



php[architect]

The Magazine For PHP Professionals

Knowledge Crunching

Headless Possibilities for PHP

What Can the NSA Teach Us About Debugging?

ALSO INSIDE

The Workshop:

Upgrading with Reckless Abandon:
Part 2

Artisan Way:

Events, Listeners, Jobs, and Queues
Oh My!

PSR Pickup:

PSR-15: HTTP Server
Request Handlers

Education Station:

The Why and How of Building
Microservices

Security Corner:

Infosec 101: The Confused Deputy

Finally{}:

Making the Cut

PHP Puzzles:

Stats 101 Grade Book

DDD Alley:

Observability Lab



The Web Developer's

SECRET WEAPON!

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

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

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

DevOps monitoring, for developers. *gasp!*

Deploying web applications at scale is easier than it has ever been, but monitoring them is hard, and it's easy to lose sight of your users.

Honeybadger simplifies your production stack by combining three of the most common types of monitoring into a single, easy to use platform.

Exception Monitoring

Delight your users by proactively monitoring for and fixing errors.

Uptime Monitoring

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

Check-In Monitoring

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



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

CONTENTS

FEBRUARY 2023
Volume 22 - Issue 02



- | | | | |
|-----------|---|-----------|--|
| 2 | You Can Make the World Better
Eric Van Johson | 27 | Stats 101 Grade Book
PHP Puzzles
Oscar Merida |
| 3 | Headless Possibilities for PHP
Ivo Lukač | 30 | New and Noteworthy |
| 7 | What Can the NSA Teach Us About Debugging?
Edward Barnard | 31 | Observability Lab
DDD Alley
Edward Barnard |
| 13 | The Why and How of Building Microservices
Education Station
Chris Tankersley | 35 | PSR-15: HTTP Server Request Handlers
PSR Pickup
Frank Wallen |
| 19 | Infosec 101: The Confused Deputy
Security Corner
Eric Mann | 38 | Events, Listeners, Jobs, and Queues Oh my!
Artisan Way
Matt Lantz |
| 21 | Upgrading with Reckless Abandon: Part 2
The Workshop
Joe Ferguson | 40 | Making the Cut finally}}
Beth Tucker Long |

Edited in the basement of the NSA

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.

You Can Make the World Better

Eric Van Johson

As a PHP developer, you are uniquely positioned to shape the digital landscape and create powerful and useful applications that can change the world. Whether you are working on a small website or a large-scale enterprise system, your work has the potential to make a real difference in people's lives.

But, as with any field, the journey of a PHP developer can be challenging. There will be times when you encounter difficult bugs and frustrating challenges that test your patience and determination. In these moments, losing sight of the bigger picture and feeling discouraged can be easy.

It's important to remember that these struggles are a natural part of the development process and ultimately make the work rewarding. Each obstacle you overcome, each bug you resolve, and every time you can refactor code to make it more readable and efficient is an opportunity to learn and grow as a developer and to become even more skilled at your craft.

It's important to know that you don't have to be alone in your journey if you are a PHP developer, either professionally or as a hobbyist. The PHP community is a vibrant and supportive network of developers from all over the world.

There are several ways to tap into this community: Discord, Mastodon, Twitter, attending local or remote user groups, and of course, attending an incredible PHP conference like php[tek] in May. With hard work, dedication, and a willingness to learn, you can achieve anything you set your mind to.

And with that, let's dive into what some of the wonderful people of the community have for us in this month's edition. We start with one of our regular contributors, Edward Barnard, who gives us a bonus feature article this month, *What can the NSA Teach Us about Debugging?*

Ed dives into Bacon, encryption, ciphers, and how it helped him with debugging. I enjoyed this article, and it is a great read, well worth your time.

In our second feature article, Ivo Lukač contributes *Headless Possibilities for PHP*, where he reviews the pros and cons of headless CMSs or is it CMSes? Either way, it's another excellent read.

In Security Corner, Eric Mann brings us *Infosec 101: The Confused Deputy*, which is a security issue that arises when a trusted program or system is tricked into misusing its authority. Deception and social engineering are much more common attacks than most people realize.

Matt Lantz returns to his new Laravel focused column, Artisan Way, and contributes *Events, Listeners, Jobs, and Queues Oh my!* This article will level up your application and make it feel a lot more responsive.

Joe Ferguson completes his series of *Upgrading with Reckless Abandon: Part Two*. Not for the faint of heart. Oscar Merida takes us to school, gives us the solution to last month's puzzle, and tees up another challenge in this month's PHP Puzzles, *Stats 101 Grade Book*.

Chris Tankersley breaks down, breaking down, larger applications into smaller microservices and why adding more developers doesn't translate into quicker delivery times with the Education Stations column: *The Why and How of Building Microservices*. Frank Wallen takes time this month to review *PSR-15: HTTP Server Request Handlers*, which outlines handling and responding to HTTP requests. Edward Barnard returns in his DDD Alley column with *Observability Lab* and shows you how to solve problems with just a raspberry pi and a dream. Beth Tucker Long rounds out this month's release with *Making the Cut* and offers advice on the best way to find good developers.

Write For Us

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

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

Stay in Touch

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

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

Download the Code

Archive:

https://phpa.me/February2023_code

Headless Possibilities for PHP

Ivo Lukač

To those unfamiliar with the term headless, it is a **system with (only) an API interface for delivering content**. Even though headless solutions are somehow more bound to frontend technologies and cloud solutions, there are plenty of use cases where PHP developers might get involved.

As headless CMS solutions have gained momentum in recent years, let us check out the market from a PHP development point of view. To those unfamiliar with the term headless, it is a **system with (only) an API interface for delivering content**. It comes from the fact that a CMS is/was a layer that usually delivers HTML markup to the end-user (a visible head), so the term headless communicates that a particular CMS doesn't manage the end-user touchpoint.

The term headless spilled to other markets as well, like DAM, eCommerce, and similar, but in this article, we will focus only on CMS.

Headless Vs. Traditional/monolithic

The opposite of headless CMSs are the traditional ones, especially those with no usable public API available. That kind of product has all its features tightly coupled, focused only on the web as delivery, and is ill-equipped for multi-channel support. There is a clear distinction in capabilities between traditional and headless. If you still need just a website, you are probably better off with a more traditional CMS as you don't need to develop the "head."

Headless Vs Hybrid / Dxp

Some traditional vendors turned hybrid by adding APIs to provide a "headless" option. This move might prove valuable in the long run to keep existing clients and attract new clients by having features for both worlds. But adding a sound and usable API is not a straightforward thing to do. There might be a lot of technical debt exposed through the API and cripple the usage and developer experience. Not every API is a good API :)

More acronyms describe this kind of solution, but it looks like DXP¹ (Digital eXperience Platform) is the winner. Gartner now has a DXP magic quadrant instead of CMS. The DXP emphasizes that the system should be a focal point of various digital touchpoints, not just the web, providing APIs and tools for making those touchpoints.

Pros and Cons of Headless Cms

There are many good things about a headless approach. Headless CMSs are multi-channel by nature. Meaning

it's easier to COPE — "Create Once, Publish Everywhere" because it forces you to focus on reusable content and avoid delivery context. This enables quicker changes across channels if you set up your content governance appropriately.

With headless CMSs performance of the overall solution can be better due to the decoupled nature:

The most popular approach is to use JAMstack² and a headless CMS, which can provide very good performance. Still, you can do a fast site with traditional CMS as well, leveraging CDNs, reverse proxies, caching strategies, etc. But it is easy to forget that this approach by itself doesn't guarantee good performance. You can always make poorly performing client-side code :)

In general, headless CMSs solve some things better than traditional ones, but they also have their own problems. Traditional CMSs might be perceived as a commodity³ after so many years. Still, the fact is that systems became quite powerful products with features like Search, Media management, Multi-language, Workflow, Scheduling, Versioning, Handling edit conflicts, Taxonomies, Custom content modeling, etc.

Those are possibly "missing in action" if you consider an average headless CMS. Sure, for some features, you can find a decoupled option (DAM, search systems), but the rest need to be very core. You need to be careful not to fall into the trap and later not have those features because the CMS is not extensible nor customizable.

Headless CMSs have limitations, especially cloud-based ones, in terms of customizations. You need to be aware of what kind of resources you need to implement for your project. Not everything with the JAMstack and fat client approach⁴ is peachy.

In addition to the core content features, if you are doing a website, you need to implement the "head." "Head" features are everything a common website usually has for content delivery and more:

If you use a headless CMS, you will need to implement these features yourself — and reinvent the wheel one more time — or use another decoupled 3rd-party product for the head.

Categorizing the Cms Market

I did a simple classification of the whole CMS market. From a usage point of view, there are:

- Traditional web building CMSs

From an architectural perspective, there are:

- Monolithic, traditional CMSs, potentially with some API available
- Decoupled, either traditional CMSs that can work in headless mode, or headless CMSs with tooling for building a website

The CMS as an acronym has pushed the boundaries, so nowadays, we have either traditional CMSs on one side of the spectrum or headless CMSs on the other.

The market demand is pushing the vendors towards the center. The above-mentioned DXPs are coming from the left and are either traditional + API and/or traditional, able to work in headless mode. On the other side, pure headless is coming from the right by adding website tooling to counter DXPs.

Various Headless Options

Speaking of pure headless CMSs, you would expect they are mostly SaaS, but a good portion have an “on-premise” version; some don’t have SaaS offering at all.

There are also some distinctions on the API level. When we say API in the CMS context, nowadays, we mean RESTful⁵ API. While SOAP⁶ and other protocols may not be used as much today, they still exist, and some businesses require them to be used. On the other hand, newer protocols, such as GraphQL⁷, are gaining in popularity. What I haven’t seen so far but might be interesting to see is some kind of server push possibilities like SSE⁸.

A significant number of headless systems have a PHP SDK. Here is a list that I have found in alphabetical order: Butter⁹, CloudCMS¹⁰, Comfortable¹¹, Contentful¹², Contentstack¹³, Kontent¹⁴, Prismic¹⁵, Sanity¹⁶, and Storyblok¹⁷. Some have at least documented how to get started with PHP, like Strapi¹⁸.

All those SDKs work similarly. You need to configure the client (usually an API key or similar). Some of the SDKs can also create, update and delete content, but the most used functions would be fetching it. The most significant difference is the naming of functions so, for example, how to get a list of objects from different headless CMSs (See 1).

If there is no SDK, look for OpenAPI¹⁹ or JSONAPI²⁰ specs to make building the client wrapper easier using solutions like JanePHP²¹.

PHP Based Headless Solutions

One of the two main use cases for PHP developers is to work on a PHP based headless solution. This could be based on:

1. A traditional CMS in headless mode. Among a lot of traditional CMSs based on PHP, some added APIs (either REST or GraphQL) to the core system, like WordPress, Drupal, eZ Platform / Ibexa, Pimcore, Sulu, Concrete5, Craft, and Statamic. Some have the API available as an extension to the core, like Typo3. Can they work in a decoupled mode (delivering content only with the API) is a bit harder question to answer and would need some deeper digging and experimenting. Having an API doesn’t necessarily mean a product can work as headless. Hence there are special solutions like the Rooftop CMS, an API first distribution of WordPress trying to provide something better for such use cases. Check also a more detailed post about using WordPress as headless²². Acquia, the commercial side of Drupal, is also providing recommended ways on how to use Drupal in the decoupled or headless context²³. Ibexa (formerly eZ Platform) is doing it similarly²⁴, and Pimcore is as well, with more detailed instructions²⁵. Sulu offers 3 ways of using it as headless²⁶, and Craft has a configuration option²⁷ that switches the whole CMS to work more optimally for this use case.
2. A pure headless system. For PHP developers in need of an on-premise pure headless solution, I found only two interesting open-source PHP-based systems: API platform and Cockpit. Cockpit is a headless system built from the ground up, so it is far less complex than other CMS we mentioned. Getting started with it should be straightforward²⁸, but keep in mind the limitations of such a lightweight system. Alternatively, there is a popular platform for building API-first web apps, and it is named, well, API platform²⁹: :) It is not exactly a CMS, but it might fit some use cases. It generates a lot of scaffolding based on the data model, not just API endpoints, things like admin forms, OpenAPI specs, frontend client code, etc. It can be used by other systems to provide the API, for example, an eCommerce system like Sylius. 1. This topic was covered in one of this year’s Web Summer Camp workshops: Developing headless eCommerce with Sylius and API Platform³⁰.

