In using a system like this, the music streaming service would be able to support web and mobile application clients (see Figure 3).

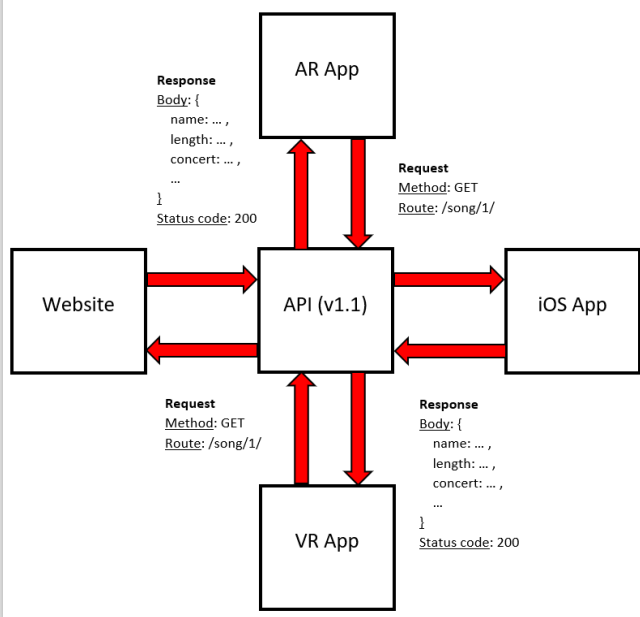
Figure 3: Diagram showing the client-server model for music streaming service



Now imagine the music streaming service has been successful for multiple years and is ready to take the next big step in music entertainment. They believe concert-like experiences via AR and VR are going to be the next big thing based on the live concerts happening in video games²³. Whenever a listener hears a song via their application that is downloaded onto their AR and VR devices, the service wants listeners to see a fictional or past concert experience related to the song. How could the service use its existing API to support AR and VR clients like this though? One method could be to update the responses from the API to contain the necessary data for AR and VR clients to display the concert experience. AR and VR devices with the application could then evoke the API services using the same HTTP requests described previously in this article. This would update the client-server model for the music streaming service to look like Figure 4. If the music streaming service had another idea on how to use a future client, it could repeat similar steps as the ones just described.

Figure 4: Diagram of the updated client-server model for the music streaming service

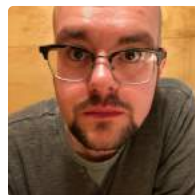
Figure 4.



Conclusion

The World Wide Web and websites are not going away anytime soon, so PHP will continue to be used. One day though, websites may go the way of the newspaper, radio, and television. They may just exist in an ever-evolving media culture. That would mean PHP would no longer be on the frontier of technology. To prepare for such a world, one method PHP developers could use to prepare for the future would be to use APIs to make their systems. This would allow them to be more available for any new clients that may be developed in the future.

The example APIs discussed in this article are limited in scope for the purposes of the discussion being had. For more in-depth information about web-based APIs, the author suggests viewers read *Principles of Web API Design: Delivering Value with APIs and Microservices* by James Higginbotham²⁴.



I am a Clinical Assistant Professor for the Games, Interactive Media, and Mobile (GIMM) program at Boise State University. The main courses I teach include full-stack web programming, the Internet of Things, and helping students prepare for the job market. Before earning my doctoral degree in Human-Computer Interaction, I worked as a web programmer for +10 years using PHP. @jack_polifka

21 cURL: <https://www.php.net/manual/en/intro.curl.php>

22 Guzzle: <https://docs.guzzlephp.org/en/stable/>

23 <https://phpa.me/axios>

24 <https://phpa.me/amazon-PrinciplesWebDesign>

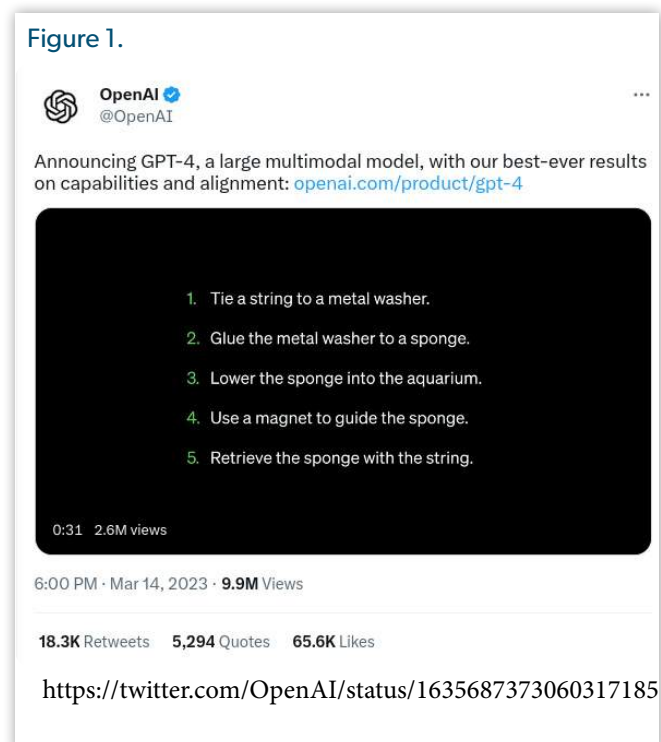
GPT: Changing The Way We Code

Tomas Votruba

It's not even 24 hours since GPT-4 was released to the public, and I'm writing this article with my own bare hands. I won't lie to you; I'm tempted to generate the content of this post. But the GPT is pre-trained on the public domain and not my brain...yet.

I want to share my experience with writing a startup based on GPT, AST, Rector, regular expressions, and ECS. Today, I want to show you how to work with GPT to generate code for you and what questions to avoid. (See Figure 1)

Figure 1.



When I first pitched this article to John, it was supposed to be about the exact technique, almost a tutorial, on how to ask GPT to generate a unit test for your code. Two weeks later, this already feels like old information, and thinking about three weeks more to publish the post, it might be as obsolete as “What is new in PHP 8.0 and how to prepare for it”.

If you're interested in an exact tutorial about how to run GPT on your machine and get an answer, I already wrote a few pieces like:

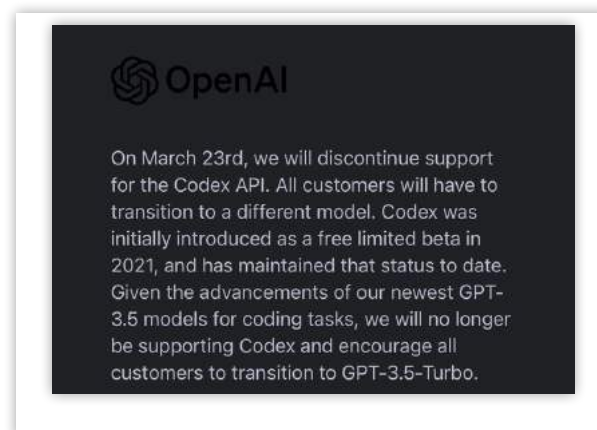
- How to ask DaVinci and Codex to get the right answer¹
- Let's Share Fails and Tricks with GPT²

I've been inspired by Marcel Pociot³ in person and online. He's an amazingly open person who knows how to use words to describe the practical step-by-step learning process in an entertaining and easygoing way. In saying that, he wrote a few practical tutorials to learn from:

- Build a ChatGPT clone using the new OpenAI Chat API⁴
- Fix your Laravel exceptions with AI⁵

GPT Changes So Fast

Originally, I wanted to write about a “codex”, which is a pre-trained model focused on code-related prompts. Similar to “Generate a unit test for this PHP code using PHPUnit with 4 items in data provider:”.



Today, I got a message from Marcel that this model will be depreciated. Not in a month or a week. But on March 23rd—in 2 days! Open AI and GPT change so fast that we must also adapt at the same pace.

We're used to waiting a year for the new PHP, a year for the new Laravel, or six months for the latest Symfony version. Every year we have one or two major upgrades that we have to get ready for. This is changing now and **will be much faster in the future**. Let's say you have two million lines of code PHP on a project using PHP 7.2 and want to get to PHP 8.2. In 2018, when Rector was just starting, this could take about

¹ <https://phpa.me/tomasvotruba>

² <https://phpa.me/tomasvotruba>

³ <https://twitter.com/marcelpociot>

⁴ <https://phpa.me/beyondco>

⁵ <https://phpa.me/beyondco-blog>

2-3 years. Today the same upgrade takes 1-2 months of full-time work. Next or even this year, this could be connected with GPT capabilities and be a matter of a single work week.

Can you imagine what this means for the evolution of software? Legacy projects will become sustainable internally. Services will integrate feedback from tools like PHPStan or Rector and will automatically propose a fix for the reported error via a pull-request. It will be merged automatically once the pull-request passes. You finish your work at 17:00 (5:00 p.m.), and the next morning at 9:00 a.m., you have 30 merged pull-requests. Not just random pull-requests, but pull-request that meet your quality standards and be written the same way you'd write them with zero energy and only need to be reviewed. If you don't like one of them, click on revert, and your GPT model will learn to do it better next time.

What are GPT Start-ups Really?

When I speak with non-developers, they assume GPT startups are very complicated projects, using complex mechanisms to use GPT to handle tasks. What is more interesting, when I speak with developers, they think GPT startups contain some wrapper around GPT REST API, business logic, repositories, and managing the output.

What if I told you that using the GPT is five lines of work? Once a month, I meet with David Velvethy⁶, an amazing friend and OpenAI expert with a realistic vision and the ability to translate ideas into working ideas. He introduced me to the GPT in November 2022, and I've never left since. I'm so grateful for him in my life. He helps me notice things I could never discover myself and gives me the confidence to look, go and do it. Every single conversation with you is like an expansion grenade to my mind; I love it.

Together, using just ChatGPT and prompts, we created **the smallest demo possible**. It has a single feature—ask GPT and get a reply. We aimed at having minimal code so that any developer and non-developer could understand it. We call it Dave GPT⁷, and this is the full index.php: (See Listing 1)

Note for security geeks—no worries, the API key is already invalid, and it is pointless to type it letter by letter and verify—I'm telling the truth :)

What part of the code is GPT API?

```
$client = OpenAI::client('sk-bZVLyQZEoCMx0y1h2nT3...');
$response = $client->completions()->create([
    'model' => 'text-davinci-003',
    'prompt' => $question
]);
return $response->choices[0]->text;
```

That's it! There is nothing simpler than this:

- You get a key, open the door and ask the question.

Listing 1.

```
1. <form method="post" action="/">
2.   <label for="textarea">
3.     Type your question:
4.   </label><br>
5.   <textarea id="textarea" name="question"
6.     rows="4" cols="50"></textarea><br>
7.   <input type="submit" value="Submit">
8. </form>
9. <?php
10. require_once __DIR__ . '/vendor/autoload.php';
11. if (isset($_POST['question'])) {
12.   $question = $_POST['question'];
13.   $answer = generate_answer_with_gpt($question);
14.   echo "<h1>Your question was: $question <br></h1>";
15.   echo PHP_EOL;
16.   echo "<h2>The answer is: $answer</h2>";
17. }
18.
19. function generate_answer_with_gpt(
20.   string $question
21. ): string
22. {
23.   $client = OpenAI::client('sk-bZVLyQZEoCMx0y1h...');
24.   $response = $client->completions()->create([
25.     'model' => 'text-davinci-003',
26.     'prompt' => $question
27.   ]);
28.   return $response->choices[0]->text;
29. }
```

- You get the answer, display it to the user and charge a fee.

Can you see the possibilities we have with such a simple script? Let's brainstorm a few startups right here:

Stanfix AI

In this project, GPT will:

- Get a PHPStan error
- Get a file that is related to the code
- Create a bash script that can create a branch in git and push it to Github
- vCreate the pull-request using the API key we provide to it

Autorect AI

This project will:

- Go through your code and find the PHP version in composer.json e.g. 7.2
- Get the goal PHP version as input - e.g. 8.2.
- Apply one Rector rule by another, to upgrade the code from PHP 7.2 to PHP 8.2.
- Opens a new pull-requests per rule that GPT generates, with a push-only token from your GitHub, Gitlab or Bitbucket project.

6 David Velvethy: <https://www.linkedin.com/in/david-velvethy/>

7 Dave GPT: <https://github.com/TomasVotruba/dave-gpt/>

In 10 minutes, you have 50 pull-requests in your project, you go through them and merge them manually.

Testgen AI CLI

I work on a project, testgenai.com⁸, that can take a PHP class as input and generate a unit test for the public method of your choice. It's easy, simple, and great for working with code you don't even want to read.

Instead of paying for a full-day workshop training with theoretical examples of writing unit tests with little practical value, why not save with an on-demand, more practical service? In 10 seconds, you can generate units tests generated specifically for your PHP code ready for copy and pasting.