Listing 1.

```
1. $page = $client->fetchPage('*', 'homepage');
2. $client->fetchPosts(['page' => $page, 'page_size' => 10]);
3. $query = new \Contentful\Delivery\Query();
4. $query->setContentType('<product_content_type_id>')
5. $entries = $client->getEntries($query);
6. $items = $client->getItems((new QueryParams())
7.     ->equals('system.type', 'article'));
8. $response = $api->query(
9.     Predicates::at('document.type', 'blog-post')
10. );
11. $results = $client->fetch(
12.     '*[_type == $type][0...3]',
13.     ['type' => 'product']
14. );
```


PHP Based “head for Headless”

If you already have a headless system that you need to use, you need to build the “head” features that integrate with whatever headless backend you have or use a “head for headless” system. Despite the fact that such solutions are dominantly cloud-based static generators and Javascript-based tools (e.g., Gatsby), we at Netgen thought there might be a need for a backend option. We created a Symfony based solution for managing the front layer decoupled from the content repository but easily integrated with any — Netgen Layouts³¹. At the moment, we have a working Contentful integration but are looking to integrate with other popular headless solutions³², be it pure headless backends or hybrid/DXP products. Besides the above-mentioned headless CMSs with a PHP SDK, other systems with a PHP SDK are interesting in this use case; for example, HubSpot³³ and Salesforce³⁴ have content management features. Netgen Layouts can work as a stand-alone application or be installed on top of any other Symfony application and provide routing for content from the local application and/or headless backend. It also provides a simple interface for non-technical users to manage what blocks will be rendered on which page, in what variation, etc.

Conclusion

Even though headless solutions are somehow more bound to frontend technologies and cloud solutions, there are plenty of use cases where PHP developers might get involved.

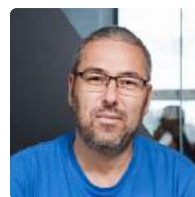
Some existing PHP-based websites might need to refactor content management features by decoupling them to a headless option.

Or existing PHP-based CMS instance needs to provide a headless mode for deploying additional websites based on fat frontend technologies, static generators, or similar.

Or you need a brand new content repo, and your know-how is PHP.

Or you are required to use a headless CMS but want to control the “head” features with a PHP-based backend.

Due to the long list of links, please see end notes on the following page.



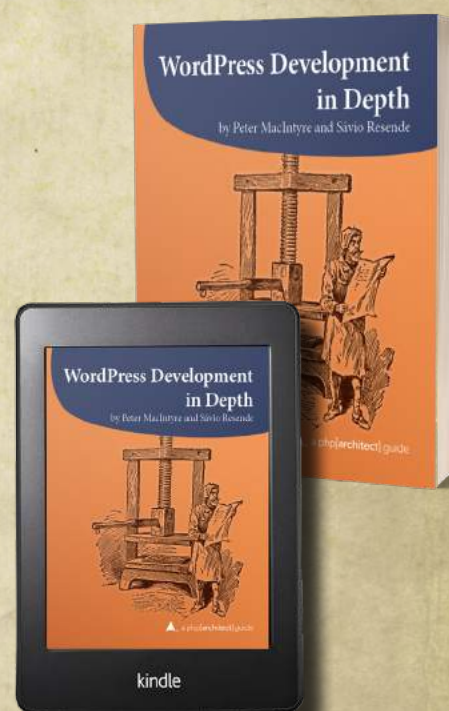
Ivo is a cofounder of @netgentweets and has been in the web engineer, architect, and speaker for 20 years. He is an organizer of @websummercamp and his current focus is on <http://netgen.io/layouts>. @ilukac

Build a custom, secure, multilingual website with WordPress

Written by PHP professionals Peter MacIntyre and Sávio Resende, this book distills their experience building online solutions for other WordPress site builders, developers, and designers to leverage and get up to speed quickly.

Order Your Copy

<https://phpa.me/wpdevdepth-book>



Resources and Links

Endnotes

- 1 DXP: <https://phpa.me/gartner-infotech-digitalplatform>
- 2 JAMstack: <https://jamstack.org>
- 3 perceived as a commodity: <https://phpa.me/markdemeny-WebContentManagement>
- 4 The JAMstack and fat client approach: <https://phpa.me/browserlondon-jamstack>
- 5 RESTful: <https://phpa.me/wikip-rest-state>
- 6 SOAP: <https://en.wikipedia.org/wiki/SOAP>
- 7 GraphQL: <https://graphql.org>
- 8 SSE: <https://phpa.me/wikipedia-server>
- 9 Butter: <https://buttercms.com/docs/api-client/php>
- 10 CloudCMS: <https://www.cloudcms.com/php.html>
- 11 Comfortable: <https://docs.comfortable.io/sdk/php>
- 12 Contentful: <https://phpa.me/contentful-developers>
- 13 Contentstack: <https://phpa.me/contentstock-developers>
- 14 Kontent: <https://phpa.me/docs-kontent-tutorials>
- 15 Prismic: <https://phpa.me/prismic-technologies>
- 16 Sanity: <https://www.sanity.io/docs/php-client>
- 17 Storyblok: <https://github.com/storyblok/php-client>
- 18 Strapi: <https://phpa.me/strapi-documentation-developer>
- 19 OpenAPI: <https://phpa.me/wikipedia-OpenAPI>
- 20 JSONAPI: <https://jsonapi.org>
- 21 JanePHP: <https://github.com/janephp/janephp>
- 22 Using WordPress as headless: <https://phpa.me/wpengine-resources>
- 23 How to use Drupal in the decoupled or headless context: <https://phpa.me/acquia-resources>
- 24 Ibexa in Headless Mode: <https://developers.ibexa.co/headless-cms>
- 25 Pimcore: <https://phpa.me/pimcore-resources>
- 26 3 ways of using it as headless: <https://phpa.me/sulu-blog-headless>
- 27 CraftCMS Configuration Option: <https://phpa.me/craftcms-docs-configsettings>
- 28 Getting Started with GetCockpit: <https://phpa.me/getcockpit-documentation>
- 29 API platform: <https://api-platform.com>
- 30 eCommerce with Sylius and API Platform: <https://phpa.me/websummercamp-developing-headless>
- 31 Netgen Layouts: <https://netgen.io/layouts>
- 32 Netgen Integration: <https://phpa.me/netgen-io-integrating>
- 33 HubSpot: <https://github.com/HubSpot/hubspot-api-php>
- 34 Salesforce: <https://phpa.me/developer-salesforce-marketing>

What Can the NSA Teach Us About Debugging?

Edward Barnard

William F. Friedman wrote training materials teaching would-be cryptographers how to break encrypted messages. Much of his advice applies equally well to modern software problem-solving and analysis.

Digital computers were essentially created to help with decrypting enemy communications. Once I realized this, it made sense to me that cryptanalysts might have useful insight into how computers work, specifically, insight into problem analysis and debugging. Indeed, they do!

William F. Friedman¹ (1891-1969) was the founding cryptanalyst of the National Security Agency (NSA). A *cryptanalyst* specializes in making and breaking codes, ciphers, and other methods of secret communication. We'll focus on what Friedman had to say, even though the NSA was not founded until 1952.

Context is important. That means we must, of necessity, take a quick dive into Friedman's world of codes and ciphers. We'll be stepping back to the 1930s. Friedman wrote a series of books titled *Advanced Military Cryptography*. The series was so significant that it was rewritten several times and, in the 1950s, gained a second author and a new title.

Figure 1.

MILITARY CRYPTANALYTICS Part I

By
WILLIAM F. FRIEDMAN
and
LAMBROS D. CALLIMAHOS

To Mr. William F. Friedman, and
LAMBROS D. CALLIMAHOS

A transcendent cryptologist and venerated teacher, without whose
prototypic texts in cryptanalytic this present volume could not have
been written, with profound admiration and respect from a devoted
pupil striving to follow in his master's footsteps.

Washington, 12 May 1956. — Lambros D. Callimahos

Figure 1 shows the new edition, now titled *Military Cryptanalytics*², by Friedman and Lambros D. Callimahos. The handwritten dedication³ from Callimahos to Friedman includes the encoded message “Bacon did not write this work.” Bacon? What did this book have to do with Bacon? (See Figure 1)

Friedman married Elizebeth Smith⁴ (1892-1980) in 1917. The Friedmans jointly taught the U.S. Army's cryptographic school from 1917-1918. That's an interesting accomplishment for the first year of marriage! That school's graduation photo (Figure 2) carries an encrypted message.

This photo is encoded using the Bacon cipher⁵, with each student either facing the camera or facing away from the

Figure 2.



camera. Elizebeth is seated at the center of the photo; William is sitting at the far right, looking left.

The encoded message reads, “Knowledge is Power”. Cabinet Magazine⁶ explains:

By facing either forward or sideways, the soldiers formed a coded phrase utilizing Francis Bacon's motto, “Knowledge is power,” but there were insufficient people to complete the “r”, and the [first] “w” was compromised by one soldier looking the wrong way...

By the time he retired from the National Security Agency in 1955, Friedman... had arguably become the most important code-breaker in modern history.

Colonel William F. Friedman kept a very special photograph under the glass plate that covered his desk... Friedman found it so significant that he had a second, larger copy framed for the wall of his study. When he looked at the oblong image, taken in Aurora, Illinois, on a winter's day in 1918, what did Friedman see? He saw seventy-one officers, soon to be sent to the war in France, from whom he had designed a crash course on the theory and practice of cryptology.

1 William F. Friedman: <https://phpa.me/wikipedia-WilliamFriedman>

2 Military Cryptanalytics: <https://phpa.me/nsa-documents-friedman>

3 Dedication: <https://phpa.me/nsa-HelpfulLinks-FriedmanDocuments>

4 Elizebeth Smith: <https://phpa.me/wikipedia-ElizabethFriedman>

5 Bacon cipher: <https://phpa.me/bacon>

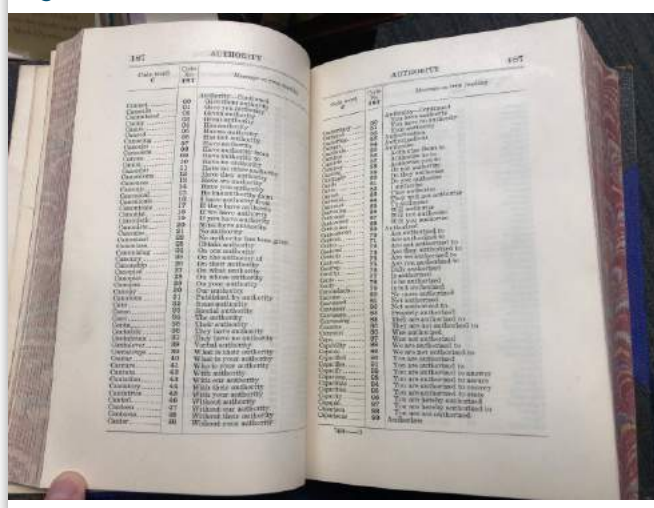
6 Cabinet Magazine: <https://phpa.me/cabinetmagazine>

My own books, naturally, carry on this tradition. The covers use Bacon's cipher to encrypt messages hiding in plain sight. This brings out a key skill with debugging: when you see something odd, follow through to the answer if you can. You'll become better and better at spotting the answers when debugging a situation.

By the 1930s, with Friedman established as the U.S. Government's chief cryptographer, he continued teaching would-be military cryptanalysts. Let's take a look at what he had to say. Those lessons apply to us today!

With *Advanced Military Cryptography, 1935 Edition*⁷, Friedman began by comparing ciphers (i.e., the encryption of text, including very-long messages) to codebooks. A codebook⁸ is a dictionary or a lookup table that allows you to look up concepts or phrases, and substitute the codebook's code word for the phrase. The codebook was literally a book. Figure 3 shows a page from the 1899 U.S. State Department code book.

Figure 3.



By this point, you'll be wondering what 19th Century codebooks have to do with debugging modern software. That's an insightful question. The answer is that, with this type of analysis, it's crucial to understand the problem and its context in-depth. We're exploring. Do bear with me; there's a reason.

The codebook idea is similar to how 18th and 19th Century navies used signal flags (Figure 4). Signal midshipmen used a codebook to translate phrases into the appropriate flag hoist. (See Figure 4)

The key feature here is the codebook's limited vocabulary. If your word or phrase is not in the codebook, you can't use the word. (There were workarounds, but the workarounds are not germane here.)

Friedman, in 1935, compared the current (1935) situation to the situation as it had been during the Great War (World War I). Armies during the Great War did not use codebooks. They were too limited. They used military ciphers.

It is necessary to add that viewpoints are always undergoing change; what is regarded as wholly impracticable today may, through some unforeseen improvement in technique, become feasible tomorrow, and it is unwise to condemn a system too hastily. For example, before the World War, and indeed for the first two years of that conflict, the use of codebooks in the theater of operations was regarded as wholly impracticable. Colonel Hitt, in his Manual for the Solution of Military Ciphers, published in 1916, says:

"The necessity for exact expression of ideas practically excludes the use of codes for military work, although it is possible that a special tactical code might be useful for preparation of tactical orders."

Also, in the official British Army Manual of Cryptography⁹ prepared in 1914 is found:

"Codes will first be considered, but as they do not fulfill the conditions required of a means of secret communication in the field, they need not be dealt with here at length."

As of 1935, however, the pendulum had swung in the other direction. Most major nations' armies now used codebooks for protecting communication.

Friedman then proposed that, with the invention of digital computers (but he did not call them that), the prohibitively expensive calculations would likely become practical:

It need only be pointed out in this connection that today code methods predominate in the secret communication systems of the military, naval, and diplomatic services of practically all the large nations of the world. Nevertheless, it is likely that within the next decade or two the

Figure 4.



7 *Advanced Military Cryptography, 1935 Edition:*

<https://phpa.me/nsa-portals-FriedmanDocuments>

8 Codebook: <https://en.wikipedia.org/wiki/Codebook>

9 *Manual of Cryptography:*

<https://phpa.me/MarshallFoundation-Library-Cryptography>

pendulum may once more swing over to the other position and cipher methods may again come to the fore, especially if mechanical and electrical cipher machines are perfected so that their operation becomes practicable for general use.

It is for this reason, if for no other, that the cryptographer who desires to keep abreast of progress must devote considerable attention to the more complicated cipher methods of the past and present time, for with the introduction of mechanical and electrical devices the complexities and difficulties of these hand-operated methods may be eliminated.

Friedman, therefore, was setting out to teach forms of military cryptography that might not appear of immediate use. He was, rather, teaching sound principles without regard to whether they immediately applied. He warned:

Consequently, if among the methods to be set forth herein certain ones appear to the student to fall outside the realm of what is today considered practicable, it should be remembered that the purpose in describing them is to present for his consideration various basic cryptographic principles, and not to set forth methods that may with a high degree of probability be encountered in military cryptography in the immediate future.

Friedman, as we all now know, was correct. The pendulum, so far as typical web software is concerned, has swung the other way. We don't use codebooks. Public-key cryptography, password hashing, certificates, SSL, etc., all use ciphers rather than codes and codebooks.

Intuition

For me, intuition comes to the forefront when it comes to debugging difficult problems. Friedman agrees! Not only does he agree, but he wrote down some useful insight. But to understand his insights, we needed the context. That's why we took this tour.

Knowledge is power.

We now know that when Friedman wrote *Military Cryptanalysis, Part I – Monoalphabetic Substitution Systems*¹⁰ in 1936, he had something similar to say. (We now know this because the NSA declassified and released the book on December 23, 2013.)

Military Cryptanalysis exposed a concept that I was handling implicitly. Friedman declaimed (page 4):

The fact that the scientific investigator works 50 per cent of his time by non-rational means is, it seems, quite insufficiently recognized. There is without the least doubt an instinct for research, and often the most successful

investigators of nature are quite unable to give an account of their reasons for doing such and such an experiment, or for placing side by side two apparently unrelated facts.

When I'm trying to solve a difficult software failure, I rely on intuition far more than I rely on specific debugging tools. The more I know about the problem context, the stresses on the system in question, and the surrounding events, the more likely I am to figure out what happened. All of these things feed my intuition.

Friedman also explained (page 3):

An active imagination, or perhaps what Hitt and other writers call intuition, is essential, but mere imagination uncontrolled by a judicious spirit will more often be a hindrance than a help. In practical cryptanalysis the imaginative or intuitive faculties must, in other words, be guided by good judgment, by practical experience, and by as thorough a knowledge of the general situation or extraneous circumstances that led to the sending of the cryptogram as is possible to obtain.

Friedman's advice, in my view, applies equally well to debugging and problem analysis. That's because it's crucial to understand the overall context and to consider the system as a whole. Friedman takes us a step further:

In this respect the many cryptograms exchanged between correspondents whose identities and general affairs, commercial, social, or political, are known are far more readily solved than are isolated cryptograms exchanged between unknown correspondents, dealing with unknown subjects. It is obvious that in the former case there are good data upon which the intuitive powers of the cryptanalyst can be brought to bear, whereas in the latter case no such data are available. Consequently, in the absence of such data, no matter how good the imagination and intuition of the cryptanalyst, these powers are of no particular service to him.

Eric Evans, in *Domain-Driven Design: Tackling Complexity in the Heart of Software* calls that "knowledge crunching". In Chapter 13, "Refactoring Toward Deeper Insight", Evans explains:

There are three things you have to focus on.

Live in the domain.

1. Keep looking at things in a different way.
2. Maintain an unbroken dialog with domain experts. Seeking insight into the domain creates a broader context for the process of refactoring.

¹⁰ *Military Cryptanalysis, Part I – Monoalphabetic Substitution Systems*: <https://phpa.me/nsa-portals-DeclassifiedFriedmanDocuments>

Evans explains that we need deep insight into the business itself before we are able to undertake transformational refactoring. Superficial knowledge of the business brings superficial results from modeling and refactoring the software.

The same, I find, is true with issue analysis and resolution. When the problems are difficult, it's the deeper understanding—empathy, if you will—that enables you to find the right answer.

When the breakthrough finally happens:

Intuition, like a flash of lightning, lasts only for a second. It generally comes when one is tormented by a difficult decipherment and when one reviews in his mind the fruitless experiments already tried. Suddenly the light breaks and one finds after a few minutes what previous days of labor were unable to reveal.

However,

Unfortunately, there is no way in which the intuition may be summoned at will, when it is most needed.

Again, in my view, Friedman's insight directly applies to computer problem analysis and debugging. Perhaps this is because computers were designed and invented for cryptanalysis. Thus, it should not surprise us that problem-solving in cryptology sheds light on problem-solving in computer science!

Even if intuition cannot be summoned at will or when most needed, can intuition be taught as a skill? I'm not sure, but we can certainly *demonstrate* those flashes of insight when they happen. Friedman's advice applies to developing our own skills in problem analysis and debugging.

How do you find and fix a problem? Certainly many known issues, challenges, or difficulties have known solutions. The more solutions you already know, the quicker you can pass from the known to the unknown.

At this point, Friedman explains, the work begins (page 2):

Success in dealing with unknown ciphers is measured by these four things in the order named. Perseverance; careful methods of analysis; intuition; luck.

Cipher work will have little permanent attraction for one who expects results at once, without labor, for there is a vast amount of purely routine labor... before the message begins to appear.

Finally, Friedman warns his students:

Often, indeed, the student will not even know whether he is on the right track until he has performed a large amount of preliminary "spade work" involving many hours of labor. Thus, without at least a willingness to pursue a fair amount of theoretical study, and a more

than average amount of patience and perseverance, little skill and experience can be gained in the rather difficult art of cryptanalysis. [Emphasis in the original.]

Superpowers

The online *First Round Review* has a useful article, "How to Spot and Magnify the Powers of Your Engineering Superheroes"¹¹. The article comes from the perspective of tech interviews, but the interview objective is to draw out and identify the candidate's "superpower".

Friedman's been describing the person we might now call "Aquaman".

Aquaman's superpower is diving deeply. This engineer is driven by solving big problems. He may not write any code for weeks on end, but will continue to dive through layers of the API to find and tackle a challenge. This superhero is nimble and acrobatic enough to penetrate operating system, database and controller layers to dig for a bug. His rare ability is being able to understand the code of each of these levels well enough to know what's happening.

Wait! No code for weeks on end? Is that allowed? What happened to the sprint? And the last sprint, and the one before that? Aquaman must be deployed with care!

"Just so," the article notes:

Early on, we had a bug in the SSL layer because we're built on top of JRuby (on the Java VM). Our Aquaman spent a month hunting the bug through JRuby, and then throughout the Java VM and down into the SSL networking layer. If I didn't have that guy who could hold his breath and had the stamina to dive to the bottom and get that SSL bug, we'd be in trouble. [Our product] would freeze occasionally because of it, and we wouldn't stay in business long with that happening. He persisted, found the bug and fixed it.

For an "Aquaman", it's the tough ones that keep the boredom away.

However, Friedman has one more lesson for us regarding keeping that boredom away. The lesson's not a good one.

By the end of 1940, Friedman's team had created an exact analog of the Japanese PURPLE¹² cipher machine without ever having seen one. In 1941, Friedman was hospitalized with a "nervous breakdown", widely attributed to the mental strain of his work on PURPLE.

Hang on to your work/life balance. That's important.

¹¹ "How to Spot and Magnify the Powers of Your Engineering Superheroes": <https://phpa.me/review-firstround-EngineeringSuperheroes>

¹² PURPLE: <https://phpa.me/wikipedia-CipherMachine>

Summary

William F. Friedman wrote a series of books aimed at introducing would-be military cryptographers to the art of code-breaking. Much of his introductory advice applies to software problem analysis and debugging. I personally suspect this is true because computers were, after all, invented for code-breaking.

Success, says Friedman, is measured by these four things in the order named:

- Perseverance
- Careful methods of analysis
- Intuition
- Luck

Intuition is essential, but it must be controlled and guided by:

- Good judgment
- Practical experience
- As thorough knowledge of the general situation or extraneous circumstances as is possible to obtain


This level of problem analysis and debugging—chasing down the tough ones—won't be attractive to those people (or their managers) who expect results at once, without labor, for there is a vast amount of purely routine labor before results begin to appear.

Often, indeed, we will not even know whether we are on the right track until we have performed a large amount of preliminary “spade work” involving many hours of labor. Absent at least a willingness to pursue a fair amount of theoretical study, and a more than average amount of patience and perseverance, little skill and experience can be gained in this rather difficult art of chasing down the tough ones.

That's what the NSA has to say. Once the easy ones have been solved, let it be your turn to root out the rest. It's fun!



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



From Capone to Cray

WHERE COMPUTERS REALLY CAME FROM

by Edward Barnard

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

<https://leanpub.com/capone>

php[tek] 2023

Early Bird Tickets Now on Sale

Here are some of the speakers



Nuno Maduro



Joe Ferguson



Jana Sloane



Beth Tucker Long



Derick Rethans



Eric Mann



David Quon



Leslie Martinich



Jason McCreary

For our full speakers list, head over to
tek.phparch.com/speakers/



May 16-18, 2023

Sheraton Suites Chicago O'Hare

tek.phparch.com

Brought to you by your friends at





The Why and How of Building Microservices

Chris Tankersley

Despite what you may think, adding new people to a team does not speed up a project but can slow it down. We'll take a look at how microservices can speed up development through the use of smaller teams, as well how to structure microservices.

Microservices have seemingly taken the development world by storm, and in the previous article, we talked a bit about what they are in relation to monolithic applications. Microservices generally help with organizational problems and allow multiple teams to work on different areas of a larger application. I wrote:

Microservices are an architectural idea that breaks down a singular application into various problem domains. Each domain is maintained as a small application by a small team that can act independently of others. These applications are then wired together, many times via APIs, to solve the overall macro problem being presented. Each team is able to work and deploy code independently of the other teams.

At a very high level, this makes sense. Complex applications will be made of smaller chunks of business logic to solve very specific problems. As applications grow more complex, those internal problems can also become more complex. This complexity means more person power, and the more people on a team, the harder it is to build an application.

Fred Brooks broached this subject in the seminal book, "The Mythical Man-Month." The idea that most people pull from that book is that adding new people to a team does not speed up a project but slows it down. Part of this is due to the ramp-up time it takes for new members to be productive, but there is another side of this that many people forget.

Group Communication

"The Mythical Man Month" cites a second reason why adding new people to a project increases delivery time. As you add new people to a project, the

communication overhead for everyone increases. Brooks defined group intercommunication as the mathematical formula:

$$n(n-1)/2$$

If "n" is the number of people on a team, as "n" increases, the number of communication channels in a team increases as well. For example, we can see the number of channels explode by adding just a few additional members:

- 4 members = $4(4-1)/2 = 6$ communication channels
- 5 members = $5(5-1)/2 = 10$ communication channels
- 6 members = $6(6-1)/2 = 15$ communication channels
- 10 members = $10(10-1)/2 = 45$ communication channels

Changes of any kind must be communicated to everyone. Developers discussing changes can easily forget to include others, leading to information not being kept in the loop. Tools like e-mail and Slack can help, but you never really replace the number of communication channels, as even disseminating information still requires each person to understand it.

Breaking up your application can help keep teams smaller, and those small teams can more quickly understand the systems and implement changes. Communication channels can be fostered between teams by specific people, which reduces overall communication channels further.

For example, at my day job, I work on a team of eight developers, with two additional developers, and we implement all of the product lines' APIs in our SDKs. This requires communication with multiple different engineering teams that can easily have just as many members as we have on our team. To

help with this, we assign one of our team members to be an intermediary with a specific product line so that we can reduce the overall number of communication lines.

Rough math would mean that, given a conservative five members on each engineering team, if we were one giant team, we would have fifty-three people on our team, excluding any managers. That would mean 1,378 potential communication channels for information to flow through.

By breaking things down into smaller engineering teams, you can have fifteen communication channels per engineering team (five members plus the intermediary), bleeding over into ours with only twenty-eight channels. We are able to keep abreast of changes and decisions on the other teams much easier than having fifty-three people all talking in meetings.

Best Tool for the Job

One of the other reasons that your engineering group may decide to move to microservices is language and tooling choices. Since each microservice is structured like its own full application, one of the technical decisions that can be made in isolation is what tools and languages to use. One service may be written in PHP, another in Java, and a third in golang.

In a traditional monolithic application, the more languages and tools you have to handle, the more complicated the overall server architecture tends to be. Deploying code for various languages can mean multiple deployment stacks, and different tools in different languages may have wildly different dependencies. This gets worse when a legacy application may hold



back newer implementations just because it needs an older language runtime.

While you can run multiple environments on a single deployment target, it just adds an additional level of complexity. More complexity means a higher chance that something will go wrong, leading to more stress during upgrades. More stress leads to a lessened want to upgrade, and longer times between upgrades lead to security issues and a higher chance of something going wrong.