Let's brainstorm and take this one step further. What if we had a CLI tool that we could run like this:

```
vendor/bin/testgenai generate --limit 20
```

It would go through all public methods and find the 20 that are the easiest to test but are not covered with an existing test. For each such public method, it would generate a unit test. Thanks to the parallel prompt run, it would take about 30 seconds in total. Then it would open 20 parallel pull-requests, so we can still review it manually with ease and clearly understand what code we accept to our codebase.

We then merge 18 of those and provide feedback for the 2 left. Then we re-run again and again—we soon increased our code coverage from 5 % to 40 %, and it's not even lunchtime.

How would you like those 3 projects in your developer toolkit for 33 € (\$36.15) monthly?

- running non-stop
- on demand
- review on a single click
- running until it is finished... or the Internet crashes

I'd be like: take my money!

Find the Added Value and Deliver

Ideas are cheap. Execution is expensive. The ability to execute separates people, not the ability to come up with ideas.

We've all probably had similar ideas to those above in the past three months. It's very easy to come up with an idea of what the next startup would do.

How Long to Make Testgen AI?

To share my experience: the idea to create some kind of unit test generator came around December 2022. I was out for a beer with my friend Kerrial, and we discussed how painful it is to work on legacy projects without any tests. Any change with Rector or ECS on such code could be risky. To move fast,

you have to move safe. We want to remove any risk and be able to scale changes with parallel pull-requests. It would also give our clients⁹ the confidence to move faster and develop business features faster too.

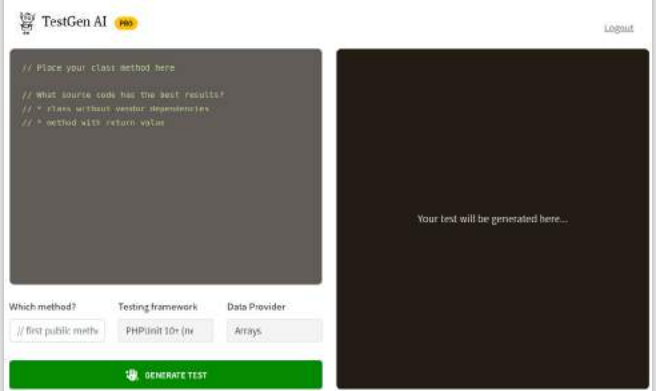
The idea was developed in early December. It was early to mid January when I first made a working prompt like the one above. There was a lot of experimenting:

- Trying different prompts to GPT
- Designing the input form and output code for a nice user experience
- Playing with the lazy response time

I launched the project on February 10th and was ready to scale. Yet, I got excited about Laravel, and its features allowed me to run queues in the background lazily. Saying that, I migrated the project from Symfony to Laravel¹⁰—my first migration of this kind. Only after this could I work on the features further and improve the project.

One way or another, it took roughly 40 days from the idea to first beta testers and 60 days to public use. Knowing what I know now, I could launch in a week, as I learned a lot about the process, payment setup, taxation, marketing, and product launches.

Figure 2



Do You Have an Idea?

Most of the work above didn't take the 5 lines of GPT but the value around it. The marketing, the input validation, and the background queues allow you to submit two prompts simultaneously without waiting 3-6 seconds for a response and reacting to user feedback.

Take a prompt, try it out in the Chat GPT, and if it works, create your first project based on some GPT skeleton.

Go for it and iterate from there. All you need is an Open AI token, and you're ready to go.

⁸ testgenai.com: <https://testgenai.com>

⁹ <https://getrector.com/hire-team>

¹⁰ <https://phpa.me/tomasvotruba-blog>

How to ~~Replace~~ Share Yourself?

Today I met with David Velvethy¹¹ for a coffee, and we talked about the future of GPT, AI, smart contracts, bank systems, and aliens, of course.

I mean, the startups above bring real value, the pricing is very cheap compared to the value they save, and it's quite easy to make them. I estimate it would take around 1-2 weeks of full-time work on each of those first to launch.

When I spoke with Dave, I felt we could do more than input-output tools and handle single tasks. I mean, I love the Unix philosophy: *Program does one thing well* and *The program output can become the input to another* —that's how REST API can work easily with JSON.

...but!

There is more in each of us. Today we have tools to share our specific know-how and automate it to scale. What I really love about what I'm about to describe is that it takes the best out of every single human and puts it out there.

A friend of mine is creating patterns for clothes that you can cut from the fabric at home and sew yourself. When I spoke with her for a few this evening, we came to the idea of using a GPT program that would be able to design upcoming trends and show a few rough designs that would be manually tuned and creatively polished to perfection. 80 % of time saved, more time for new projects.

For me, such GPT self-extraction would include my daily work such as sending pull-requests to improve code quality, to upgrade frameworks, to introduce constants for repeated strings, for removing PHPStan errors, to prepare extraction of unused or deprecated packages or to remove unused code.

For you, this "GPT clone", could be taking care of an e-commerce project, adding new features, removing the least used ones, or abstracting layers of the security architecture that grew too far.

GPT Scales You

Imagine you could do your daily work in the background, such as when you're having dinner or sleeping. Yet, you'd still be in control; you could think about the process, self-reflect on every step and fine-tune the results. This allows you to grow further as a human and help each other more with our natural gifts, doing what we love. That's my inspiration for you and my hope for how humankind approaches using AI. If we take it right, I know we can create a beautiful world full of love.



Tomas Votruba is a regular speaker at meetups and conferences and writes regularly. He created the Rector project and founded the PHP community in the Czech Republic in 2015. He loves meeting people from the PHP family so he created Friend-Of-PHP which is updated daily with meetups all over the world. Connect with him on twitter [votrubaT](#). [@votrubaT](#)

¹¹ <https://www.linkedin.com/in/david-velvethy/>

php[architect]
[consulting]

- Get customized solutions for your business needs
- Create a dedicated team or augment your existing team
- Improve the performance and scalability of your web applications
- Leverage the expertise of experienced PHP developers
- Building cutting-edge solutions using today's development patterns and best practices

consulting@phparch.com



12 Factor Applications: Parts 7-12

Chris Tankersley

Last month we started looking at 12 Factor Applications, a set of rules for building robust, maintainable distributed applications. Developed by Heroku Engineer Adam Wiggins in 2011, it raised some important questions for distributed applications.

Many of the questions posed in 12 Factor Applications generally work for web applications. We looked at the first six tenants already, which mostly revolved around how we should store and configure code. Tenants seven through twelve will deal much more with how we architect and think about our code.

7. Port Binding

“Export services via port binding”

The type of service your application is providing should be designated by the port that it binds to, and this should be the discovery mechanism for how to deal with an application. While there are many ways to document this, ports are a pre-defined way to determine what service is available on a port.

What this means is that if you are serving an HTTP application, your application binds to and listens on port 80 or 443, the traditional and designated HTTP and HTTPS ports. If your application is purely a PHP application running behind PHP-FPM, you would bind to and expose yourself on port 9000, the standard PHP-FPM port. The Internet Assigned Numbers Authority¹ keeps a list of reserved ports up to around port 50,000.

Suppose you are deploying to a containerized system. In that case, your container can detail what ports it is listening on and allow the orchestration layer to bind other ports to it via redirection. For example, if you are deploying using the `php:apache` container you can expose ports 80 and 443 on the container. However, Docker or Kubernetes may assign random ports for public accessibility and redirect

those ports to the container’s 80 and 443 ports.

Binding and announcing via ports allows a standard way for services to be determined. For PHP applications themselves, this is mostly a non-issue and is a problem more for the container or production service itself. It is not foolproof, as most systems do not enforce port binding beyond “one application to a port.”

8. Concurrency

“Scale out via the process model”

12 Factor Applications should scale horizontally by implementing more processes, not vertically by requesting more resources. This is not uncommon in web applications already, where for many years, TCP stack limitations have been much more of an issue than how much RAM or CPU a PHP script consumes. In fact, it is so common PHP is designed to run per process anyway and let external systems handle the process creation. It is only a recent problem where our applications have to worry about concurrency.

PHP’s origins lie within running as a CGI process² or through the Common Gateway Interface. This was a way for web servers to execute external programs and return the result. When a web server got a request for a PHP script via CGI, the web server started a new PHP process and ran. PHP was concurrent via CGI.

Fast forward to Apache’s `httpd` web server and `mod_php`. This module directly embedded PHP processing into `httpd`’s engine, so each `httpd` thread was now also a PHP thread. PHP gained

concurrency through `httpd`’s own internal thread handling.

As CGI fell out of favor for FastCGI processes, PHP introduced PHP-FPM. This is a wrapper that keeps a few PHP processes around and ready to handle requests, but the idea is still the same. Multiple PHP processes are executed in parallel and scaled by adding new processes.

Container technologies just build on these same fundamentals. Need to handle more connections? Spin up another container and scale out horizontally.

Other languages that run as a single thread, like Node.js, also have additional complexity in their logic to handle concurrency. They need to make sure to separate request data between requests, as the script may be handling multiple individual HTTP requests simultaneously. You cannot store “the current user” as a global object like you can in PHP, as the Node.js process is not a “fire and forget” architecture.

These design decisions will become more prevalent as PHP moves toward more single-threaded, long-running asynchronous processes, like with Swoole or ReactPHP. You will need to actively think about what it means to scale an application to run as multiple processes.

This is on top of the traditional questions like what happens if your ReactPHP application encounters a user that is logged in? How do you handle sessions or file uploads? PHP applications that scale horizontally need to be aware of issues like file storage, distributed caches, and remote sessions.

¹ <https://phpa.me/iana-assignments>

² <https://phpa.me/wikipedia-CGI>



9. Disposability

“Maximize robustness with fast startup and graceful shutdown”

12 Factor Applications are designed to be started, stopped, and destroyed at a moment's notice. They favor fast startup times, clean shutdown routines, and the ability to update whenever a new configuration is pushed out.

This is another problem that is already somewhat solved by virtue of how PHP is designed. Traditional PHP applications started with a request, execute, and then end with the request. There are normally no long lifetimes for a PHP script, unlike Node.js processes or Java applications. We almost never “start” a PHP application or “stop” one, but we may start or stop a web server. If you are in this situation, congratulations!

If you are running a newer-style application that takes advantage of asynchronous, single-threaded applications like OpenSwoole or ReactPHP, these do not prescribe to the start-execute-die philosophy. They require explicitly starting a process, and that process lives until it is told to stop. Requests are handled asynchronously by a single thread (which may or may not generate its own threads, but generally, we are talking about one entry point for an application).

Applications like this should ensure their startup times are as quick as possible. If they are caching configuration or doing many setup steps, ensure that they are accomplished as quickly as possible. The quicker an application comes online, the faster your application can scale.

For shutting down, these PHP scripts should implement signal handling. This allows POSIX signals³ to be sent to an application and have the application intelligently respond. Did your application get a SIGTERM signal? Stop accepting requests, finish any current ones, and then stop. Receive a SIGHUP? Reload your configuration without stopping.

`pcntl_signal()`⁴ allows a script to register handlers for different signals. While it is beyond the scope of this article, I would suggest registering handles for SIGTERM and SIGHUP, the two most common “stop” or “restart” signals. Think about how your application can cleanly end execution or do a soft restart to cause the least amount of disruption to users.

10. Dev/Prod Parity

“Keep development, staging, and production as similar as possible”

One of the oldest issues with programming is “it works on my machine.” Well, it may not be the oldest since many people developed on and deployed to the same shared multi-million dollar machine in the early days of computing. Still, it is an ever-present problem in modern computing.

We should strive to make sure that our development, staging, and production environments are the same. I do not mean similar, but the same. Doing so helps reduce issues relating to small environmental differences with third-party applications or even big ones, like Windows versus Linux.

If you are deploying to bare-metal servers, look into using configuration systems like Puppet or Ansible. These allow you to programmatically define how systems are set up in configuration files. This is a methodology known as “Infrastructure as Code.”⁵ These configurations can be applied to any machine to ensure they always match.

These scripts can also be run against virtual machines and are traditionally how local development systems like Vagrant made sure virtual machines stayed consistent through builds. Tools like Terraform are designed specifically to work with virtual machines and hosted virtual platforms like AWS.

Containers and their images now tend to fill this niche. Images are built from definition files that describe exactly how to construct them, and containers are all copied from these

images when they are created. Parity is kept because all installations can point to the same image, and each image is a byte-for-byte replica.

Having environmental parity is important as it saves both developer and deployment times. Developers can hop back to older images to test bugs or not waste time with outdated local environments when production needs change. Did production just update to PHP 8.2? Update the configuration, so everyone pulls that production image.

Personally, keeping environment parity is not an excuse for robust programming. By being pragmatic about the solutions we devise, we can solve many problems. Our applications will be more robust if we design applications to run under various environments. Saying, “This application only runs on PHP 8.0.16 with MySQL 8.0.12 on Ubuntu 23.04”, and building containers around that is not a solution we should be striving for.

11. Logs

“Treat logs as event streams”

From the perspective of a 12 Factor Application, all logs should be directed to stdout (although, as a personal hot take, applications should be directing their output to the most appropriate standard stream). stdout is one of the standard streams⁶ that POSIX environments define as writeable file descriptors⁷. These streams provide input and output paths for a program and its operating environment.

The reason for logging into a standard stream instead of something like a log file is to allow flexibility in the running environment. For development, having your PHP application log to stdout or stderr makes it much easier to launch the dev server and just watch the logs as the application runs. All the logging goes to the terminal that the developer is watching.

In a production environment, these streams can easily be redirected by the environment. For many hosting

3 <https://phpa.me/dsa-cs-tsinghua>

4 <https://phpa.me/php-net>

5 <https://phpa.me/wikipedia-infrastructure>

6 <https://phpa.me/wikipedia-standard>

7 <https://phpa.me/pubs-opengroup>



solutions, the `stdout/stderr` output is pushed up to the web server, and the web server decides what to do. In other cases, PHP is directed to log all of that output to its own separate file.

The key is that the application itself is as accommodating as possible with logging and does not define the ultimate location or process for logging—it simply outputs to a known, accepted interface that works with almost anything.

For PHP applications, this can be accomplished in a few ways. The quickest and easiest way is to start adding calls to `error_log()`⁸ in the appropriate places in your application. This function, by default, writes to whatever PHP has been configured to be as the error log. This can be `stdout`, `stderr`, a file, or whatever. The function lets the PHP engine handle it all.

This is appropriate yet somewhat heavy-handed in that either you, as the developer, need to manually add in checks to see if something should be logged (like has the user configured debug-level logging, or just warnings), or you end up logging everything. While disk space is a cheap commodity today, a developer's time is not. Reading through millions of lines of logs can be exhausting, even with automated tools.

A better option is implementing a PSR-3⁹ compatible logger. 99% of the time, this means implementing `monolog/monolog`¹⁰, PHP's most widely used logging system. Monolog allows a developer to log messages at different pre-defined levels, like debug, info, error, or emergency. Then, the user can decide what levels they would like to see logged.

While Monolog can take a variety of different logging methods, if we take 12 Factor Application design into account, we would want to log to `php://stdout`. If you want to take proper Unix design patterns, you would want to log to `php://stderr` instead. Either case will work.

Why do I keep saying to use `stderr` instead of `stdout` for logging? The way the standard streams are designed, `stdout` is meant for application output, meaning the result of the program. If you are writing a script that calculates the number of words in a document, you would output the result, say 3000, to `stdout`. This allows program output to be easily redirected to another process. Based on the name, many people assume that `stderr` is only for errors. `stderr` is meant not only for errors but also for diagnostic output. Logs, by definition, are diagnostic output, not program output, so the more appropriate output is `stderr`.

Monolog loggers can be passed around to various parts of the application to allow for logging. Creating a handler requires only two lines of code: one to create a new logger, and a second to add a Handler. The Handler is an object that takes the log message and outputs it somewhere. Since PHP allows us to write to `stderr` via a stream, we can create a `StreamHandler` that points to `php://stderr`, the stream URI for `stderr` in the operating system. (See Listing 1)

Listing 1.

```
1. <?php
2.
3. use Monolog\Level;
4. use Monolog\Logger;
5. use Monolog\Handler\StreamHandler;
6.
7. // create a log channel
8. $log = new Logger('my-app');
9. // Log only anything higher than a Warning level
10.
11. $stream = new StreamHandler(
12.     'php://stderr',
13.     Level::Warning
14. );
15.
16. $log->pushHandler($stream);
```

The `$log` object can then be stored in something like a dependency injection system and passed around your application. You can use the logger to log a message whenever you need to log something. The Handler will determine if it is a high enough level to be output, and where (in our case, `stderr`).

```
// add records to the log
$log->warning('Foo');
$log->error('Bar');
```

If we are deploying to a system like Docker or Kubernetes, they tend to watch `stdout` and `stderr` by default. They can pick up these streams and redirect them even further to another logging system. Our application does not care or dictate where the logs go. Our application just logs them to a known, standard output and lets other external systems handle it from there.

12. Admin Processes

“Run admin/management tasks as one-off processes”

This is one of those instances where the Heroku pedigree of 12 Factor Applications really shows. The idea behind this is that any administrative tasks, like database migrations, should be run as separate processes but inside the same environment as the running application. It should be a process, not a function of the startup process.

If you couple this with the “Dev/Prod Parity” tenant, you can execute commands inside the same exact environment as the running application. Just like with an application, commands run in a known environment against the same release and config provided to the main application. If you cannot follow the “Codebase” or “Config” tenants, it becomes impossible to ensure your admin tasks execute correctly.

⁸ <https://phpa.me/php-manual>

⁹ <https://www.php-fig.org/psr/psr-3/>

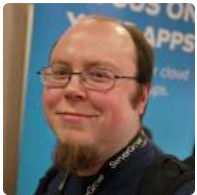
¹⁰ <https://github.com/Seldaek/monolog>



Is 12 Factor for You?

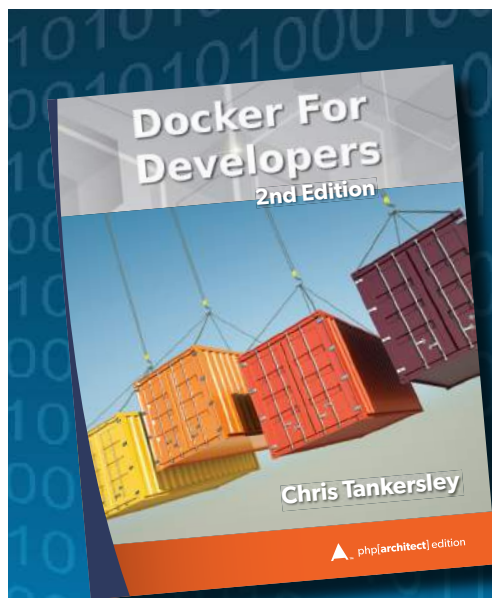
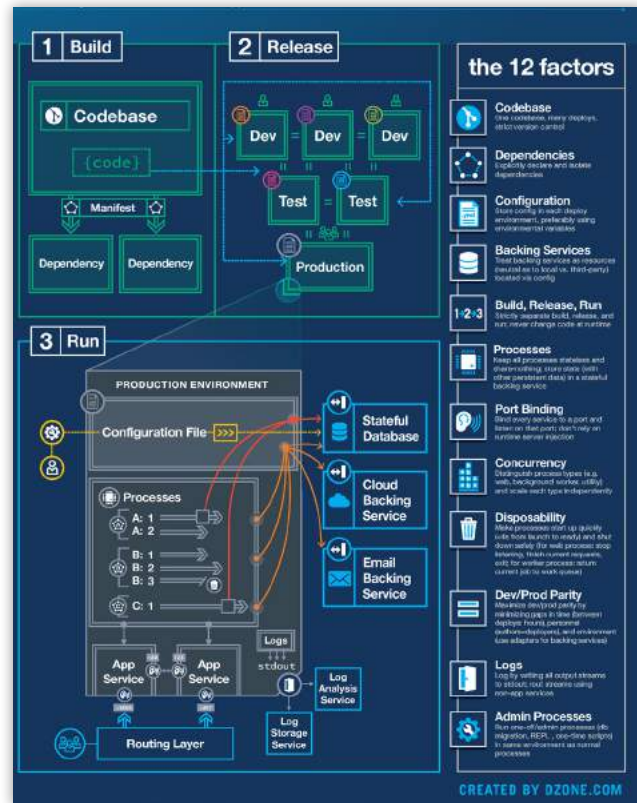
I hope you have enjoyed this dive into 12 Factor Applications. While it shows its age in a few places, in a few places we see where PHP was ahead of the curve; I think many of the ideas it lays out are still important in 2023. Maybe the nginx response with Microservice Reference Architecture¹¹ makes more sense in today's environment for you.

Building maintainable applications will never go away, and each application is unique. 12 Factor Applications still has some good information and generates some good questions for any developer about the application they are developing. As with any best practice, use what makes sense for you, but it never hurts to look back at old “standards.”



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

¹¹ <https://phpa.me/nginx-microservices>



Docker For Developers is designed for developers looking at Docker as a replacement for **development environments** like virtualization, or devops people who want to see how to take an existing application and integrate Docker into their workflow.

This revised and expanded edition includes:

- Creating custom images
- Working with Docker Compose and Docker Machine
- Managing logs
- 12-factor applications

Order Your Copy
<https://phpa.me/docker-devs>

The Risks of Free Conference Internet

Eric Mann

Now that the snow is melting, we're beginning to see the first signs of Spring. With Spring comes the rain, wildflowers and honeybees, bouncing bunnies in the park, and conference season. Traveling for conferences and other events can be exciting for many. But what most don't realize is just how risky it can be.

Conference Wifi

Most tech conferences boast free wireless connectivity for attendees. Normally, this means you can check email during breaks or follow along during coding demonstrations. It's great but easy to overuse. Once you connect to the wireless network, everyone else on the network can see you, too!

Think for a moment—how freely do you give away the wireless password for your own home? Once you connect to the conference network, all of the attendees have the same level of access to your machine that a guest would have in your home.

If you're also on a work machine, this could be a catastrophic security failure.

Development Environments

Consider for a moment that you use Docker to containerize your primary application. It's a standard PHP project that leverages MySQL as a data store. Your `docker-compose.yml` for local development might look something like the following: (See Listing 1)

Building your project locally will stand up your application and the database upon which it's dependent. Everything works as expected, and you're off to the races with development. Even though you're away from home, your local clone of the database lives in a persistent volume mount¹, so you can work even if the conference internet seems spotty.

There's only one problem...

Network Scanning

Nmap² ("Network Mapper") is one of my favorite free security utilities. This handy utility can scan any network you're attached to and automatically enumerate any hosts on that network by their IP address. You'll be able to find your own machine but also retrieve information about other computers that share your network space.

Listing 1.

```
1. version: "3.5"
2.
3. networks:
4.   front:
5.     name: front
6.
7. volumes:
8.   db_data:
9.
10. services:
11.   app:
12.     build:
13.       context: .
14.     restart: always
15.     ports:
16.       - "8080:8080"
17.     environment:
18.       SERVER_PORT: 8080
19.     networks:
20.       default:
21.         front:
22.
23.   sql:
24.     image: mysql
25.     ports:
26.       - "3306:3306"
27.     restart: always
28.     environment:
29.       MYSQL_ROOT_PASSWORD: root
30.       MYSQL_DATABASE: project
31.     volumes:
32.       - db_data:/var/lib/mysql
33.     networks:
34.       default:
35.         front:
```

If you know your own IP on the network, you can scan for other live hosts with the following one-line command:

```
nmap -sn 192.168.86.24/24
```

Once you have a list of live hosts, you can pick one (or two or more) and inspect the host directly. Nmap allows you to enumerate any open ports and will also attempt to expose the

¹ a persistent volume mount:
<https://docs.docker.com/storage/volumes/>

² Nmap: <https://nmap.org>



service listening on that port. Merely run the `nmap` command directly against the target IP address: (See Figure 1)

Figure 1.

```
ericmann@pop-os:~$ nmap 192.168.86.24
Starting Nmap 7.80 ( https://nmap.org ) at 2023-03-07 14:15 PST
Nmap scan report for ericmannbp16tb5.lan (192.168.86.24)
Host is up (0.0079s latency).
Not shown: 995 closed ports
PORT      STATE SERVICE
3306/tcp   open  mysql
5000/tcp   open  upnp
7000/tcp   open  afs3-fileserver
8080/tcp   open  http-proxy
49165/tcp  open  unknown

Nmap done: 1 IP address (1 host up) scanned in 5.19 seconds
ericmann@pop-os:~$
```

The preceding screenshot was taken directly from my Linux machine, which lives on the same wireless network as the Mac on which I'm running the `docker-compose.yml` file from earlier. The key thing to note—both MySQL and the application itself are visible on the network!

In fact, anyone on the same wireless network can now exercise the endpoints on my application or attempt to log into my local database. This would include *every single attendee on the conference network* at an event!

Understanding Exposure

Years ago at a conference, I accepted the challenge from one of the organizers to “hack an attendee live.” Doing so involved running through the entirety of the exercise you just saw:

- Use Nmap to enumerate hosts on the network
- Pick a host at random and enumerate the exports ports/services on their machine
- Guess the default username and password pairs for their exposed services (i.e., `admin/admin` or `root/root`)

It was a fairly effective demonstration. In under five minutes, I found several exposed machines. The first target was a WordPress developer working on a large e-commerce project. To avoid synchronizing PCI data from a production server over an untrusted network, they'd cloned their production database to a local instance so they could work offline.

Unfortunately, they exposed that database to the entire conference over a standard port with default (`root/root`) credentials.