The tooling to deploy microservices also tends to help encapsulate the environment for the service, usually via containers. You can have one service running PHP 7.3, another running PHP 8.0, another running NodeJS 10, and a fourth running NodeJS 16. None of these will conflict with or impose restrictions on the other services

Microservices in Practice

If our teams are small and are focused on one specific problem area, then that means the output of a team only involves the problem they are assigned to. What you end up with at the end of the day are just standard projects, all broken down via problem domains.

In true demo fashion, let's take a basic blog and turn it into various microservices. Every good blog has a way to manage blog posts, so that will be the core of one service. We will want a way to authenticate users for proper security, so we will also need a Users service.

We will store each of our services in its own folder as they are their own applications. You can set up a basic folder structure like the following:

```
/path/to/microservices/
+ users/
+ posts/
```

Posts Microservice

Since each microservice is basically its own project, the first thing we need to do is set up our `composer.json` file.

```
composer init -n --name demoapp/posts --autoload src
```

As with any project, you are free to use whatever framework you want to build your application. To cut down on a lot of the code, I will use `pocket-framework/framework`¹ just because it makes it a bit clearer on the routes I define in my controllers. Feel free to use whatever framework you like to make your own microservices.

Speaking of controllers, let's make a basic API that allows us to get a list of posts and the full content of a post. (See Listing 1)

`PostService` is a small service layer that handles working with the posts themselves via the database. There is no special code here other than it allows us to fetch and save things

to the database, just like in any other content management system.

This controller will handle the routes to display all the posts via the `/api/posts/` route and fetch the individual post via the `/api/posts/{id}` route. The only special thing we will do is

Listing 1.

```
1. <?php
2.
3. #[RouteGroup('/api/posts')]
4. class APIController
5. {
6.     public function __construct(
7.         protected PostService $postService
8.     ) { }
9.
10.    #[RouteInfo('/')]
11.    public function listAction()
12.    {
13.        return new JsonResponse([
14.            'embedded' => [
15.                'posts' => $this->postService->all(),
16.            ],
17.        ]);
18.    }
19.
20.    #[RouteInfo('/{id}')]
21.    public function getPost(
22.        ServerRequestInterface $request,
23.        string $id
24.    ): JsonResponse {
25.        try {
26.            return new JsonResponse(
27.                $this->postService->find($id)
28.            );
29.        } catch (\RuntimeException $e) {
30.            return new JsonResponse(
31.                [
32.                    'title' => 'Post Not Found',
33.                    'detail' => 'Post was not found',
34.                ],
35.                404
36.            );
37.        }
38.    }
39. }
```

return a 404 response if we cannot find a specific post.

So if we follow the install instructions for `pocket-framework` and set up our `index.php` file, we can run the API to play with using the built-in PHP development server:

```
php -S localhost:8080 -t public/
```

You could hit `http://localhost:8080/api/posts/` and get back an empty list of posts. That is perfectly fine, though, because we have not loaded anything. Baby steps!

¹ `pocket-framework/framework`: <https://phpa.me/pocket-framework>



You might ask yourself at this point, ‘How is this any different than a normal application?’ The point is—it really is not. We will simply keep the scope small for this application and have it handle posts. From all other perspectives, this is built like any other application.

User Microservice

Now that we can get posts, let’s take a look at users. Users as a concept are one of those things where the idea is simple, but the implementation can vary wildly from application to application. As applications gain more and more entry points, it becomes more important for users to stay consistent across the larger application layer.

Like the post service, we will make a brand new project—this time centered around users.

```
composer init -n --name demoapp/users --autoload src
```

We can create a basic API that allows someone to get users, create new users, and return an existing user. Since we will eventually want users to be able to create posts, let’s create a way to validate a user’s password. Most of this will be wrapped in our User service layer and, just like our Posts application, there is no secret or special code here. We are just hiding our database access code and business logic at the service layer.

```
$user = $this->userService->find($id);
$body = $request->getBody()->getContents();
$data = json_decode($body, true);
if (password_verify(
    $data['password'],
    $user['password']
)) {
    return new JsonResponse([
        'token' => $this->userService->getToken($id),
    ]);
}
```

At this point in our application, all we are worried about is having a nice API to return data. We are not concerned with how this data is used in other parts of the application and are concerned with just returning the information. (See Listing 2)

Since this application is fully standalone, we can run it just like our Post service. We will want to run it on another port, just to be safe, since our CMS will need both the Posts and User service to run at the same time:

```
php -S localhost:8081 -t public/
```

Gluing Them Together

At this point, we have two separate applications running that do what they need to do; however, they do not talk. In the case of the Posts, unless we manually enter something in the database, we do not have a way to create posts or link them to a user. We now encounter one of the first unique pieces of a microservice architecture, which is the idea of Service Discovery.

Listing 2.

```
1. #[RouteGroup('/api/users')]
2. class ApiController
3. {
4.     public function __construct(
5.         protected UserService $userService
6.     ) { }
7.
8.     #[RouteInfo('/')]
9.     public function listAction()
10.    {
11.        return new JsonResponse([
12.            '_embedded' => [
13.                'users' => $this->userService->all(),
14.            ],
15.        ]);
16.    }
17.
18.     #[RouteInfo('/', methods: ['POST'])]
19.     public function createAction(
20.         ServerRequestInterface $request
21.     ): JsonResponse {
22.         ... Create User ...
23.     }
24.
25.     #[RouteInfo('/{id}')]
26.     public function getUser(
27.         ServerRequestInterface $request,
28.         string $id
29.     ): JsonResponse {
30.         ... Get User or Fail
31.     }
32.
33.     #[RouteInfo('/{id}/login', methods: ['POST'])]
34.     public function login(
35.         ServerRequestInterface $request,
36.         string $id
37.     ): JsonResponse {
38.         ...
39.     }
40.
41.     #[RouteInfo('/{id}/validate', methods: ['POST'])]
42.     public function validate(
43.         ServerRequestInterface $request,
44.         string $id
45.     ) {
46.         ... Verify Password
47.     }
48. }
```

We need a way for the Posts microservice to validate a user token, but how does the Post microservice know that the User microservice exists? We need a way to allow microservices to register that they exist and how to access them. In many cases, this could be built into whatever deployment platform is being used or some other system for individual services to query and find out about services.

Service discovery can take many forms. Many times this is either included as a DNS layer, like in the case of Docker



Compose, Service Deployment manifests in Kubernetes, or simple key-value stores, like etcd.² For the purposes of our demo, we will make a small API that allows services to register themselves and other services to look them up by name.

Let's make another small application called `service-discovery` inside of our file structure:

```
/path/to/microservices/
+ users/
+ posts/
+ service-discovery/
```

Now let's make a super simple API that just saves things to a JSON file: (See Listing 3)

We can run this on another port, alongside the other two services:

```
php -S localhost:8082 -t public/
```

Since we have a posts microservice and a user microservice running, let's register them to this service discovery app:

```
curl -X POST \
-H "Content-Type: application/json" \
-d '{"name": "posts",
    "address": "<http://localhost:8080>"}' \
<http://localhost:8082/api/services/>
```

```
curl -X POST \
-H "Content-Type: application/json" \
-d '{"name": "users",
    "address": "<http://localhost:8081>"}' \
<http://localhost:8082/api/services/>
```

Now we can query the service discovery app for a specific service. If we wanted the users service, we could get back any addresses registered to that service:

```
$ curl <http://localhost:8082/api/services/users>
[
  {"address": "<http://localhost:8081>"}
]
```

We can go back to our Posts microservice and use the service discovery API to find the users service. We can then query that service to validate a token that was returned when a user authenticates: (See Listing 4)

The user can now send a POST request to create a new post in the system. We can take the `author_id` that they supply and an Authorization header with a token to see if the token they sent matches a known token in the Users microservice. If it does not, we return a 403 response which means they do not have authorization for this request.

Listing 3.

```
1. #[RouteGroup('/api/services')]
2. class ApiController
3. {
4.     protected array $services = [];
5.
6.     public function __construct()
7.     {
8.         if (file_exists(getcwd() . '/services.json')) {
9.             $this->services = json_decode(
10.                 file_get_contents(getcwd() . '/services.json'),
11.                 true
12.             );
13.         }
14.     }
15.
16.     #[RouteInfo('/')]
17.     public function listAction()
18.     {
19.         return new JsonResponse([
20.             'embedded' => [
21.                 'services' => $this->services
22.             ]
23.         ]);
24.     }
25.
26.     #[RouteInfo('/', methods: ['POST'])]
27.     public function createAction(
28.         ServerRequestInterface $request
29.     ) {
30.         $body = $request->getBody()->getContents();
31.         $data = json_decode($body, true);
32.         $this->services[$data['name']] = [
33.             'address' => $data['address']
34.         ];
35.         file_put_contents(
36.             getcwd() . '/services.json',
37.             json_encode($this->services)
38.         );
39.
40.         return new JsonResponse($body);
41.     }
42.
43.     #[RouteInfo('/{serviceName}')]
44.     public function getDepartment(
45.         ServerRequestInterface $request,
46.         string $serviceName
47.     ) {
48.         return new JsonResponse(
49.             $this->services[$serviceName]
50.         );
51.     }
52. }
```

If the authentication looks good, we can save the post to the database.

The main difference here between a monolithic application and a microservice is that instead of using classes that we include encapsulating the User checking, we instead call

² etcd.: <https://phpa.me/discovery-protocol>



Listing 4.

```

1. #[RouteGroup('/api/posts')]
2. class APIController
3. {
4.     protected $services = [];
5.     public function __construct(
6.         protected PostService $postService
7.     ) {
8.         $guzzle = new Client();
9.         $uri = 'http://localhost:8082/api/services/users';
10.        $response = $guzzle->get($uri);
11.        $body = $response->getBody()->getContents();
12.        $data = json_decode($body, true);
13.        $this->services['user'] = $data[0];
14.    }
15.
16.    // ...
17.
18.    #[RouteInfo('/', methods: ['POST'])]
19.    public function createAction(
20.        ServerRequestInterface $request
21.    ) {
22.        $body = $request->getBody()->getContents();
23.        $data = json_decode($body, true);
24.
25.        $auth = $request->getHeader('Authorization');
26.        $token = str_replace('Bearer ', '', $auth);
27.        $guzzle = new Client();
28.        $res = $guzzle->post(
29.            $this->services['user']
30.                . '/api/users/'
31.                . $data['author_id']
32.                . '/validate',
33.            [
34.                'json' => ['token' => $token]
35.            ]
36.        );
37.
38.        if ($res !== 200) {
39.            return new EmptyResponse(403);
40.        }
41.
42.        $post = $this->postService->createPost($data);
43.        return new JsonResponse($post);
44.    }
45. }

```

a “third party” service to handle all the work. Encapsulation begins to move toward a full-on service layer via HTTP, not just code namespaces.

Why All the Extra Effort?

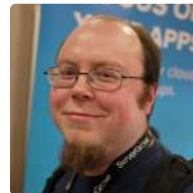
This example is very narrow in scope, but when we look at larger applications splitting these layers up into various teams can make a lot of sense. As I mentioned with the Users microservice, authentication can be quite a complicated system. While these examples gloss over exactly how we save passwords or how that authentication token is generated, it

is a perfect example of where the other layers do not need to know how all that happens.

All the Posts microservice needs to know is how to validate a token. It should not care what kind of token it is. The Posts microservice just knows that it needs to pass it to the Users service. If the token type changes down the road, as long as the API URL for validation never changes, the Users microservice can make any needed changes under the hood.

A secondary benefit is scalability. While we have two basic services, plus the service discovery layer, imagine we were serving thousands of articles a second. We could deploy multiple copies of the Posts service out on the internet in different data centers to spread the load out. These multiple copies of the microservices may also be put behind an API Gateway, which acts as a load balancer for requests. An API Gateway works with a service discovery system to route public API requests back to a specific microservice.

Next month we will wrap up our discussion about Microservices by looking at one key feature I skipped here, which is API versioning. If we are following good Microservice practices, we should be able to update the Users service without ever breaking the Posts service. This includes being able to introduce breaking changes while letting other dependent APIs upgrade at their own pace.



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

Related Reading

- *Education Station: Monolith vs Microservices* by Chris Tankersley, January 2023.
<https://phpa.me/education-Jan-2023>
- *CQRS - Reasoning and Architectural Prospects* by Alexandros Gougousis, September 2021.
<https://phpa.me/cqrs-sept-2021>

hookrelay

"I wish we had webhooks as awesome as Stripe..."

Get reliable webhook delivery, with debugging and logging, in minutes.



Stripe-quality webhooks for everyone

So you want to add webhooks to your app, and after having worked with the webhooks that Stripe provides, you want yours to be as great as theirs. Well, that's more than just sending a JSON payload to your customer's URL and calling it a day, right?



Deploy webhooks in minutes, not days.

Sign up now, and you can start receiving webhooks from integrations like GitHub, etc., even before you have an app to receive them.

<https://phpa.me/hookrelay>



Infosec 101: The Confused Deputy

Eric Mann

When two InfoSec practitioners get together, they often resort to a sort of short-hand in conversation to make things easier. This leverages slang, jargon, or other insider references that are opaque or confusing to those outside our community. Rather than coming up with new terms, it's often easiest to spend that time disambiguating the jargon already in use. This month we'll dive deep into a concept that seems to come up frequently—particularly among less technical stakeholders. This is the “confused deputy”.