We identified *which* attendee this was based on some additional information exposed by their machine (namely its server name) and contacted them directly during the event. They were a bit embarrassed to have accidentally exposed so much information, but we helped them lock their system down quickly.

It's not always easy to recognize when or how this kind of issue will come up. There is nothing wrong with the previously-referenced `docker-compose.yml` file. In fact, this is a default

configuration referenced in thousands of tutorials and even Docker Hub's own documentation on the official image for MySQL³!

There are two ways to prevent this kind of system exposure:

1. Use internal networking⁴ within the Docker host *only* to avoid exposing ports on the primary machine.
2. Specify an explicit IP address when exposing ports⁵ to avoid binding to `0.0.0.0`. By binding explicitly to `127.0.0.1`, you permit local access to the port and prevent remote users from talking to your containers.

In every case, understand exactly what you're exposing and where, even on a private network. While this will help better protect you the next time you're traveling for work, it will also minimize your exposure to potential compromises within your *home* network should they arise.

Preventing Embarrassment

While locking down port binding to just a local or trusted network is a great way to minimize your exposure to a fellow conference attendee, it's by no means the *only* way. In fact, the best way to prevent your machine from being popped by a rogue blackhat is to leave it at home.

If you're at a conference, be fully there. Avoid working on business requests. Keep your work laptop powered off—ideally, don't bring it in the first place.

Ultimately, though, avoid connecting anything important to the conference wireless network. Not your laptop. Not your phone. Not your tablet. Suppose you **do** need internet while you're on-site. In that case, tether to your mobile device and leverage the private nature (no one else at the conference is connected to your phone!) of the network to keep your traffic and environment secure.

Avoid an embarrassing conversation with the resident security expert by keeping your systems off a potentially exploitable network in the first place. It'll also go miles towards keeping your company, products, and customers secure!



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](https://twitter.com/EricMann)

3 the official image for MySQL: https://hub.docker.com/_/mysql

4 internal networking: <https://phpa.me/DockerDocs>

5 exposing ports: <https://phpa.me/DockerPorts>



A Grumpy Programmer's Introduction To NeoVim and PHP

Chris Hartjes

Hello friends! You might know me from my years of shouting at PHP developers to write tests for their code and my (hopefully) entertaining talks at conferences. It probably will not surprise readers that I also have some opinions about text editors and Integrated Development Environments (IDEs). In this article, I want to discuss The One True New Editor, NeoVim.

What is Neovim?

NeoVim describes itself as a “hyperextensible Vim-based text editor”. While this isn’t the most helpful description, I think we need to take a step back and look at its progenitor, Vim.

Vim (which stands for Vi Improved) is a “highly configurable text editor built to make creating and changing any kind of text very efficient”. NeoVim initially started as a project to “clean up” Vim by rewriting it with more modern languages and reducing the amount of “code bloat” that any long-running project can collect. The big selling point was the decision to rely on the Lua programming language for creating plugins instead of Vimscript.

My introduction to Vim was back in the early 2000s when I was trying to figure out how to use another controversial editor (Emacs) and a co-worker offered to show me why “Vim is better”. I found that Vim’s nature as a modal editor was a better fit for my own way of working, and I’ve been relying on muscle memory to guide my keystrokes ever since.

Along the way, I heard about an effort to “rewrite” Vim from folks who thought a text editor needed improving and disliked Vimscript’s idiosyncratic ways. Once NeoVim had a stable release, I installed it. I started using it as my primary text editor for everything other than PHP. I slowly started working on creating a NeoVim environment with the tools that I felt reflected how I programmed in PHP.

These days, NeoVim is a stable modal editor with a well-documented API (essential for any tool where people will want to create plugins and extensions) and a very active and passionate community surrounding it. I could probably write a very long article about the pros and cons of text-centric, modal editors, but this is a PHP magazine. Social media is a much better forum for arguing with people about the best way to exit Vim other than shutting off your computer. Instead, I want to talk about how I use NeoVim when working on PHP applications.

Build-your-own IDE

I am not the first NeoVim enthusiast to support the position that you can build your own IDE using it. Full disclosure—I am a paying user of PhpStorm because I believe in supporting the creation of useful tools for the PHP ecosystem, even if I am not going to be using it all the time. It is a great tool. I just prefer NeoVim and am willing to do the work to create a customized development environment for myself through the use of plugins and extensions that rely on 3rd party tools.

At a high level, I am looking to accomplish the following:

1. Rely on the use of Language Server Protocol (LSP) tools to analyze my PHP code
2. Make it easy to find files in my projects
3. Make it easy to find code in my projects
4. Always evaluate the use of tools and brutally cut out things that don’t work the way I want them to

I have written about my NeoVim configuration in the past on my blog, but it does evolve, so let me share what I am currently using. But it fits well with my philosophy of not committing time to tools I have to fight with. With NeoVim, I usually have a few options for how to add the functionality I am looking for. As good as many editors and IDEs are, your choices are often limited.

I am currently using a combination of Lua and Vimscript for my configuration, and in my opinion, most of it is very boring. The real magic is in my selection of plugins. Here is my current list of plugins that I am relying on. Don’t worry; I will review my choices and highlight the ones I have found helpful. Unless otherwise indicated, all the plugins are available via Github by searching for them by user/plugin name.

Packer

```
return require('packer').startup(function()  
    use 'wbthomason/packer.nvim'
```



Packer is what I use to manage all my packages. Over the years, there have been many different plugins to support packaging. I don't think you can go wrong picking one and sticking with it. In any case, it is usually pretty easy to switch. In my case, I chose Packer due to its easy integration with Lua.

Language Server Protocol

```
use 'neovim/nvim-lspconfig'
```

Microsoft initially created the Language Server Protocol (or LSP), which has been embraced by tool and plugin makers to integrate language support into NeoVim. In my case, I am using it for the following:

- PHP language support via the paid-for features of Intelephense¹
- Autocompletion with nvim-cmp²
- Typescript support (my preferred flavor of JavaScript when I do use it)

Much like my opinion that Composer saved PHP, I believe LSP saved the Vim plugin ecosystem. I would say that by using plugins and configuring NeoVim to use LSP-based tools, you get most of what PhpStorm can offer you in terms of syntax highlighting, linting, code traversal, and searching.

I have a separate configuration file for LSP-related settings. View the full configuration in Listing 1 on the next page.

In the “build-your-own IDE” environment, creating keyboard shortcuts is essential. I've been extremely fortunate as a 25+year developer to have not experienced any RSI injuries related to this work. NeoVim, along with an Advantage2 LF Kinesis ergonomic keyboard and Ploopy trackball, allows me to keep my hands on my keyboard 90% of the time when coding. So having keyboard shortcuts that don't require finger stretches or twisting my wrist to use a mouse is a big help.

After a while, muscle memory takes over, and I don't have to actually remember “Leader-fd is how I find the definition for this object”. This is really no different than learning the keyboard shortcuts that the developers of an IDE had already picked for you. Customizing your tools in a way that leads to better outcomes is always a good choice to make.

General Plugins

```
-- General plugins
use 'dracula/vim'
use {
  'nvim-treesitter/nvim-treesitter',
  run = ':TSUpdate'
}
use 'onsails/lspkind-nvim'
```

¹ Intelephense: <https://intelephense.com>

² nvim-cmp: <https://phpa.me/neovimcraft>

I like using the Dracula³ theme and also added in some help for Tree-sitter⁴ and pretty icons when interacting with LSP.

As I evaluate other plugins for editing things besides PHP, I will stick them in this section. I've recently been trying to find the right combination of plugins to allow me to use Xdebug from inside Vim in a way that, again, resonates with how I work. All that stuff would go in this section.

Version Control Systems.

```
-- See the git status of the current line in the gutter
use 'airblade/vim-gitgutter'
```

It has been a very long time since I had to use any other version control system than Git. There are lots of plugins available

Since most developers are using version control systems of some kind, being able to see the status of individual lines is very helpful.

PHP-specific

```
-- PHP plugins
use 'tpope/vim-dispatch'
use 'StanAngeloff/php.vim'
use 'stephpy/vim-php-cs-fixer'
use 'noahfrederick/vim-composer'
```

This section tends to change based on what type of PHP projects I have been working on. Right now, I am just using plugins for dealing with PHPCS-fixer and Composer. In the past, I have used plugins for helping to work on Laravel projects. Search engines are definitely your best bet when looking for framework-specific plugins.

Coding Style

```
-- Respect .editorconfig files for a project
use 'editorconfig/editorconfig-vim'
```

I'm a big believer in using EditorConfig⁵ to allow your editor to respect coding styles. You never want to be able to figure out who wrote some code by looking at it! I'm also a big fan of adding automated ways to format your code whenever you commit your changes or push your changes elsewhere. Automate everything you can to save your brain power for things like remembering how to actually quit Vim.

³ Dracula: <https://github.com/dracula/vim>

⁴ Tree-sitter: <https://github.com/tree-sitter/tree-sitter>

⁵ EditorConfig: <https://editorconfig.org>



Listing 1.

```

1. --- Configuration for LSP, formatters, and linters.
2. local nvim_lsp = require("lspconfig")
3.
4. -- short cut methods.
5. local t = function(str)
6.
7.   return vim.api.nvim_replace_termcodes(str, true, true, true)
8. end
9.
10. local opts = { noremap=true, silent=true }
11. vim.api.nvim_set_keymap('n', '<space>e',
12.   '<cmd>lua vim.diagnostic.open_float(<CR>)', opts)
13. vim.api.nvim_set_keymap('n', '[d',
14.   '<cmd>lua vim.diagnostic.goto_prev(<CR>)', opts)
15. vim.api.nvim_set_keymap('n', ']d',
16.   '<cmd>lua vim.diagnostic.goto_next(<CR>)', opts)
17. vim.api.nvim_set_keymap('n', '<space>q',
18.   '<cmd>lua vim.diagnostic.setloclist(<CR>)', opts)
19. vim.api.nvim_set_keymap('n', '<space>f',
20.   '<cmd>lua vim.lsp.buf.formatting(<CR>)', opts)
21.
22. local on_attach = function(client, bufnr)
23.   -- Enable completion triggered by <C-X><C-O>
24.   vim.api.nvim_buf_set_option(bufnr, 'omnifunc',
25.     'v:lua.vim.lsp.omnifunc')
26.
27.   -- Mappings.
28.   -- See `:help vim.lsp.*` for docs on the below functions
29.   vim.api.nvim_buf_set_keymap(bufnr, 'n', 'gd',
30.     '<cmd>lua vim.lsp.buf.declaration(<CR>)', opts)
31.   vim.api.nvim_buf_set_keymap(bufnr, 'n', 'gD',
32.     '<cmd>lua vim.lsp.buf.definition(<CR>)', opts)
33.   vim.api.nvim_buf_set_keymap(bufnr, 'n', 'K',
34.     '<cmd>lua vim.lsp.buf.hover(<CR>)', opts)
35.   vim.api.nvim_buf_set_keymap(bufnr, 'n', 'gi',
36.     '<cmd>lua vim.lsp.buf.implementation(<CR>)', opts)
37.   vim.api.nvim_buf_set_keymap(bufnr, 'n', '<C-k>',
38.     '<cmd>lua vim.lsp.buf.signature_help(<CR>)', opts)
39.   vim.api.nvim_buf_set_keymap(bufnr, 'n', '<space>wa',
40.     '<cmd>lua vim.lsp.buf.add_workspace_folder(<CR>)', opts)
41.   vim.api.nvim_buf_set_keymap(bufnr, 'n', '<space>wr',
42.     '<cmd>lua vim.lsp.buf.remove_workspace_folder(<CR>)',
43.     opts)
44.   vim.api.nvim_buf_set_keymap(bufnr, 'n', '<space>wl',
45.     '<cmd>lua print(vim.inspect(
46.       vim.lsp.buf.list_workspace_folders()))<CR>', opts)
47.   vim.api.nvim_buf_set_keymap(bufnr, 'n', '<space>D',
48.     '<cmd>lua vim.lsp.buf.type_definition(<CR>)', opts)
49.   vim.api.nvim_buf_set_keymap(bufnr, 'n', '<space>rn',
50.     '<cmd>lua vim.lsp.buf.rename(<CR>)', opts)
51.   vim.api.nvim_buf_set_keymap(bufnr, 'n', '<space>ca',
52.     '<cmd>lua vim.lsp.buf.code_action(<CR>)', opts)
53.   vim.api.nvim_buf_set_keymap(bufnr, 'n', 'gr',
54.     '<cmd>lua vim.lsp.buf.references(<CR>)', opts)
55. end
56.
57. -- PHP
58. nvim_lsp.intelephense.setup {
59.   cmd = { "intelephense", "--stdio" },
60.   filetypes = { "php" },
61. }

```

Listing 1 Continued.

```

62. --- Linter setup
63. local filetypes = {
64.   typescript = "eslint",
65.   typescriptreact = "eslint",
66.   php = { "phpcs", "psalm" },
67. }
68.
69. local linters = {
70.   phpcs = {
71.     command = "vendor/bin/phpcs",
72.     sourceName = "phpcs",
73.     debounce = 300,
74.     rootPatterns = { "composer.lock", "vendor", ".git" },
75.     args = { "--report=emacs", "-s", "-" },
76.     offsetLine = 0,
77.     offsetColumn = 0,
78.     sourceName = "phpcs",
79.     formatLines = 1,
80.     formatPattern = {
81.       "^.*(\\|\\d+):(\\|\\d+):\\|\\|s+(.*)\\|\\|s+-\\|\\|s
82.       +(.*)\\|\\|r\\|\\|n)*$",
83.       { line = 1, column = 2, message = 4, security = 3 }
84.     },
85.     securities = { error = "error", warning = "warning" },
86.     requiredFiles = { "vendor/bin/phpcs" }
87.   },
88.   psalm = {
89.     command = "./vendor/bin/psalm",
90.     sourceName = "psalm",
91.     debounce = 100,
92.     rootPatterns = { "composer.lock", "vendor", ".git" },
93.     args = { "--output-format=emacs", "--no-progress" },
94.     offsetLine = 0,
95.     offsetColumn = 0,
96.     sourceName = "psalm",
97.     formatLines = 1,
98.     formatPattern = {
99.       "^[^=]+= (\\|\\d+) = (\\|\\d+) = (.*)\\|\\|s-\\|\\|s
100.       (.*)\\|\\|r\\|\\|n)*$",
101.       { line = 1, column = 2, message = 4, security = 3 }
102.     },
103.     securities = { error = "error", warning = "warning" },
104.     requiredFiles = { "vendor/bin/psalm" }
105.   }
106. }
107.
108. nvim_lsp.diagnostics.setup {
109.   on_attach = on_attach,
110.   filetypes = vim.tbl_keys(filetypes),
111.   init_options = {
112.     filetypes = filetypes, linters = linters,
113.   },
114. }

```



Telescope

```
-- Telescope support
use 'nvim-lua/plenary.nvim'
use 'nvim-telescope/telescope.nvim'
use 'sharkdp/fd'
use {'nvim-telescope/telescope-fzf-native.nvim', \
    run = 'make' }
```

Telescope⁶ is a tool that allows you to fuzzy-find things in lists. In my configuration, I use it to fuzzy find files in my project directory using the 3rd-party tool *fd*, which provides Vim/NeoVim compatibility.

Auto-completion

```
-- nvim-cmp support
use 'hrsh7th/nvim-cmp'
use 'hrsh7th/cmp-nvim-lsp'
```

Anyone who used Vim 10-15 years ago can tell you how finicky it could get to enable good auto-completion. These days it's as easy as picking one of several plugin options. I chose *nvim-cmp* as it also integrates nicely with any LSP servers you are happening to use.

Better Diagnostics and Status Messages

```
-- shows diagnostics and other errors
use {
    "folke/trouble.nvim",
    requires = "nvim-tree/nvim-web-devicons",
    config = function ()
        require("trouble").setup {
        }
    end
}
end)
```

This is also an area where you can have lots of flexibility in tools and what you want to show. The plugin I am using works nicely with LSP to show all sorts of details at the bottom of the editor screen.

The documentation for it is excellent, even showing some keymappings to make its features easier to use.

What's Next On My List?

I am constantly evaluating my tools to make sure they continue to meet my needs. As I write this in March 2023, I'm looking at the following features:

- Commit to an Xdebug solution
- CoPilot support as I feel it's important to do enemy research
- Running unit tests from inside the editor
- More Git usage inside the editor

At the end of the day, the One True Editor is just a tool that I have decided to adopt and get it working the way I currently need it to. Take the time to explore and embrace your tools. I know that my use of Vim over the years has convinced me that keyboard-centric modal editing is an advantage. It is great that a lot of editors (and other applications) offer some kind of "Vim mode", but the best Vim mode is actually using Vim and/or NeoVim.

I hope sharing my setup inspires you to closely examine your own tools and learn them well to take advantage of what they offer. My configuration might not work for you but don't be afraid to copy it and experiment—you will be surprised what you learn about your working habits along the way.

:w!q



Chris Hartjes has been helping to build web sites of all shapes and sizes since 1998, mostly with PHP while focussing on helping developers develop the skills needed to write automated tests for their applications. He currently works as a web development consultant from his Fortress of Grumpitude deep in the snowy wilds of Canada @grumpyprogrammer@phpc.social



⁶ Telescope: <https://phpa.me/github>



Maze Rats

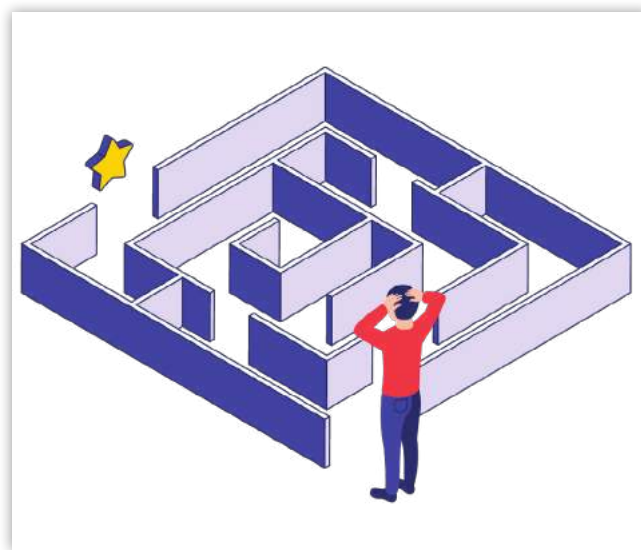
Oscar Merida

Back in the January 2021 installment for PHP Puzzles, Sherri Wheeler looked at drawing a path through a grid. In this first part, we delve into the work of maze generation.

Recap

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.



Maze Algorithms

Generating and representing mazes is a programming puzzle with a long history. One of the earliest mazes was Maze War¹:

Maze is a multiplayer first-person shooter maze game in which players traverse a flat maze and shoot opponents to score points. The maze layout is represented by a grid of spaces that are either empty or solid and form a flat plane containing walls of equal height. The game contains a default maze layout, but players can provide their own upon starting the game.

Instead of figuring it out from scratch, let's build on the work that's already been done. My initial thought on how to represent it in PHP is as a 2-dimension array holding *something* that represents which walls of the cell are open or closed.

The Wikipedia entry on Maze generation algorithms² also provides an overview on the subject. It starts with:

A maze can be generated by starting with a predetermined arrangement of cells (most commonly a rectangular grid but other arrangements are possible) with wall sites between them. This predetermined arrangement can be considered as a connected graph with the edges representing possible wall sites and the nodes representing cells.

That got me thinking that another approach could be to have a two-dimensional array where each entry represents a “vertex” in the grid, and we'd then track which of the 2 (for edge vertices) or 4 (for inner vertices) walls that connect at the vertex are open and closed. Quickly, we'd be diving into graph theory though to figure out how to represent the graph of our walls and its dual, which tracks how cells are connected to each other.

Doing that would let us create mazes of any shape and might lend itself to creating more organically shaped mazes, but that's not our task here. Many maze-generating examples online (doing research is allowed) assume a grid of squares, with each cell represented by some value.

Back to Grid Basics

Before getting lost in the maze, let's zoom in on a single cell. What do we need to track about each one? Well, each cell has four walls. Each wall can be either on or off. Hmmmm, we might be able to use that easily with a computer. A cell can have zero walls, four walls, and $2 \times 2 \times 2 \times 2 = 16$ possible combinations of walls in between. Are you seeing what I'm seeing?

¹ Maze War: <https://phpa.me/wikipedia-maze>

² Maze generation algorithms: <https://phpa.me/wikip-maze-algorithms>



We can use four bits to represent if our walls are on or off. If a cell had no walls, the value of each wall is zero, and then we'd represent it with four binary bits as:

N S E W Decimal Hexadecimal
0 0 0 0 0000 = 0 0

If we have a cell with four walls, our bits are all on.

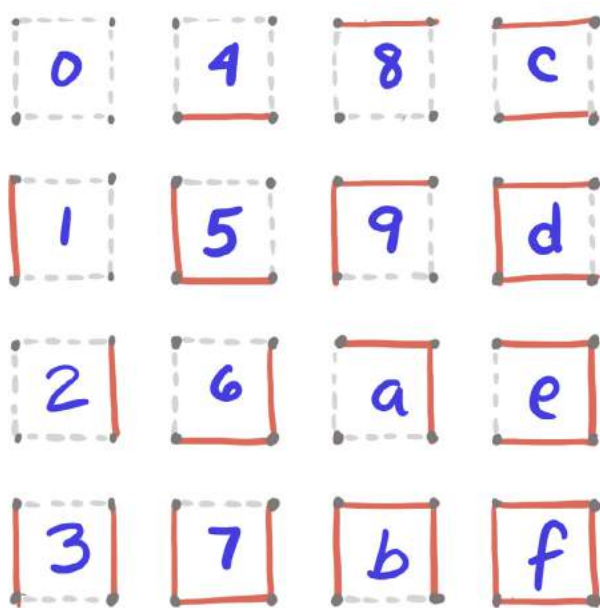
N S E W Decimal Hexadecimal
1 1 1 1 1111 = 15 f

Aha! We can count from 0 to 16 to enumerate all possible states

| N S E W | Dec | Hex | N S E W | Dec | Hex |
|---------|-----|-----|---------|-----|-----|
| 0 0 0 0 | 0 | 0 | 1 0 0 0 | 8 | 8 |
| 0 0 0 1 | 1 | 1 | 1 0 0 1 | 9 | 9 |
| 0 0 1 0 | 2 | 2 | 1 0 1 0 | 10 | a |
| 0 0 1 1 | 3 | 3 | 1 0 1 1 | 11 | b |
| 0 1 0 0 | 4 | 4 | 1 1 0 0 | 12 | c |
| 0 1 0 1 | 5 | 5 | 1 0 1 1 | 13 | d |
| 0 1 1 0 | 6 | 6 | 1 1 0 0 | 14 | e |
| 0 1 1 1 | 7 | 7 | 1 0 0 0 | 15 | f |

Figure 1 may help you visualize what each of the hexadecimal values above represents. For one, cells with only one wall are given a value that is a power of 2, because binary. Cells with more than one wall are the sum of these. For example, “d” in figure (decimal 11) is the sum of 8+2+1. In any future application, if we need to check if two cells have walls in common or to combine the layout of two cells, we can use | (OR).

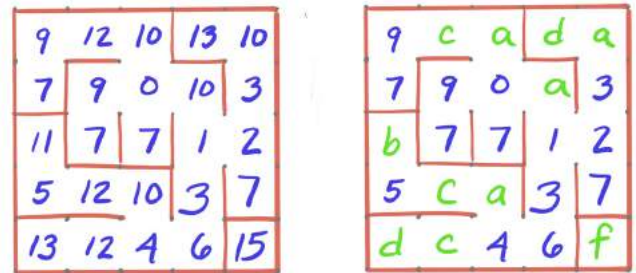
Figure 1.



Implementing It Manually

Figure 2 shows a 5x5 grid with a maze I sketched to test out how the representation might work. To help in translating each cell to our bit values, I used the key above.

Figure 2.



I walked the grid manually, and depending on the state of the walls for a cell, wrote the number that corresponded. Then, I wrote the decimal numbers in hexadecimal.

Converting It to Code

As a PHP array, our maze would be shown as:

```
$maze = [
    [0x9, 0xC, 0xA, 0xD, 0xA],
    [0x7, 0x9, 0x0, 0xA, 0x3],
    [0xB, 0x7, 0x7, 0x1, 0x2],
    [0x5, 0xC, 0xA, 0x3, 0x7],
    [0xD, 0xC, 0x4, 0x6, 0xF]
];
```

Visualizing the Maze

We can generate a rough visualization if we draw the bottom and right walls of each cell, along with the borders. We'll use some constants and dip our toes in bitwise operations to help us along.