Security and Trust

Nothing is more important in the security world than trust. This is central to the concepts of both authentication and authorization. It's the key concept upon which the security of any system or interaction is built. But it can also be easy to breach in more complex systems.

In a simple system, one entity might represent the root of trust for everything. In larger, more complex, or even distributed systems, you begin to rely on multiple parties to provide attestation and build that foundation of trust. Assume, for example, your simple application expands to permit users to log in with either a password or by clicking a magic link sent via email.

In this world, you need to trust both the security of your own authentication system and those of every possible email provider used by your customers. Similarly, if you leverage social login by way of OAuth, you now must also trust the security of any permitted social network to protect your application as well.

Each of these external parties is effectively a “deputy” of your application. They don't perform the heavy lifting of your business logic or operations, but when it comes to authenticating user sessions, they carry as much weight as your application itself. Not every deputy is created equally, though, and it's possible for the deputies you rely on to get confused.

The so-called “confused deputy” problem¹ is when a trusted program or system is tricked into misusing its authority. As described earlier, this could be when a social or other external login provider is tricked into providing attestation for the wrong entity.

A common example for web development is cross-site request forgery (CSRF)². In these attacks, a victim is unknowingly tricked into sending a request to a website other than the one they're visiting, frequently through the use of iFrames to steal stored cookies and session tokens. A standard example



Barney Fife, the deputy from the Andy Griffith Show, was considered somewhat inept and easily confused. His character is the stereotypical example of a “confused deputy.” [Public Domain](#).

would be creating a malicious back transfer³ by embedding a specifically-crafted request into a hidden `` tag somewhere on the page.

Confused deputy issues arise almost exclusively from a lack of planning defense-in-depth. Every engineer makes assumptions about how a system *should* work and fails to think

¹ “confused deputy” problem:
<https://phpa.me/wikipedia-confused-deputy>

² cross-site request forgery (CSRF):
<https://phpa.me/wikipedia-cross-site>

³ creating a malicious back transfer: <https://phpa.me/people-berkeley>



through how things could fail. Said another way: the happy path in coding is the expected use case, but edge cases will always exist and often result in unexpected or easily manipulated data flows.

Never Reply to a Phish

In a recent security training program, I instructed my team never to reply to a suspicious or unexpected email message. A few members were confused by this advice and asked for a deeper explanation.

For example, assume your boss emails you a message asking you to click a particular link or download a file to verify some critical information. This is behavior out of the ordinary for her, so your first inclination is to reply to the message to ask for an explanation.

Don't do it.

Instead, write a *new* message to your boss asking if the previous message was legitimate. You might also want to *forward* the suspicious message to your company's infosec team for further inspection as well. Both of these suggestions feel odd, but there's a specific reason. An attacker might be leveraging *you* as the confused deputy to attack your boss!

Rather than email a malicious attachment or link to your boss directly, they've emailed you while *impersonating* her account. She might already be on alert for this kind of attack, so the direct approach would likely fail. But sending it to you instead is much more innocuous. Once you reply, though, much of the original context is lost.

Now you've legitimately sent a message to your boss that appears to reply to a message she sent but contains malicious content. She trusts you and is far more likely to click a link or download an attachment you've sent in an attempt to "remember" whether or not she sent it in the first place. You're the deputy in this scenario, and you've inadvertently legitimized a phishing attack and aided the malicious actor in infiltrating your network.

Don't be Barney Fife.

Defense-in-depth

The best defense is a good offense.

Threat modeling is the practice of thinking through—and documenting—all of the ways your application can be attacked, misused, or potentially fail. It requires critical thinking and imagination, particularly from those heavily invested in the initial architecture who planned for the expected use cases and data flows of the system. But it trains you to expect and build around edge cases—the rare but possible ways something could fail or be abused.

This is a critical component of the concept of defense-in-depth⁴. It forces your team to instill additional security controls within your application to prevent as many edge cases as possible. It helps you to identify when and which systems are being used as deputies and how they could become confused.

The best way to defend your application is to go on the offensive. Think about the ways an attacker could bypass your security controls, even if they seem outlandish. Then ensure you're hiring the right team—using the right deputies—and not letting them be easily confused.



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)

⁴ concept of defense-in-depth: <https://phpa.me/fortinet-resources>

Secure your applications against vulnerabilities exploited by attackers.

Security is an ongoing process not something to add right before your app launches. In this book, you'll learn how to write secure PHP applications from first principles. Why wait until your site is attacked or your data is breached? Prevent your exposure by being aware of the ways a malicious user might hijack your web site or API.

Order Your Copy
<https://phpa.me/security-principles>



Upgrading with Reckless Abandon: Part 2

Joe Ferguson

This upgrade process is not for the faint of heart, nor is it for those who spend hours meticulously curating their commits and merges. We're intentionally being reckless because we're confident that our tests cover enough of our code to know when a problem will arise.

Last month we covered an ancient Laravel application that we'll bring into the modern era with reckless abandon. What gives us the audacity to think we can blatantly ignore several major upgrade guides? Tests. A long history of building PHP applications is helpful but not required. Our test suite describes how our application *should* behave; we know where the problem is if any fail. This confidence is what empowers and emboldens us to smash several framework version upgrades together.

Don't forget we have to bring all our application's dependencies along for the ride. Application dependencies could very well make or break this upgrade process, so the first major effort in this process is to ensure everything supports the framework version you're targeting. For our purposes, I've verified everything is compatible with the latest version of Laravel (9 at the time of this writing).

Version control purists are probably already cringing because they know this pull request and merge will be ugly. This upgrade process is not for the faint of heart, nor is it for those who spend hours meticulously curating their commits and merges. We're intentionally being *reckless* because we're confident that our tests cover enough of our code to know when a problem will arise. This is such a version control mess because we're going to create a *brand-new application* and paste the new files on top of the old ones. (and let Git sort them out!) In case it's unclear: create a new branch now; we're going rain fire down on this application...

Dragon Drop Upgrades

Now we're ready to dive into the heart of the upgrade, which I call the Dragon Drop method. (Say Drag and Drop fast—you should hear it). This highly offensive and barbaric method starts by creating a fresh application, and then we start dragging and dropping our old code into the new application. Here be dragons! We're doing extremely dangerous and reckless things, but we have confidence in our test suite. If you are *not* confident, make sure you keep these changes reversible. (See Listing 1)

Configuring the New Application

The config folder is the heart of building environment variables into the application configuration. Our application utilizes Laravel Scout and MySQL so we need to add the

following to our .env: SCOUT_DRIVER=database and add the following block to config/scout.php:

Listing 1.

```
1. laravel new app
2. cd app/
3. composer require laravel/ui
4. php artisan ui bootstrap --auth
5. composer require laravel/scout
6. php artisan vendor:publish \
   --provider="Laravel\Scout\ScoutServiceProvider"
7. composer require league/csv
8. composer require doctrine/dbal
9. composer require laravel/browser-kit-testing --dev
10. npm install && npm run dev
```

```
'mysql' => [
    'mode' => 'NATURAL_LANGUAGE',
    'model_directories' => [app_path().'/Models'],
    'min_search_length' => 0,
    'min_fulltext_search_length' => 4,
    'min_fulltext_search_fallback' => 'LIKE',
    'query_expansion' => false
],
```

I typically modify app.php, mail.php, and services.php with various settings and secrets to configure mail transport and any other third-party services the application may be interacting with. Make sure you review the rest of the config folder for anything you might have forgotten about. Sometimes this can be quite the adventure!

Test Configuration and Setup

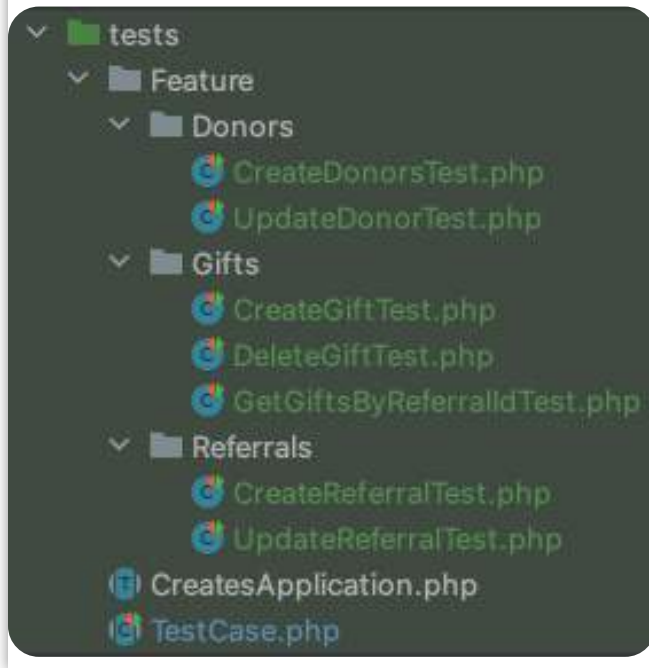
Right off the bat, we're going to remove tests/Feature/ExampleTest.php and tests/Unit/ExampleTest.php to copy over our tests from the existing application. Our test suite is broken down by donors, gifts, and referrals and we test if each can be created and updated as well as more specific tests based on requirements from our users. (See Figure 1)

Because we also removed the tests/Unit folder we need to remove the following lines from phpunit.xml:

```
<testsuite name="Unit">
  <directory suffix="Test.php">./tests/Unit</directory>
</testsuite>
```




Figure 1.



Let's run our tests and see the chaos and carnage with `./vendor/bin/phpunit` and we'll see the error "Call to undefined function factory()", our first broken thing! (See Figure 2)

So our test suite runs, and as expected, it fails miserably and at least concisely for now. Failing tests are not important here as we haven't implemented anything the tests are attempting to check.

The reason we're seeing this error is that factory usage has changed, and we need to update our tests from using `$user = factory(App\User::class)->create();` to using `$user = \App\Models\User::factory()->create();` instead. Once we do that, we see our tests fail because we haven't run our migrations since there are no tables.

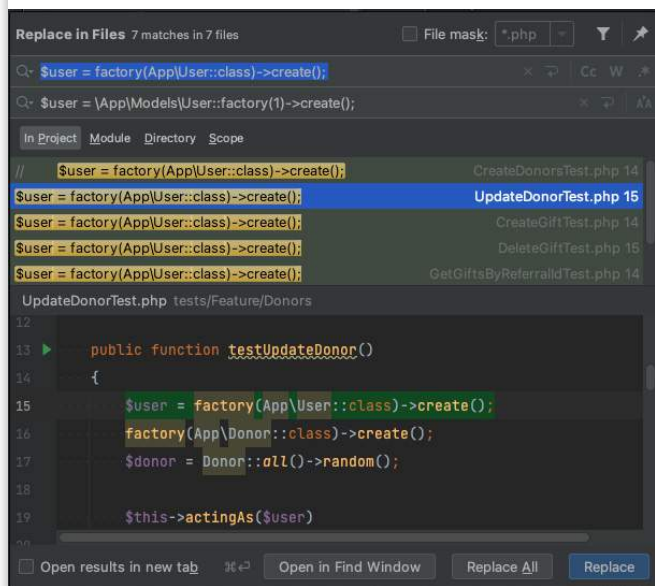
If you are not already comfortable with your code editors find and replace functionality, now is a great time to get familiar with it. I use it extensively in these types of projects to ensure I don't forget to update an old method. (See Figure 3)

The path from here is largely up to you. If you're TDD oriented, you'll probably want to chase the missing `factory()` method errors from the tests and continue on the path of red-green development. If you are more logical, you'll know that there are still a lot of files to copy over, and maybe you'll start with the database folder next. Since we're being reckless, we'll copy our `app/Models` files before continuing with the database folder, as the next step would be to give the tests some data to test against.

Figure 2.



Figure 3.



Models and Tables and Factories and More

Over the years, the Laravel framework has had different opinions on where the model class files should be stored. In



our ancient application, they are stored in `app/` and name spaced to `App`, but modern versions have moved them to `app/Models`, so we know we'll need to update our namespace for each Model as well as update references to the models as we go, from `app/User.php` to `app/Models/User.php` and `App\` User namespace to `App\Models\User`. To read more about the framework's model folder, check out Laravel Daily's post¹. We'll want to add the `use Illuminate\Database\Eloquent\Factories\HasFactory; Trait` in our tests to finish the updated model factory configuration.

Model Factories have changed over the years much like other parts of the framework. This is one scenario where instead of copying files directly, I'll recreate them using `php artisan make:factory Gift` to create `database/factories/Gift-Factory.php`, in which I'll copy the old return method from the old file to the new file and update the `$faker->` usage to the helper `faker()->`. (See Listing 2)

With our updated factories, we can update `database/seeds/DatabaseSeeder.php` with our faked data to test our application with: (See Listing 3)

Last but not least, we're going to copy our `database/migrations` files from the old application to the new application.