To generate our maze, we use bitwise AND—a single & to see if the only bits in common between our cell and the desired wall state, EAST, for example, are true. I started rendering the map only using pipes |, plus signs +, and minus signs -. Once the maze looked like my sketch, I looked through some extended ASCII box drawing entities³ to tie it all together. (See Listing 1 and Figure 3)

³ box drawing entities: <https://phpa.me/w3-entitynames>



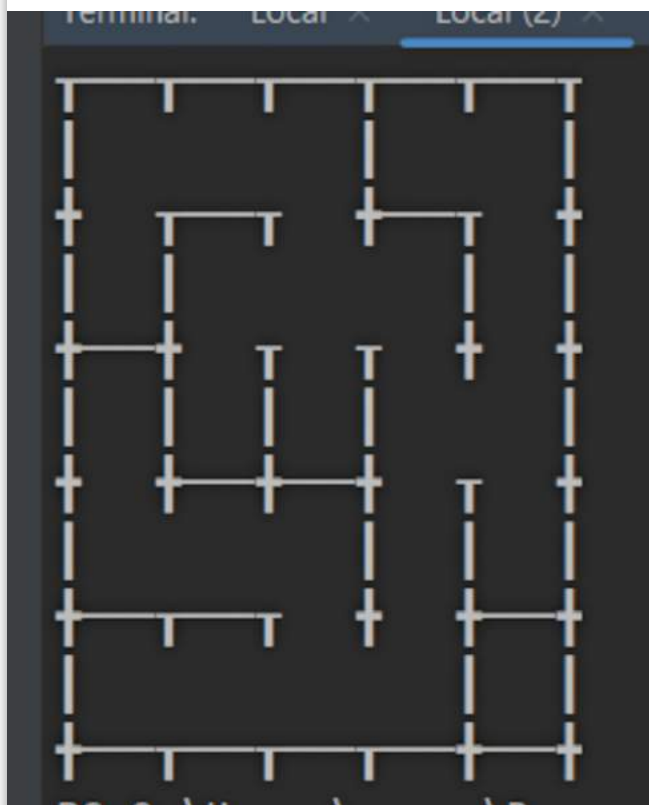
Listing 1.

```

1. <?php
2.
3. $maze = [
4.     [0x9, 0xC, 0xA, 0xD, 0xA],
5.     [0x7, 0x9, 0x0, 0xA, 0x3],
6.     [0xB, 0x7, 0x7, 0x1, 0x2],
7.     [0x5, 0xC, 0xA, 0x3, 0x7],
8.     [0xD, 0xC, 0x4, 0x6, 0xF]
9. ];
10.
11. // draw the top line
12. const EAST = 0x2;
13. const SOUTH = 0x4;
14.
15. $repeat = count($maze[1]);
16. echo '□' . str_repeat('—□', $repeat) . PHP_EOL;
17. foreach ($maze as $row) {
18.     $east = '□';
19.     $south = '□';
20.     foreach ($row as $cell) {
21.         // draw the east walls
22.         $east .= (($cell & EAST) == EAST ? '—' : ' ');
23.         // draw the southern walls
24.         $south .= (($cell & SOUTH) == SOUTH ? '—' : ' ');
25.         . (($cell & EAST) == EAST ? '□' : '□');
26.     }
27.     echo $east . PHP_EOL . $south . PHP_EOL;
28. }

```

Figure 3.



Make a Maze

On to part two!

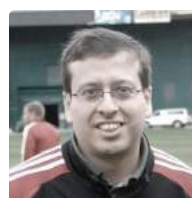
Archlin studies your notes on how you propose to record mazes in the PHP-o-nomicon. “This might do after all, apprentice,” he mutters as one of his great ears twitches excitedly. “This might keep the treklins busy and out of trouble. Come, we have little time left for the next step.”

He strides out of his study in the great tower of Sharadun towards the Great Courtyard. He points his trunk to the ground below. “I will cast a spell to build a maze below, and each hour, the maze will change. You will design the walls of each maze on the fly.”

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. There should be a single entrance in the north edge, a single exit in the south edge, and no dead ends in the maze.

Some Guidelines And Tips

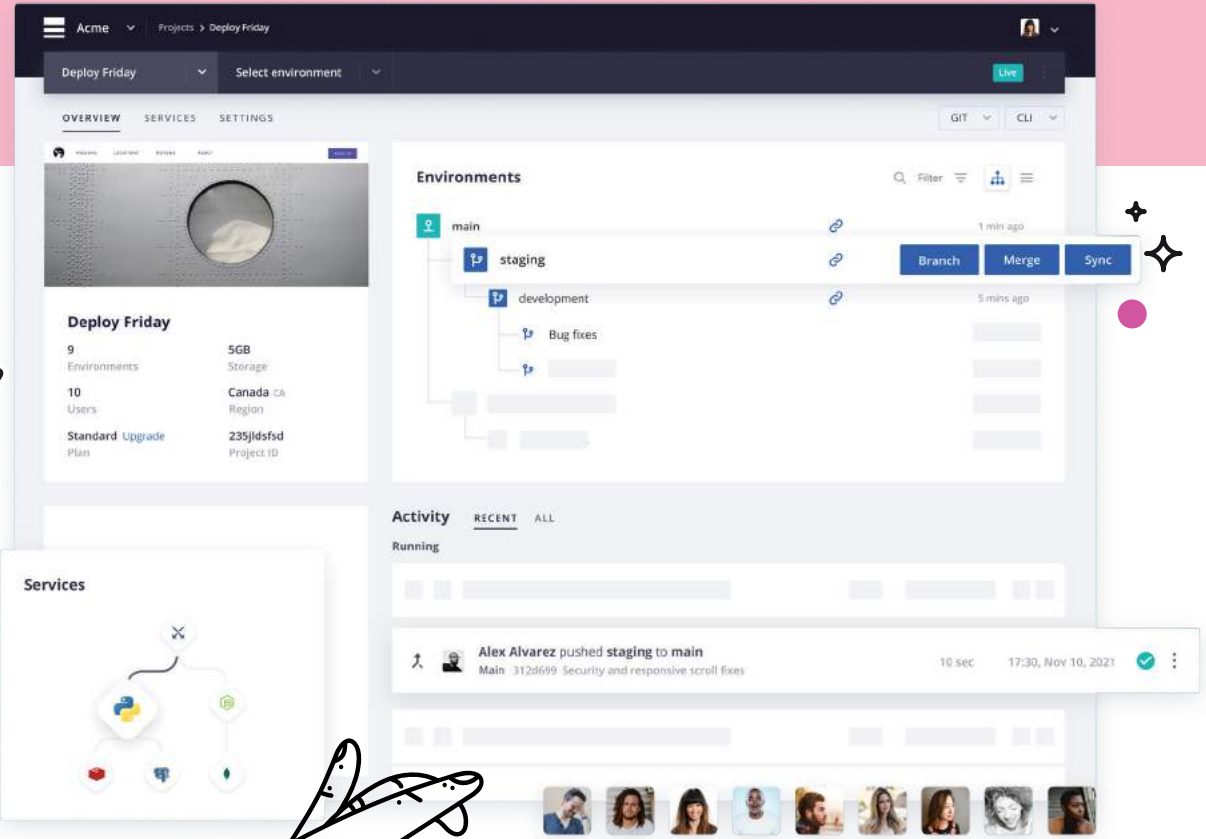
- The puzzles can be solved with pure PHP. No frameworks or libraries are required.
- Each solution is encapsulated in a function or a class, and I’ll give sample output to test your solution against.
- You’re encouraged to make your first attempt at solving the problem without using the web for clues.
- 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 PsySH⁴ 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](#)

Deploy Friday. Now you can.

The PaaS to build, iterate, and deploy applications your way



The screenshot displays the platform.sh dashboard for a project named "Deploy Friday". The interface includes a top navigation bar with "Acme" and "Projects > Deploy Friday". Below this, there's a "Select environment" dropdown and a "Live" button. The main content area is divided into several sections:

- Overview:** Shows a summary of the project, including "9 Environments", "10 Users", and "Standard Upgrade Plan". It also lists resources: "5GB Storage", "Canada CA Region", and "235jldsfds Project ID".
- Environments:** A tree view showing the deployment pipeline. It includes "main", "staging", and "development" branches. The "staging" branch is highlighted, and there are buttons for "Branch", "Merge", and "Sync".
- Activity:** A section showing recent activity, including a message: "Alex Alvarez pushed staging to main" with details like "Main 312d699 Security and responsive scroll fixes" and a timestamp of "17:30, Nov 10, 2021".
- Services:** A diagram showing the service architecture, including a database and various application components.

Decorative elements include a vertical stack of cloud provider logos (AWS, Google Cloud, Azure, Oracle, Vercel) on the left, a hand cursor pointing at the bottom left, and a series of user avatars at the bottom right.

Impedance Mismatch

Edward Barnard

When I drew out our PHP application on the whiteboard, it looked like a Big Ball of Mud. We changed the drawing slightly, and something interesting appeared.

Build an API

Within my work project, we've been considering how best to build an API. That created an interesting insight into Domain-Driven Design. I discovered there's an intersection between modern API development, microservices, automated containerized deployments, and cloud native application design. That intersection is Domain-Driven Design!

API design (in part) focuses on providing business capability. For example, when designing a stateless REST API, we can't ensure transactional consistency between two separate API calls. If two operations must succeed or fail together, that's the very definition of transactional consistency. Both operations must therefore be part of the same API call. They can't be separate.

This makes complete sense if we think of those two operations as part of the same DDD Aggregate. James Higginbotham's *Principles of Web API Design: Delivering Value with APIs and Microservices* (2021), for example, teaches collaborative design, Aggregates, Event Storming, and Bounded Contexts.

It's the same way with microservice design. When you understand the reasons and boundaries with Domain-Driven Design, those are the boundaries for your microservices.

From a team perspective, Microservices exist because of the need for *not* collaborating. Microservices represent the Single-responsibility principle¹ applied to teams. Different teams working in the same codebase inevitably create (code) conflicts during day-to-day development. Assign responsibility for each microservice to one specific team, and (in theory) we'll avoid having teams stepping on each other.

A microservice, by definition, is separately and independently deployable. Again, this is the theory; it does not always work that way in practice! This practice removes *temporal* coupling between teams.

What if you only have one team? Don't do microservices. There's no team-related justification for the added complexity of individual microservices. That is, there's no need to reduce coupling between development teams.

Modular Monolith

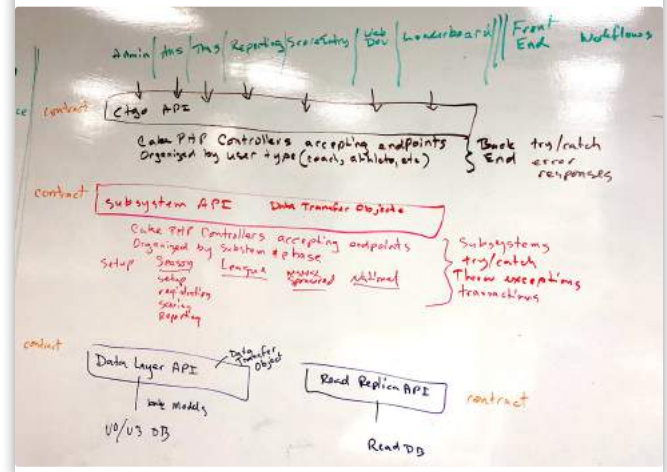
API design and microservice design intersect at an interesting location: the modular monolith. Shopify Engineering developer Kirsten Westeinde provides an excellent article

showing how Shopify achieved a modular monolith, "Deconstructing the Monolith: Designing Software that Maximizes Developer Productivity"².

The general approach is this: go ahead and draw the boundaries that would exist if you created well-architected microservices. What's interesting to me is that those boundaries are, in fact, Bounded Contexts in the DDD sense.

When I drew this idea on our whiteboard, I discovered something interesting. I had drawn out a Big Ball of Mud³! See Figure 1.

Figure 1.



There was a structural problem keeping us from successfully implementing Domain-Driven Design.

Our situation is this. Over the course of a half-year, we have three phases:

- Local competition (the regular sports season)
- Regional competition (state championships)
- National competition

Within each of those three phases, we have a similar sequence from a software standpoint:

- System-Administrative setup
- Registration

² "Deconstructing the Monolith: Designing Software that Maximizes Developer Productivity": <https://phpa.me/shopify-Engineering>

³ Big Ball of Mud: <http://www.laputan.org/mud/>

¹ Single-responsibility principle: <https://phpa.me/wikipedia-Single>



- Competition setup (based on registration numbers and groupings)
- The competition
- Post-competition reporting

Drawn on a whiteboard, the above all looks simple and perfectly clear.

Now comes our source of mud.

We have three distinctly different sets of users with completely different sets of requirements:

- The athletes competing
- Team coaches and other team staff
- System administrators

We created our PHP software, over the course of many years, as three modules:

- Athlete Management System (for the athletes)
- Team Management System (for the coaches)
- Admin module (for system admins)

We didn't slice it both ways (i.e., we kept athlete code with athlete code, team code with team code, etc.), and that's a key reason we now have a Big Ball of Mud.

Where, for example, is our business logic supporting Registration? It's scattered across the three modules (Athlete, Team, Admin). Until last year, we used the same code for Season, State, and National competitions. At the end of a season, we literally deleted the Season code and replaced it with the Tournament code. Then, at the beginning of the next season, we hoped we found the right copy of the code to put back in. "Big ball of mud" was, for us, a euphemism.

Now, however, we have a new business requirement. That requirement is leading us to consider API design. Researching API design led me to discover what the experts have to say. We have two layers! In fact, we might have three layers.