Listing 2.

```
1. # old factory
2.
3. $factory->define(App\Gift::class, function (Faker $faker) {
4.     $referrals = \App\Referral::all();
5.
6.     return [
7.         'description' => $faker->sentence,
8.         'amount' => $faker->randomFloat(2, 0, 10000),
9.         'referral_id' => $referrals->random()->id,
10.    ];
11. });
12. # new factory
13. public function definition()
14. {
15.     $referrals = \App\Models\Referral::all();
16.
17.     return [
18.         'description' => fake()->sentence,
19.         'amount' => fake()->randomFloat(2, 0, 10000),
20.         'referral_id' => $referrals->random()->id,
21.    ];
22. }
```

We're only copying over migrations we created in our application; we're not overwriting the migrations that come with Laravel, dated in 2014 and 2019. (See Figure 4)

We've copied our Factories, Migrations, and seeders from the old application to the new application.

¹ Laravel Daily's post: <https://phpa.me/laraveldaily>

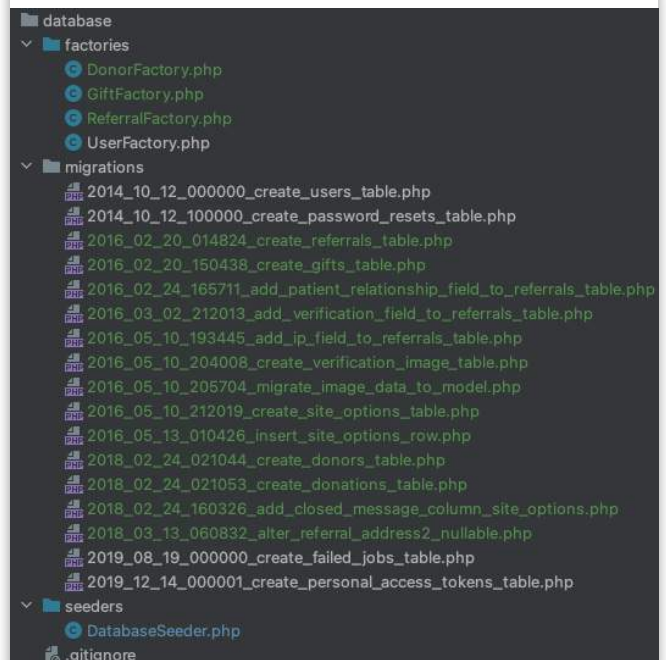
Listing 3.

```
1. # old DatabaseSeeder
2. public function run()
3. {
4.     App\User::create([
5.         'name' => 'admin',
6.         'email' => 'admin@admin.com',
7.         'password' => Hash::make('admin'),
8.    ]);
9.
10.    factory(App\User::class, 5)->create();
11.    factory(App\Referral::class, 15)->create();
12.    factory(App\Gift::class, 35)->create();
13. }
14.
15. # new DatabaseSeeder
16. public function run()
17. {
18.     \App\Models\User::factory()->create([
19.         'name' => 'admin',
20.         'email' => 'admin@admin.com',
21.         'password' => Hash::make('admin'),
22.    ]);
23.
24.     \App\Models\User::factory(5)->create();
25.     \App\Models\Referral::factory(15)->create();
26.     \App\Models\Gift::factory(35)->create();
27. }
```

The Data Migration in the Migrations

You might have caught `2016_05_10_205704_migrate_image_data_to_model.php` in the migrations list. This resulted from three months of application usage by our user, who needed more programmatic access to one of the image uploads,

Figure 4.





a verification image. The migration loops over all of our Referral models and checks to see if the verification image exists. Previously we were not tracking the hash of our uploaded file as part of the image data, just adding a field. So to move the data from the Model to a new table, we wanted to do the data migration in the database migration.

Data migrations during your database migrations can be risky and should try to be avoided. The data set migrated here is never more than a few thousand rows, so the impact is minimal. If I was ever worried, I could go back and rip out the data transformation as it's no longer needed. Would I do this on a large database? probably not but in a small project we can likely get away with it. (See Listing 4)

Listing 4.

```
1. public function up()
2. {
3.     $referrals = Referral::all();
4.
5.     foreach ($referrals as $referral)
6.     {
7.         if (empty($referral->verification))
8.         {
9.             $input = [
10.                 'referral_id' => $referral->id,
11.                 'filename' => $referral->verification,
12.             ];
13.         }
14.         else
15.         {
16.             $input = [
17.                 'referral_id' => $referral->id,
18.                 'filename' => $referral->verification,
19.                 'hash' => sha1_file($referral->verification),
20.             ];
21.         }
22.
23.         rificationImage::create($input);
24.     }
25.
26.     Schema::table('referrals', function ($table) {
27.         $table->dropColumn('verification');
28.     });
29. }
```

Listing 5.

```
1. Route::get('/',
2.     ['as' => 'home',
3.     'uses' => 'HomeController@index']);
4. Route::get('/referrals',
5.     ['as' => 'referrals.index',
6.     'uses' => 'ReferralController@index']);
7. Route::post('/referrals',
8.     ['as' => 'referrals.create',
9.     'uses' => 'ReferralController@create']);
```

Figure 5.

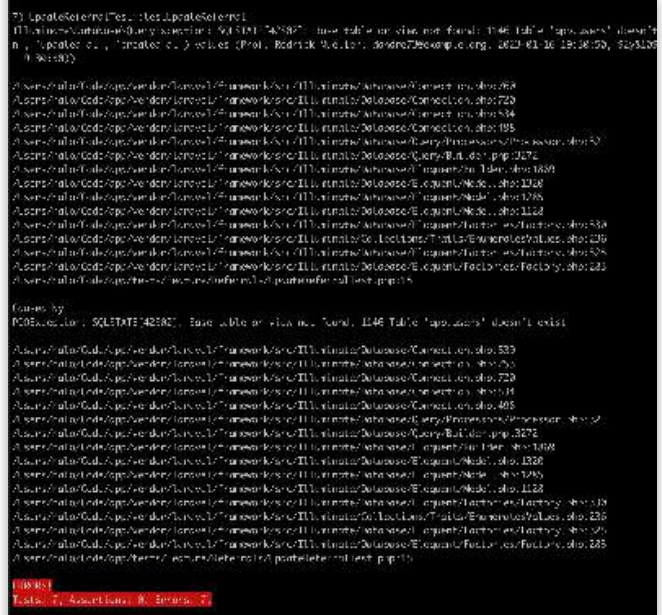
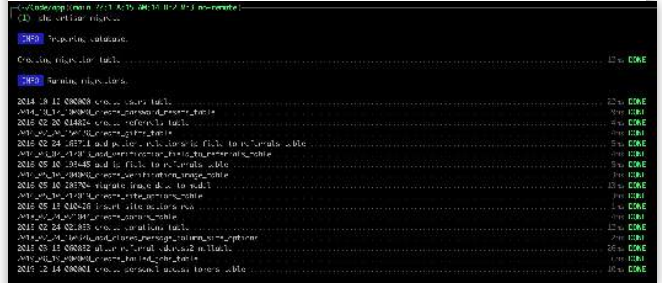


Figure 6.



Periodically you want to run your tests to keep an eye on our progress during this upgrade adventure. Currently, my tests are complaining about not having run the database migrations: (See Figures 5 and 6)

We can rerun our tests, and we'll still see failures, but by now, they're all failing because we don't have any routes. Once we add our application's routes to routes/web.php, our tests will fail because we don't have any controllers to process the requests. We can go ahead and copy our routes over to the new application, but we need to spend some time updating

Listing 6.

```
1. Route::get('/',
2.     [HomeController::class, 'index'])
3.     ->name('home');
4. Route::get('/referrals',
5.     [ReferralController::class, 'index'])
6.     ->name('referrals.index');
7. Route::post('/referrals',
8.     [ReferralController::class, 'create'])
9.     ->name('referrals.create');
```



the old route definition syntax to the new class-based one. An example of the old route syntax: (See Listings 5 and 6)

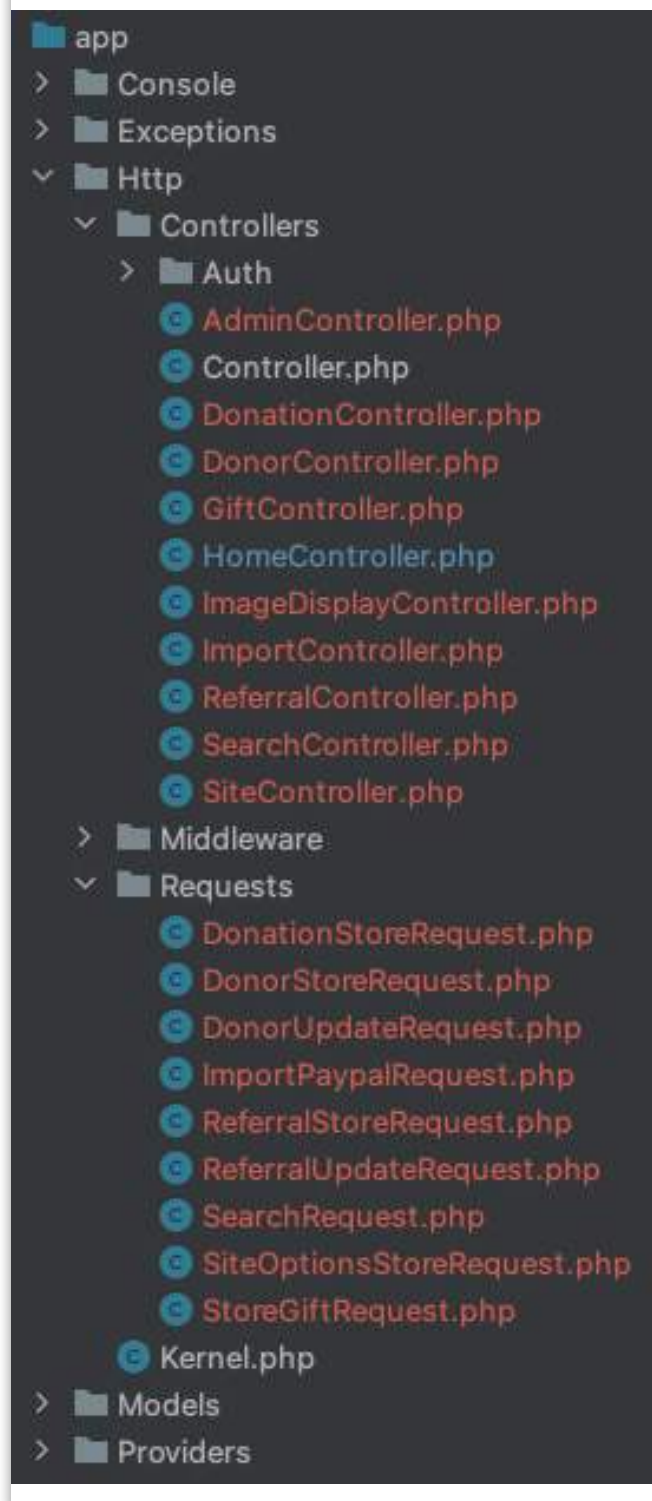
Copying App/http Files

Next up is to copy our controller classes to handle our routes. While we're grabbing our controllers, we'll also grab the Requests folder to bring our Form Request classes along with our adventure; these classes contain all of our form

validation logic. We need to do another round of Search and Replace to update our references to Models and their new namespace. (See Figure 7)

With our app/Http folder complete for now, we're ready to move on. If you're following along, you might have noticed some "class not found" errors popping up. You might need to run `composer dump-auto` to regenerate the class map of the new files we've dropped into the application.

Figure 7.



Copying Views and Front End

If modern front-end development is something that interests you, now is the perfect time to dive into your likely old layouts and views and modernize them. I like boring old front-end stuff, so in my case, I'm just copying the files I need from the public folder that has already been compiled, and I'll deal with modernizing to Webpack or Vite later, maybe never as this stuff isn't in my wheelhouse. I'm pretty aware of this technical debt that I'm dragging from the old project to the new project; make sure you are also keeping track of what trade-offs you're making. (See Figure 8 on the next page)

We don't have to make any changes to our views since our controllers are returning named views, and we haven't renamed any of them. We've picked them up from the old application and dropped them into the new one. Since I dislike the front end, this part always feels like cheating. In a commercial project, I would migrate the front end to Vite; despite the old and boring Bootstrap configuration, it works well!

Now we can run our tests and see the final result, a passing test suite! (See Listing 7)

Listing 7.

```
1. $ php vendor/bin/phpunit
2.
3. PHPUnit 9.5.26 by Sebastian Bergmann and contributors.
4.
5. ....
6.                                     7 / 7 (100%)
7. Time: 1.07 seconds, Memory: 22.00MB
8.
9. OK (7 tests, 28 assertions)
```

Now that we have migrated our application to the latest version of the framework, we're ready to pull our new application back into our repository in a clean branch to make the ugliest pull request of our lives. If you're a version control purist, look away, we're about to get nasty. (See Figure 9 on the next page)

```
$ git checkout -b reckless-abandon-upgrade
$ rm -rf .
$ git commit -am "Scorched Earth - removed files"
$ cp -r ../app .
$ git add .
$ git commit -am "Added new application"
$ git push origin reckless-abandon-upgrade
```



Figure 8.