Unfortunately, I erased the whiteboard but did not take a photo of the next drawing. I created three top-level APIs. I laid out one API each for Athlete Management, Team Management, and System Administration. Each can be stateless and has distinctly different authentication and authorization requirements. Each of those APIs can be designed for its specific consumer. For example, the team coach, through the Team Management API, might invite an athlete to compete. Through the Athlete Management API, that athlete completes their registration.

I then laid out a second set of APIs. I drew out three business phases:

- Local competition (the regular sports season)
- Regional competition (state championships)
- National competition

Next, I drew out the five steps listed above for each business phase as "subsystems" for lack of a better description. We now had a matrix of independent components.

For example, a team's Head Coach might need a list of all coaches on the team with their phone numbers. However, our

help desk (in the System Administration module) will also need that same list. Notice there's a difference in authorization:

- The Head Coach can only list that coach's team.
- The help desk needs to be able to list staff contacts for any team.

In other words, the *response* has the exact same content. Still, the *request* comes from two different modules, which will have already authorized that request according to different criteria.

On the whiteboard, then, I drew out two layers of REST APIs. Our Big Ball of Mud suddenly looked like a well-structured modular monolith!

Making actual internal HTTP requests for the sake of structure would be silly. That's unnecessary traffic which, as a bonus, adds a new point of failure.

However, there's an advantage to formally describing both layers of APIs as an OpenAPI Specification (OAS). The request and response structures (called Models) can be generated via the Swagger-Codegen⁴ Open Source project. I used the Node.js version from OpenAPI Generator CLI⁵.

I ignore the generated API calls (except as examples of usage) and instead copy the generated models into my project. They include any validation criteria that I specified in my OpenAPI Specification. It works out rather nicely!

Data Transfer Object

All this research was useful. I realized we could insert our API boundaries within the current Big Ball of Mud without any actual API calls. That is, without any HTTP requests. We could make internal method-to-method (or function) calls, passing the generated API request and response objects.

That would solve a second problem, essentially protecting us from changes to our third layer: the database schema. Our current codebase uses variables and properties throughout the code, which are based on table column names. When a table column, table relationship, or table name changes, we need to find and adjust code in many places.

Now, with the API-based Model classes, our database queries can use the API-based classes instead of classes directly tied to the database schema.

However, let me hasten to point out that I've just described a nasty anti-pattern. Martin Fowler, who documented the Data Transfer Object pattern in the first place, explains the problem with Local DTO⁶:

DTOs are called Data Transfer Objects because their whole purpose is to shift data in expensive remote calls. They are part of implementing a coarse grained interface which a remote interface needs for performance. Not just

⁴ Swagger-Codegen: <https://phpa.me/github-swagger>

⁵ OpenAPI Generator CLI: <https://phpa.me/github-OpenAPI>

⁶ Local DTO: <https://martinfowler.com/bliki/LocalDTO.html>



do you not need them in a local context, they are actually harmful both because a coarse-grained API is more difficult to use and because you have to do all the work moving data from your domain or data source layer into the DTOs.

My diagram was safe on the whiteboard, but as code, maybe not! If the *only* use of request and response objects is to construct the object in a lower layer which *will* cross an HTTP API boundary as a response object, we're on the right path. But to marshal, serialize, and deserialize, merely for an internal boundary, is likely not the best approach.

Create Seams

In our case, there's certainly value in creating "seams" in the code, allowing us to begin creating the two-layer structure I described above. We can think and reason about our codebase on the whiteboard with two layers of RESTful APIs. Inserting boundaries, such as typical PHP method-to-method calls, allows us to begin "taming" our codebase one piece at a time.

Michael C. Feathers, in *Working Effectively with Legacy Code* (2005), observes (Chapter 4, "The Seam Model"):

One of the things that nearly everyone notices when they try to write tests for existing code is just how poorly suited code is to testing. It isn't just particular programs or languages. In general, programming languages just don't seem to support testing very well. It seems that the only ways to end up with an easily testable program are to write tests as you develop it or spend a bit of time trying to "design for testability". There is a lot of hope for the former approach, but the latter hasn't been very successful...

When you start to try to pull out individual classes for unit testing, often you have to break a lot of dependencies. Interestingly enough, you often have a lot of work to do, regardless of how "good" the design is.

Feathers defines "seam":

A seam is a place where you can alter behavior in your program without editing in place.

At this point, I'm wondering whether I have a good idea, or a bad idea, using the API-design paradigm to better organize our monolith. Ian Cartwright, Rob Horn, and James Lewis published "Patterns of Legacy Displacement: Effective modernization of legacy software systems"⁷ online July 2022:

When faced with the need to replace existing software systems, organizations often fall into a cycle of half-completed technology replacements. Our experiences have taught us a series of patterns that allow us to break this cycle, relying on: a deliberate recognition of the desired outcomes of displacing the legacy software, breaking this displacement in parts, incrementally delivering these parts, and changing the culture of the organization to recognize that change is the unvarying reality.

One of their patterns is Transitional Architecture⁸: "Software elements installed to ease the displacement of a legacy system that we intend to remove when the displacement is complete."

Consider the renovation of a building. An architect has provided you with renderings of the finished product and builders are standing by to start. But the first step is to put scaffolding up on the building site.

Hiring the scaffolding itself and paying a team to construct it is an unavoidable investment. It is needed to enable critical work to be done, and buys risk mitigation during the renovation increasing the safety of the workers. It may even unlock new options... Once the work is completed, another team will arrive and dismantle the scaffold, and you are pleased to see it go.

In a legacy displacement context, this scaffolding consists of software components that ease, or enable, building the current evolutionary step towards the target architecture... Replacing a large legacy monolith in one go is risky, and we can improve the safety to the business by displacing it in several steps.

Aha! We can, for example, introduce one seam (or other transitional architecture) at a time without changing observable behavior. Martin Fowler calls this "refactoring: improving the design of existing code". Seams can allow us to create automated tests protecting that part of the codebase. Seams will enable us to replace (rewrite) the portion that's on one side of the seam or the other—without affecting anything else. Seams allow us to rewrite one small piece at a time while continuously deploying to production rather than waiting for a Big Bang rewrite to complete.

Impedance Mismatch

By impedance mismatch⁹, I mean the general idea of needing to adapt one shape to meet a different shape. See Figure 2 (see next page), courtesy Library of Congress, "Riveter at work on Consolidated bomber, Consolidated Aircraft Corp., Fort Worth, Texas".

⁸ Transitional Architecture:

<https://phpa.me/MartinFowlertransitional-architecture.html>

⁹ impedance mismatch: <https://phpa.me/wikipedia-impedance>

⁷ "Patterns of Legacy Displacement: Effective modernization of legacy software systems": <https://phpa.me/MartinFowler>

Figure 2.



In our application, we essentially have three user-facing applications: athletes, team coaches, and system administrators. Obviously, it's a business need to serve each of those types of users well. However, our internal business rules correspond to a flow of dates on a calendar. All user types interact on a given calendar date. Do we arrange the implementation (e.g., registration) by actor or activity? We have a mismatch.

We can do both by creating an internal API—even if just on a whiteboard. Place a boundary between the athlete-facing module and its registration-activity code. Place another boundary between the team-coach-facing module and its portion of the registration-activity code. Insert a similar boundary within the System Administration module.

Now, in theory, we can move all the registration-activity code out of their respective modules and place them in their new home, the Registration Subsystem. We're now one step closer to a well-structured modular monolith.

Of Course, with a legacy codebase, it's never that easy. Kent Beck advises¹⁰:

For each desired change, make the change easy (warning: this may be hard), then make the easy change.

Summary

Drawing diagrams on a whiteboard proved key to better understanding our Big Ball of Mud and what to do about it. We then considered API design—only on the whiteboard, not in code—which led us to the idea of creating a modular monolith.

With that API design on the whiteboard, we created internal boundaries. Aha! As Michael Feathers explained 18

years ago, creating seams allows us to deal with legacy code effectively. By breaking dependencies, seams allow us to build automated tests around those parts of our system.

Our whiteboard-imposed seams fit the pattern of Transitional Architecture. They are intended to be temporary. They facilitate reaching our goals (whatever our specific goals might be) of bringing our legacy application forward into the modern era.

The next time you're faced with an “unseamly” legacy application, consider how you might approach API design. How might you restructure it into microservices—but only on the whiteboard? Where would you draw the boundaries around separate areas of responsibility? You now have the possibility of creating seams, bringing your application under control one tiny step at a time.



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)

Elevate your PHP skills

Subscribe to the php[architect]
YouTube Channel

https://youtube.com/phparch

¹⁰ Kent Beck advises: <https://phpa.me/kentbeck-0032>



PSR-18: HTTP Client

Frank Wallen

This month we will look at PSR-18, the HTTP Client. Use of APIs has grown significantly over the years, from providing sports stats and news headlines to driving the backend of websites, processing payments, and conversations with AI chatbots. Separated services like these give us greater control of content, and resources, and improved performance through asynchronous requests. This article will not be covering API design but rather the design around the consumer of APIs. However, I recommend some talks presented at Tek that you will want to attend to learn more about APIs: Matthew Turland's *"OpenAPI: More Than Documentation"* and Tim Bond's *"Attackers want your data and they're getting it from your API"* on Day 2 (Wednesday), and Ian Littman with *"API Design Patterns for the REST of Us"* on Day 3 (Thursday).

PSR-18¹ offers a design for a common interface for sending PSR-7² messages and PSR-7 responses. However, it does not define any support for asynchronous requests, as PHP does not run asynchronous threads nor offer Promises, and is considered out of scope for this PSR. This PSR intends to present an interface in support of the Liskov Substitution Principle so that a library can swap out an HTTP Client implementation for another. These interfaces can be found on Packagist³ and Github⁴.

Let's review the design:

Client Interface

```
namespace Psr\Http\Client;

use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

interface ClientInterface
{
    public function sendRequest(
        RequestInterface $request
    ): ResponseInterface;
}
```

The interface is simple. There is no requirement for configuration, only that the Client observes that handling requests and responses follow PSR-7 message interfaces. In the specific case of HTTP 1.xx status code responses, the Client MUST reassemble what is returned to return a valid HTTP 2.xx response. The Client may also need to perform additional actions on sending and receiving requests, such as compressing and decompressing the message body. When altering the HTTP request or response, it is the responsibility of the Client to ensure the object remains consistent internally. An example from the PSR states: "...if a Client chooses

to decompress the message body then it MUST also remove the Content-Encoding header and adjust the Content-Length header."

How the sendRequest handles the request is up to the designs of the implementor. It can be as simple as the following example:

```
use Guzzle\Http\Client;
use Psr\Http\Client\ClientInterface;
use Psr\Http\Message\RequestInterface;
use Psr\Http\Message\ResponseInterface;

class MyClient implements ClientInterface
{
    public function sendRequest(
        RequestInterface $request
    ): ResponseInterface {
        $client = new Client();
        return $client->send($request);
    }
}

use Clients\MyClient;
use Psr\Http\Client\ClientExceptionInterface;

$client = new MyClient();
$request = $httpRequestFactory->createRequest(
    'GET',
    'https://tek.phparch.com/'
);

try {
    $response = $client->sendRequest($request);
} catch (ClientExceptionInterface $e) {
    throw new Exception($e);
}

// ... get data from response ...
```

¹ PSR-18: <https://www.php-fig.org/psr/psr-18>

² PSR-7: <https://www.php-fig.org/psr/psr-7/>

³ Packagist: <https://packagist.org/packages/psr/http-client>

⁴ Github: <https://github.com/php-fig/http-client>



Our script here creates a new instance of `MyClient` and gets a request object created by a request factory (we discussed this for PSR-17⁵). Then in our `try/catch`, we send the request to `MyClient::sendRequest`, where a Guzzle client is created and the request sent through. Exceptions specific to PSR-18 are caught, and a regular exception is thrown. In the real world, another catch should be added for any other non-`ClientException` errors. If, for example, `MyClient` was unable to instantiate a Guzzle instance, we would get a `Not Found` exception rather than `ClientException`.

Following are the Exception interfaces that should be implemented to properly identify errors in the Client. If the Client receives a well-formed HTTP 400 or HTTP 500 response, this should not be treated as an exception and **MUST** be returned to the calling library. Only if the request could not be sent or the response could not be parsed into a PSR-7 response object should a `ClientExceptionInterface` exception be thrown.

Client Exception Interface

```
namespace Psr\Http\Client;

interface ClientExceptionInterface extends \Throwable
{
}
```

If the request object passed to the Client is not properly formed or missing critical information like the method or host, then a `RequestExceptionInterface` exception should be thrown.

Request Exception Interface

```
namespace Psr\Http\Client;

use Psr\Http\Message\RequestInterface;

interface RequestExceptionInterface
    extends ClientExceptionInterface
{
    public function getRequest(): RequestInterface;
}
```

If the request cannot be sent due to a network failure or timeout, then a `NetworkExceptionInterface` exception should be thrown. Note that a request exception is not the same as a network exception. If it is not a network error, then a request exception can be thrown. A Client **MAY** also throw exceptions such as a `TimeoutException` as long as it implements the correct interface.