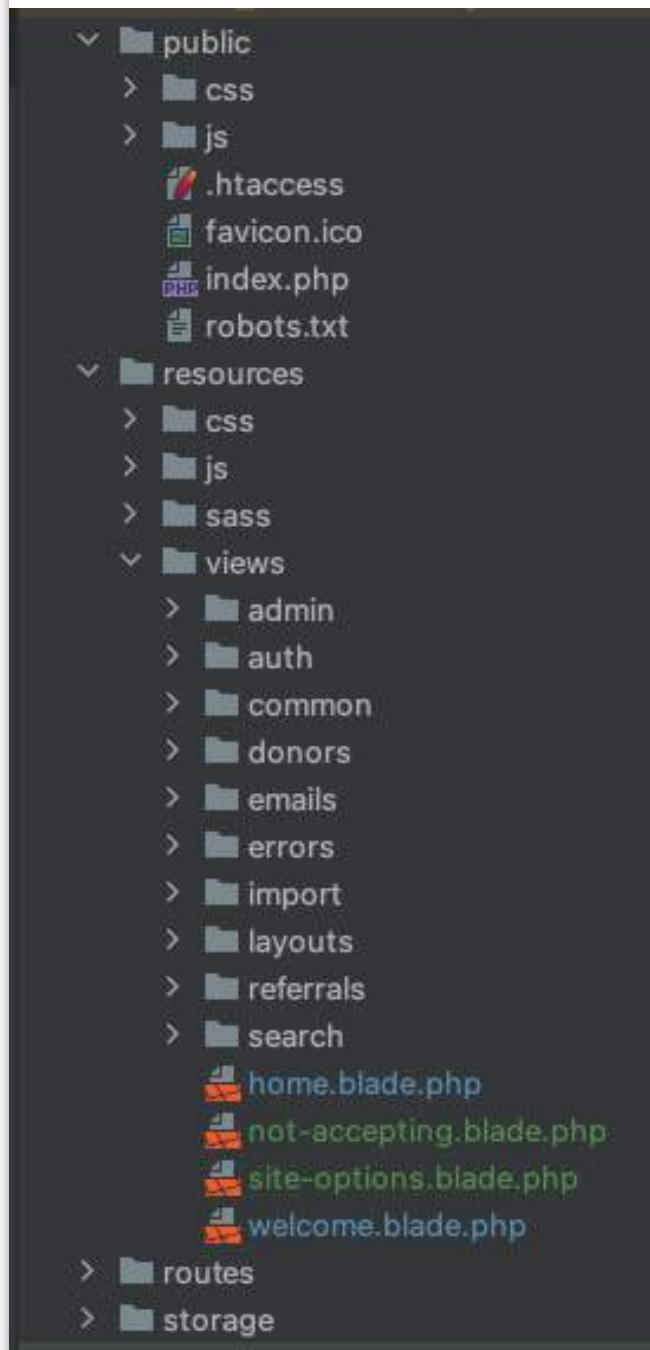
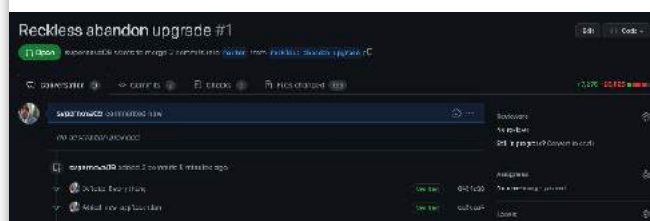


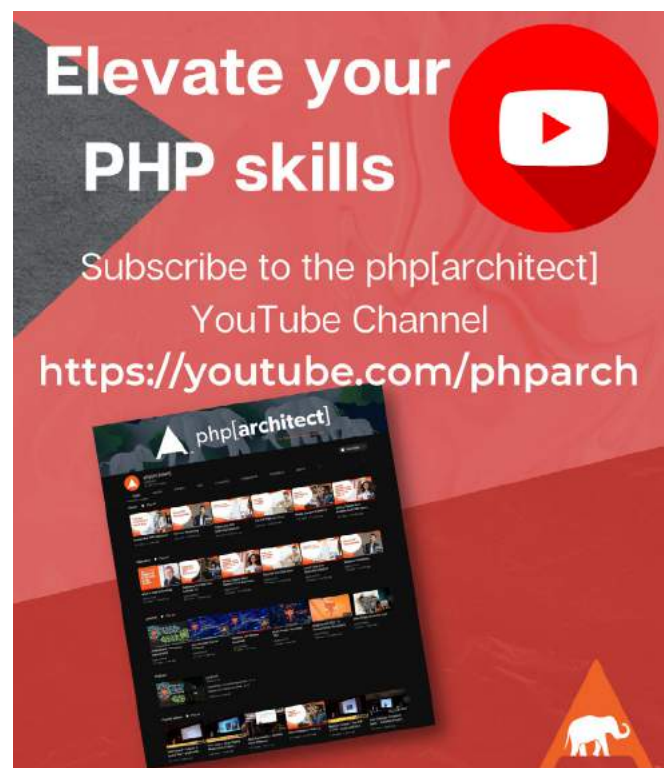
Figure 9.



The resulting PR is large, with lots of additions and removals, but we've gone from PHP 7.1 to PHP 8.2, Laravel 5.5 to 9, and we couldn't have done it without our test suite. The moral of our journey of reckless abandon is that tests *allow* us to be reckless. Arguably we're not all that reckless, true recklessness would be doing this upgrade **without tests**.



Joe Ferguson is a PHP developer and community organizer. He is involved with many different technology related initiatives in Memphis including the Memphis PHP User group. He's been married to his extremely supportive and amazing wife for a really long time and she turned him into a crazy cat man. They live in the Memphis suburbs with their two cats. [@JoePFerguson](#)



Stats 101 Grade Book
Oscar Merida

Let's go back to basics and look at some different ways to measure the average value in a list of numbers. More specifically, they're ways of measuring central tendency. These handy techniques to identify "the middle" of a set prove useful when analyzing sales, traffic, or other frequency counts.

Recap

Given the 40 grades for a course below, find the mean, median, and mode for all grades. Then plot a histogram of letter grades, ignoring pluses and minuses, given the indicated scale.

Grades:

91, 86, 70, 81, 92, 80, 73, 85, 70, 87, 74, 82, 77, 83, 90, 90, 87, 83, 93, 72, 84, 87, 83, 73, 86, 81, 86, 77, 75, 89, 77, 80, 79, 95, 69, 78, 89, 84, 70, 72, 89,

Letter Grade Assignment

Letter	Percentage	Letter	Percentage
A+	97–100%	C	73–76%
A	93–96%	C–	70–72%
A–	90–92%	D+	67–69%
B+	87–89%	D	63–66%
B	83–86%	D–	60–62%
B–	80–82%	F	0–59%
C+	77–79%		

Preparations

Let's quickly turn our set of grades above into an array of integers.

```
// assume it's all on one line
$input = '91, 86, 70, 81, 92, 80, 73, 85, 70, 87, 74, 82, 77, 83, 90, 90, 87, 83, 93, 72, 84, 87, 83, 73, 86, 81, 86, 77, 75, 89, 77, 80, 79, 95, 69, 78, 89, 84, 70, 72, 89, ';
$grades = array_map(
    fn($i) => (int) trim($i),
    explode(',', trim($input, ' '))
);
```

I appreciate PHP's succinct way of working with arrays and short arrow functions once you get used to it. We'll get an array that looks like Listing 1.

Listing 1.

```
1. array(41) {
2.   [0]=>
3.   int(91)
4.   [1]=>
5.   int(86)
6.   [2]=>
7.   int(70)
8.   [3]=>
9.   int(81)
10.  [4]=>
11.  int(92)
12.  [5]=>
13.  int(80)
```

Calculating the Mean

The mean is the usual number we think of when we think of the "average." It's a straightforward calculation. We need to sum up the values and divide them by the number of items in our set.

```
if ($grades) {
    $mean = bcdiv(array_sum($grades), count($grades));
    echo "The mean is $mean" . PHP_EOL;
}
// The mean is 81.682926829268
```

Finding the Mode

The mode is the number, or numbers, that occur most frequently. You can have more than one, so we'll need to consider that possibility in our solution. We can solve this using built-in PHP functions and without any loops.

```
// 1. get the frequency of each grade
$freq = array_count_values($grades);
// 2. Find the highest frequency
$maxFreq = max($freq);
// get the unique set of grades with that frequency
$modes = array_filter($freq, fn($i) => $i == $maxFreq);
// clean it up for output
$modes = array_keys($modes);
sort($modes);
echo "The mode(s) with a frequency of $maxFreq are\n" .
    implode(' ', $modes);
// The mode(s) with a frequency of 3 are
// 70, 77, 83, 86, 87, 89
```



Where's the Median?

The median is the number that is right in the middle of your data set. Half the numbers are above it, and half are below. Calculating this number might be tricky since it's based on how many items are in our array.

```
// Sort the grades
sort($grades);

if (count($grades) % 2 === 1) {
    // we have an odd number, and can use the number
    // in the exact middle of our array.
    // For a zero-indexed array that's the integer below
    // 1/2 of the total number items
    $midKey = floor(count($grades) / 2);
    $median = $grades[$midKey];
} else {
    // we have an even number, we need to average the
    // numbers around the middle of our array.
    $lowKey = count($grades) / 2;
    $highKey = $midKeyHigh + 1;
    $median = ($grades[$lowKey] + $grades[$highKey]) / 2;
}

echo "The median is $median";
```

Results

We've calculated three measures of central tendency, each giving us a different take on where most people fell on our grading scale. The mean is 81.6829, which tells us that most of the grades are near that value. However, the mean can be skewed by a single high or low outlier — you know, that kid who's always “blowing the curve.”

We have the other measures to give us a different perspective. The median of 83, slightly higher than our mean, tells us half the class is below that grade and half the class is above it. So, most people are closer to 83 and 81.6829. If there's a stark difference between the mean and median, you likely don't have an even or normally distributed data set.

Finally, our modes of 70, 77, 83, 86, 87, and 89 with a frequency of 3 tell us that 18 of our 40 students earned that mark. If we were to pick a score at random, we would have a 45% chance of finding a student with one of those three grades.

Visualizing Our Distribution

Our next step is to plot a simple histogram of the grade distribution—ignoring pluses and minus attached to letter grades. A histogram is a kind of bar chart where we plot a range of values on one axis and the frequency of observed values on the other.

Since we can ignore pluses and (did you catch that?), our simplified buckets for frequency are shown in the table below. We can also assume that final grades are rounded to the nearest integer.

Letter	Percentage	Letter	Percentage
A	90–100%	D	60–69%
B	80–89%	F	0–59%
C	70–79%		

Again, we can do this without looping—if we apply knowledge of array functions as shown in Listing 2.

Listing 2.

```
1. // letter grade frequency
2. rsort($grades);
3.
4. // so our histogram plots A's first
5. // Use array map to convert each int to a letter grade
6. $letters = array_map(function($grade) {
7.     switch (true) {
8.         case ($grade >= 90): return 'A';
9.         case ($grade >= 80): return 'B';
10.        case ($grade >= 70): return 'C';
11.        case ($grade >= 60): return 'D';
12.        default: return 'F';
13.    }
14. }, $grades);
15.
16. // now count the frequency of each
17. $letterFreq = array_count_values($letters);
18.
19. // output a basic histogram
20. echo PHP_EOL . "Histogram of Grades" . PHP_EOL;
21. foreach ($letterFreq as $letter => $count) {
22.     echo "{$letter} | " . str_repeat('█', $count)
23.     . " ({count})" . PHP_EOL;
24. }
```

We get a rudimentary but useful graph of the grade distribution.

Histogram of Grades

```
A | █████ (5)
B | ████████████████████ (21)
C | ████████████████ (14)
D | █ (1)
```

Grade Deviations

Let's continue with grades and statistics for next month's puzzle. Consider the grades in the table on the next page for a set of students. Each is a final grade in one of 6 subjects. Based on the table below, calculate each student's final Grade Point Average (GPA), the range of GPAs, and the standard deviation of the GPAs.

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

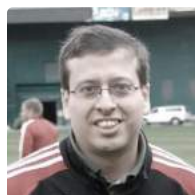
¹ the grades: <https://phpa.me/gist-omerida>

Data for next month's puzzle.

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

Use the following table to calculate the GPA.

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



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](#)

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 PsySH¹ 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.

¹ PsySH: <https://psysh.org>

New and Noteworthy

PHP Releases

PHP 8.2.1 Released:

<https://www.php.net/archive/2023.php#2023-01-05-2>

PHP 8.1.14 Released:

<https://www.php.net/archive/2023.php#2023-01-05-1>

PHP 8.0.27 Released:

<https://www.php.net/archive/2023.php#2023-01-05-3>

News

php[tek] 2023 Speakers and Talks

Here at php[architect], we are getting really excited to be bringing back php[tek] this year, May 16th through the 18th, at the Sheraton O'Hare Hotel and have posted the speakers and talks. Early Bird tickets are still available.

<https://tek.phparch.com/speakers/>

Symfony 6.2.5 Released

The latest release of one of the strongest frameworks.