Network Exception Interface

```
namespace Psr\Http\Client;

use Psr\Http\Message\RequestInterface;

interface NetworkExceptionInterface
    extends ClientExceptionInterface
{
    public function getRequest(): RequestInterface;
}
```

Conclusion

PSR-18 provides a standardized interface for HTTP clients, allowing for easy swapping of different client implementations. There are many ways that an HTTP Client can be implemented, which one can see in the following list⁶ at Packagist. Using a battle-tested library is always recommended (or at least studied), and it's a simple matter to wrap one's client code around one of these libraries. Next month we'll be looking at PSR-20: Clock⁷ (for a consistent interface for getting the time and an easy method for mocking time in tests). Speaking of time, enjoy *your* time at **php[Tek] 2023**, and I will see you there!

Related Reading

- *PSR Pickup: PSR-13: Link Definition Interfaces* by Frank Wallen, December 2022.
<https://phpa.me/psr-dec-2022>
- *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>
- *PSR Pickup: PSR-17: HTTP Factories* by Frank Wallen, March 2023.
<https://phpa.me/2023-03-psr>



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 *Rancherito* where he can live with his family, have a dog, and, of course, a cat. [@frank_wallen](https://twitter.com/frank_wallen)

⁵ PSR-17: <https://www.php-fig.org/psr/psr-17>

⁶ list: <https://phpa.me/PackagistImplementation>

⁷ PSR-20: Clock: <https://www.php-fig.org/psr/psr-20>



The Subtle Art of Optimal DaaS

Matt Lantz

All too often, when any Laravel developer is requested to start a project, they get zoned in on the selection of things like Livewire vs. Inertia or which Spatie packages to use. Generally, their focus is on the code, not the infrastructure or the DevOps tooling. Regardless of their stack, most developers have a cloud provider they are most comfortable with: Digital Ocean or AWS, or one of the many others. Second, particularly in the Laravel community, the next question is, do I use Forge or Vapor for this project? What we're examining is the subtle art of DaaS selection and optimization.

Most developers, throughout their careers, dabble in brief moments of DevOps work, and more often than not, most projects utilize some DevOps-as-a-Service solution. This is most notable in the cases of smaller organizations with greenfield projects that do not have the sort of predefined solutions and compliance regulations that large-scale organizations are bound to. Most developers are working in smaller teams (or small teams within bigger teams) and need to handle their own DevOps solutions.

So, much like conference organizers who have to review the big picture of the venue, travel times, and food options—most Laravel developers will, at some point, need to check the possibilities different DaaS solutions provide and determine which is the best fit for their project. Furthermore, as their project grows and changes, developers should continue intermittently reviewing if their DaaS solution is optimized for their project and team needs.

Forge: The Original

Laravel Forge was released as an elegant server management SaaS tool back in the days when numerous developers were still trapped using CPanel for many clients. It provided server provisioning essentials focused entirely on Laravel. Fast deployments with GIT (though these can create technical downtime) were also available immediately. Since its initial release, the Laravel team has enabled PHP upgrades, server modifications, system monitoring, and

many more features. Forge lets developers provision servers quickly on any cloud provider and handles spinning up servers with specific focuses like Databases, Caches, or Search Engines. Given the classic provisioning nature of a Virtual Server, developers can also directly access the machines and dig around where needed. As a DaaS solution Forge provides a simple way of handling application servers and their maintenance.

Vapor: The Sequel

In many ways, Laravel Vapor has been a sequel to Forge. Forge handled classic server provisioning and deployments. Vapor, on the other hand, is a non-agnostic serverless deployment system. Built on top of AWS exclusively, developers can deploy their code through its systems. Much like Forge, it provides an entire elegant management layer on top of AWS for things like Redis Caches, Databases, etc.; the primary difference is that each of these is configured within AWS in your VPC.

Given that Vapor is deploying code into AWS Lambda functions, you do end up losing the classic controls of the server, but, what you gain is the lack of server maintenance required. It can also scale the above resources and configure assets to be delivered through CloudFront CDN. Overall, it takes more configuration to get Vapor set up within your application, but the payoff of zero downtime deployments and zero server maintenance quickly makes up for it.

Kubernetes: The Enterprise Standard

The above options are often the first and most considered options in the Laravel community, and for a good reason, they were made by the Laravel team and continue to get new features added to them. However, Kubernetes does offer a comparable experience to Vapor while enabling teams to be cloud provider agnostic. Kubernetes is the enterprise standard, and most cloud providers offer a low-maintenance instance of it on their platforms. Kubernetes internal commands can handle deploying your containers, unlike Forge and Vapor, which don't require containers (though Vapor works with them); everything is containerized when it comes to Kubernetes. Since your code gets packaged into a container, and the build image is in a container registry, you can promptly roll back deployments to the previous state. Using additional toolings like ArgoCD or other GitOps apps means you can then migrate your deployment controls to git repositories which provide elegant solutions to release management cycles etc. With Kubernetes and tools like ArgoCD, we also gain the capability of autoscaling and auto-regenerating pods when systems crash or get overwhelmed. It requires more DevOps knowledge than Vapor, but developers who have learned how to upgrade and modify provisioned servers (like Forge ones) can quickly learn Kubernetes.



What about Scaling?

It's worth noting that optimizing your DaaS solution, as we've seen, only sometimes correlates to your application requiring scaling. Scaling a Laravel application is a different challenge we will not deeply inspect here. Though your DaaS solution impacts how you select and configure your scaling, your DaaS tooling is both an enabler and a restrictor. Scaling a Laravel application requires ensuring that your code can run on not just one server but any number of servers, or in the case of Vapor, any number of functions. Doing this requires ensuring that all resources: database, queue, sessions, and cache are separate from the server, which Laravel has made quite simple. Then pending on the above DaaS solution, there are ways to configure your scaling options.

Forge:

When considering scaling an application deployed with Forge, we can explore options like Laravel Octane and horizontal and vertical scaling. In the most straightforward context, we can quickly scale up a server's resources to handle more demanding traffic. Suppose this case gets too expensive or unstable, having all traffic making demands on a single server. In that case, we can quickly deploy the application to a second server and set up a load balancer within Forge in minutes. Regarding using Octane, there are further considerations that likely involve coding changes and may require provisioning new servers anyway to configure nginx properly, but Forge provides all these options.


Vapor and Kubernetes:

Scaling is pretty automated with these options. Though there may be some configuration adjustments regarding how many resources you're running per environment, most of the scaling configuration in these choices relates to your cost comfort. Each of these can also work with options like Laravel Octane, but as mentioned, it will likely require some code changes. Though Vapor has extensive documentation on implementing Octane with Vapor, there is very little documentation in the Kubernetes community on this topic.

DaaS software has grown significantly in the last several years, enabling developers to release code environments quickly and effectively. If you're part of a small, focused team, harnessing Laravel Forge or Vapor may be the perfect fit; however, if you've got sufficient resources or are just curious, Kubernetes can provide a robust, scalable solution. As projects progress, they may outgrow their initial DaaS solution or find it no longer suits the team handling the work. Whether you jump onto the serverless train, enjoy upgrading servers, or want to replicate enterprise configurations, there is a DaaS solution out there that can fit your project, skillsets, and above all, your budget.



Marian Pop is a PHP / Laravel Developer based in Transylvania. He writes and maintains [LaravelMagazine.com](https://laravelmagazine.com) and hosts "The Laravel Magazine Podcast". [@MattyLantz](https://twitter.com/MattyLantz)



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>



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/>



Stop Waiting

Beth Tucker Long

What's on your 'Someday' list? What do you keep putting off, waiting for the right time? Dust off that list. Someday has arrived....Stop Waiting!

In the Before Times*, I went to a lot of conferences and heard about many helpful tools. Every talk I went to, I thought to myself, "Oh! I should check that out. That sounds super helpful!" They went on my list of "Things to Do" one after another until that list was so overwhelming and old that I had no idea where to start on it. I was really busy working on projects, so of course, I would go through the list when I had more time...but that "more time" never materialized.

(The Before Times are what I affectionately call those blissfully unaware days before the start of the pandemic, which are simultaneously not that long ago and so long ago that I barely remember them.)*

Besides tools and strategies, one major item that has been sitting idle on my "Things to Do" list for at least a decade is hiring an intern. Of course, I had lots of great reasons why it was never the right time to do this:

- I'm swamped right now, so I need to get caught up on my work before I can dedicate time to an intern.
- I don't have anything documented to help onboard them, so I need to put together onboarding documents first.
- I'm unsure if I will have enough work at the right level for a junior employee, so I need to find more appropriate client work first.
- Our budget doesn't have a lot of wiggle room, so I need to save up more money for this first so to have enough money to provide a good wage and training.
- My company is so small that an intern wouldn't get much out of working with us. I need to get some more programmers on staff first.

A few months ago, though, I was talking to a friend who was finishing up their degree at a local college. They were struggling to find an internship because many companies had discontinued their internship programs during the pandemic, and there were way too many students seeking internships compared to the available programs. They asked me if I would consider hiring them as an intern. They were familiar with my

company and were excited about working on a smaller team. I didn't have time to talk myself out of it. I now have an intern, and I can't believe I waited this long to hire one.

I was so worried about the extra time and resources I would need to devote to an intern in order to give them the best experience possible that I never stopped to consider how beneficial it would be to me. With my intern, I am starting them out slowly so they can get their feet under them which means they are handling all of the smaller tasks that normally clog up my time so I can spend more time digging into the more advanced tasks. I have also been tossing items from my "Things to Do" list their way and asking them to research and learn to use the tools I had marked as possibly being helpful additions to our workflow. It's a great opportunity for them to learn more about what is out there, and it means the tool can be set up and ready to use for me, clearing that initial effort hurdle that prevented me from checking them out for so long.

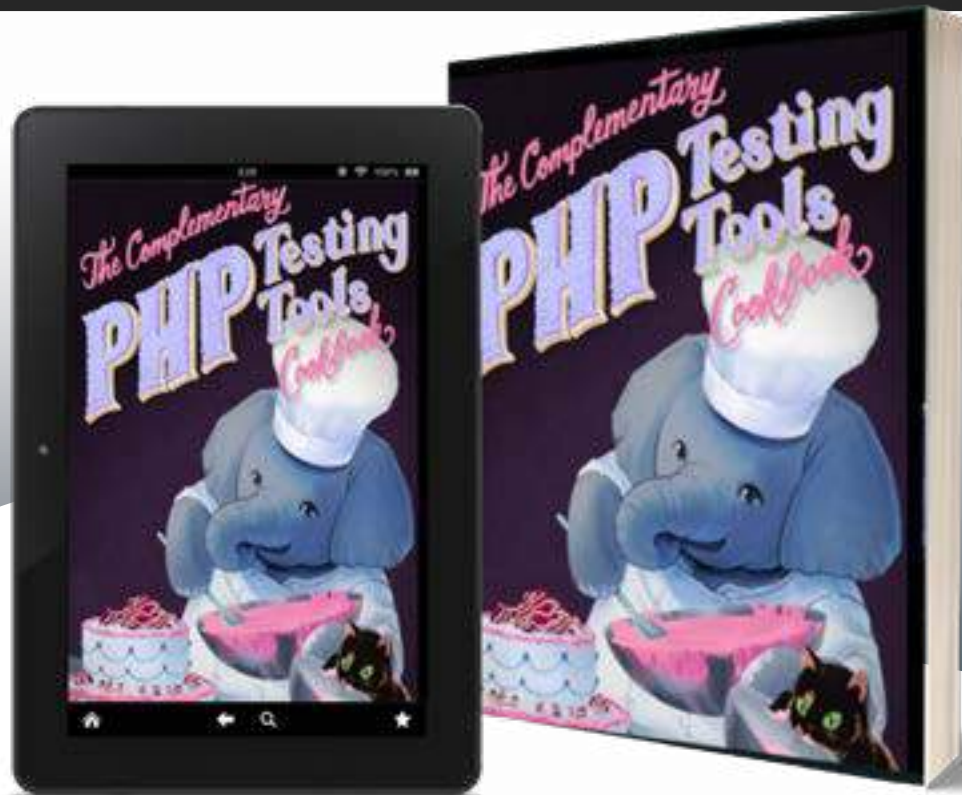
Suddenly, our testing is improving. Our ticketing system is getting more organized and fully-featured. Teaching them to follow our procedures is highlighting where we have massive inefficiencies that we have all become used to and no longer notice.

Instead of this being a sacrifice we make to help improve someone else's career, this internship is drastically improving our company and my stress level! I wish I had done this years ago.

What's on your "Someday" list that you know you want to do but haven't gotten done? What is your first step? Don't give yourself time to come up with excuses. Make it happen.



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



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



PhpStorm

Enjoy
productive
PHP

jetbrains.com/phpstorm