<https://symfony.com/blog/symfony-6-2-5-released>

Jet Brains: The State of Developer Ecosystem 2022

These are the results of the sixth annual survey conducted by JetBrains to capture the landscape of the developer community.

<https://www.jetbrains.com/lp/devecosystem-2022>

PHP Annotated – January 2023

January installment of PHP Annotated, catching you up on the most exciting things that have happened in the PHP world.

<https://phpa.me/annotated-2023-01>

All of the Easter Eggs in PHP

I'm a little late to the party on this one, but I thought it was cool. A list of PHP Easter Eggs. I remember a couple of these, specifically the April 1st one, and how cool I thought they were at the time.

<https://php.watch/articles/php-easter-eggs>

Why Curly Brackets Go On New Lines

Where will you fall on this debated topic?

<https://stitcher.io/blog/why-curly-brackets-go-on-new-lines>

5 Websites to Learn More Sql

Read content from these 5 Websites for more knowledge and understanding of SQL

<https://phpa.me/learn-more-sql>

php[podcast] is Back

Eric and John return to the mic with a new format for the php[podcast]. They talk about what's new in the magazine and what might be coming down the road, as well as the latest on php[tek] 2023. If you haven't listened to the podcast in a while, give this new format a try and let us know what you think. You can catch the audio podcast where ever you get your podcast from or check out the video stream on our YouTube channel at:

<https://youtube.com/phparch>

https://phpa.me/ep_2023_1_1

Getting Out of PHP Dependency Hell with Composer

While you are checking out some of our new content, Scott Keck-Warren released a new educational video where he talks about Composer and some of its features.

<https://phpa.me/vid-dependency-hell>

Observability Lab

Edward Barnard

I realized that I need to experiment with various off-the-shelf products, such as Elasticsearch. Here is how and why I set up a Raspberry Pi as a safe testbed. We'll repartition a USB hard drive to be suitable as the boot device while reaching past the boot device limitations.

Can a Raspberry Pi help with Domain-Driven Design? Yes, it can! We'll first look at why I took this approach (and thus why you might choose to as well). Then we'll walk through the exact steps I took to create a usable environment.

The Problems to Solve

The USA Clay Target League facilitates trap shooting as a competitive sport at the high school and college levels. We want to provide greater visibility to our stakeholders. For example, as season registration opens, we would like to provide near-real-time dashboards showing year-over-year registration numbers (among many other things).

The idea of “greater visibility” is not specifically a PHP question. My development environment (a MacBook laptop) is well suited to PHP-specific exploration; however, it's not as well suited to the more wide-ranging investigation I have in mind here.

One option is to run Linux inside VirtualBox or VMWare Fusion. But there's another option, and that's what we'll explore here: the Raspberry Pi.

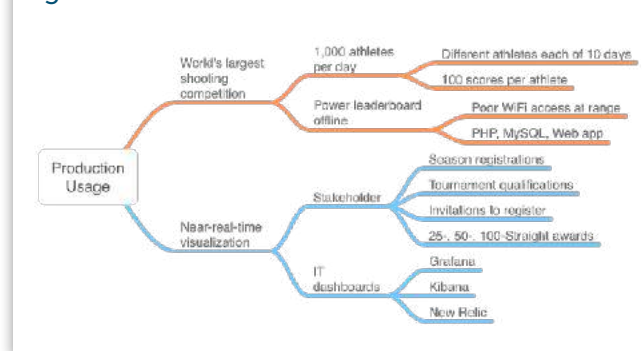
The Raspberry Pi has become far easier to use over the years. I ordered a 4 GB Raspberry Pi 4B¹ from RAK Wireless. I ordered the power supply and preloaded Raspbian OS on a micro SD card from the same sales page. (The item is still in stock as of 15 January 2023.) The Pi now comes with both wired and wireless connectivity, two USB-2 and two USB-3 slots, and two micro HDMI connections.

There's a large community with many tutorials and videos online. Ubuntu Desktop officially supports the Pi. Based on what I had in mind (which I'll explain below), I also ordered a micro HDMI cable to connect to my spare monitor and a wired (USB-2) keyboard and mouse.

I ordered a USB hard drive for additional storage. After all, I expect to index many documents into Elasticsearch. Finding a hard drive with a separate power supply was more difficult. This is needed because if the Pi is supplying power to too many devices, the Pi can starve itself of power and thus fail. The smallest I could find is a Western Digital My Book with 4 TB capacity.

I created a mind map (Figure 1) showing the scope of the problem. We initially have two production usages of the Raspberry Pi as production hardware.

Figure 1.



The upper (orange) branch of the mind map describes the annual Minnesota State High School League Trap Shooting Championship². We expect 7,900 athletes to compete over nine days in June 2023. It's actually nine separate daily championships, with roughly 900 competing each specific day. The mind map shows I'm planning for the slightly larger capacity of 1,000 per day for ten days. The competition occurs in a rural area with minimal cell phone service or Wi-Fi connectivity.

All shooters shoot 100 clay targets, with each target scored as a “hit” or a “miss”. We are thus entering around 90,000 scores each day (900 athletes times 100 targets each). We have discussed bringing up a laptop or desktop computer to display leaderboard standings on a large screen without having to be continuously online. We've been considering the Raspberry Pi as an inexpensive and lightweight solution.

Although the Minnesota tournament is the largest, we facilitate dozens of similar tournaments around the country at approximately the same time. Most large trap-shooting ranges are rural with unreliable Wi-Fi connectivity. Having physical compute hardware on-site would have advantages over our current web (cloud-based) processing. Thus replicating a Raspberry Pi solution might actually be a good idea!

The lower (blue) branch of the mind map shows a typical “dashboard” approach. Connect to a screen on the wall showing something of interest in real-time or near-real time. This could include the number of season registrations by state or region, the number of athletes who hit 25 consecutive targets this week, and so on. We could also show monitoring

1 4 GB Raspberry Pi 4B: <https://phpa.me/rakwireless-raspberry-pi4>

2 Minnesota State High School League Trap Shooting Championship: <http://championship.mnclaytarget.com>

of server load, database traffic, or Domain Events coming from our PHP codebase.

Gaining Experience

The problem, from my standpoint, is that we have too many options to consider. I would like a completely safe “laboratory” environment. That’s an environment where it doesn’t matter if I do something wrong or even “brick” the system. I’d also like to be able to run something for 24 or 48 hours. If I do that from a virtual linux setup on my laptop, my laptop needs to stay connected.

With the Raspberry Pi (or something similar), I have a relatively safe way to explore and experiment. If I screw up the operating system, I just pop in a different micro SD card and boot back up. Since I’ll have a full linux environment, I can do backups or repository commits of anything I don’t want to lose when I reformat the hard drive.

However, the Raspberry Pi is based on the chips used in mobile devices (ARM architecture) rather than the chips used in PCs and web servers (x86 architecture). That means most software does *not* support Raspberry Pi.

Does it support the software I need for exploration? I see that:

- The Raspberry Pi foundation includes a project Build a LAMP Web Server with WordPress³.
- There are several tutorials aimed at installing Elastic Stack (formerly known as ELK stack) on Raspberry Pi.
- Docker and Portainer run on the Pi.
- Kubernetes appears to support Raspberry Pi as well.

That’s enough support to justify exploring the Raspberry Pi as my laboratory environment. There’s another bonus for me personally: I’ve been interested in learning ARM architecture and assembly language. That, obviously, requires having an ARM environment where it’s safe to *really* mess things up. Now I have one!

Scope of Exploration

I laid out another mind map (Figure 2) showing possible areas to explore. This map exposes something interesting.

Consider the ELK stack: Elasticsearch, Logstash, and Kibana. The general idea is that Logstash will watch your various server logs, extracting information and shipping it over to Elasticsearch. Elasticsearch indexes (i.e., ingests) this information into a central datastore. Kibana then queries the Elasticsearch database in various clever ways to present actionable information.

There’s an interesting insight here. We’re describing a PHP application, but nothing just described comes from our PHP code. Yes, I’m stating the obvious, but think about that. The

visibility, the actionable information, comes from *infrastructure*. To gain actionable insight, we needed to compose different products (applications) in a way that they work together to produce that insight.

In other words, I need a laboratory where I can safely explore infrastructure. If I come up with something useful, we’ll probably want to deploy exactly that to production—or, at least, to sit alongside production.

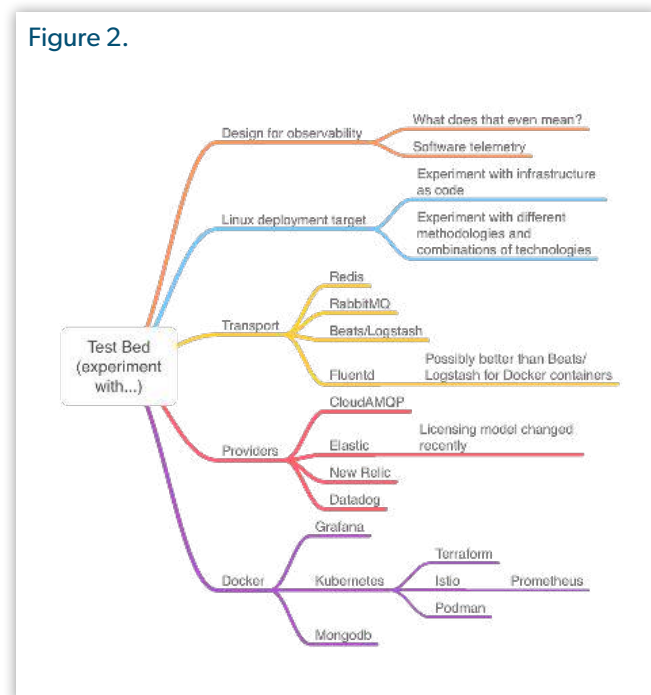
Thus, “deployment” is a fundamental requirement. But that means we’ll want an *organized* and *reproducible* deployment. Raspberry Pi is not the same environment as an instance on Digital Ocean, but we’ll want to achieve the same result! This brings in the idea of scripted deployments, perhaps containerized with Docker. We might consider Ansible, using Raspberry Pi as a deployment *target*, running Ansible on my laptop.

However, we already know we’re potentially dealing with complexity. Therefore, Kubernetes may be worth exploring, which leads us to consider Podman, Terraform, Istio, and Prometheus.

That brings us back to the original insight, which is that this is all an exercise in mixing, matching, and gaining experience with existing products. There’s an old saying, “When you’re up to your neck in alligators, it’s hard to remember that your initial objective was to drain the swamp.”

The original objective was to achieve somehow near-real-time visualization of business-specific data for stakeholders and performance/capacity monitoring for IT staff. This led to the insight that we need to design a system for observability. That’s going to require some experimentation.

Figure 2.



³ Build a LAMP Web Server with WordPress:
<https://phpa.me/projects-raspberry-pi4>



The Testbed

Fortunately, there is a *lot* of information on getting started with Raspberry Pi and using it for various projects. I found The Official Raspberry Pi Beginner's Guide: How to use your new computer⁴ by Gareth Halfacree, the most useful. It's available as a full-color paperback, but it's also available as a free PDF⁵.

I decided to install the full Ubuntu Desktop as described in these two official Ubuntu videos:

- How to get started with Ubuntu Desktop on Raspberry Pi⁶, 4 minutes
- Ubuntu Desktop on Raspberry Pi⁷, 20 minutes

Given an SD and micro SD card reader/writer and some blank 64GB micro SD cards, I used the Raspberry Pi installer and chose Ubuntu Desktop per the above video instructions. Use tutorials as needed to open the command line (search "terminal" on ubuntu desktop), and as root ("sudo bash"), update software:

- apt update
- apt upgrade
- apt autoremove
- reboot

One of the Linux software distribution channels is "snap", but the snap store has trouble updating itself. Kill the process and then have it update itself like this:

- pkill snap-store
- snap refresh snap-store

Ubuntu was annoying me by logging me out every five minutes. Search for "settings", and within settings, choose privacy, then choose screen. Turn off any behavior you deem annoying.

Be sure to do a system shutdown (sudo shutdown 0) prior to turning off the power. There is no power-off switch; you either pull the plug or turn off the power strip (which *might* disable surge protection; I'm not sure on that).

The biggest difficulty, and why I wrote this article, has to do with the external USB drive (the 4 TB MyBook). I used the Pi Installer to write the Ubuntu Desktop operating system to that drive, setting it up as the boot device. Simply plug the MyBook into one of the Pi's USB-3 slots. Open up the Pi installer and choose MyBook as the destination.

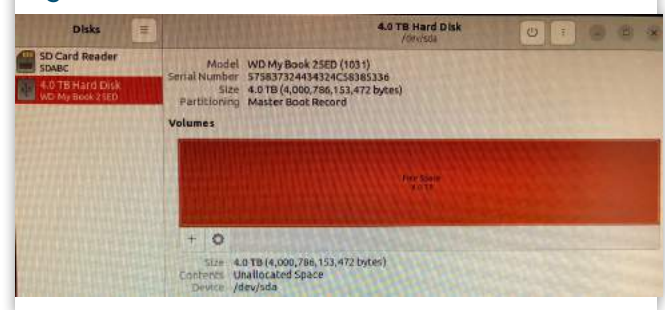
Shut down the Pi (shutdown 0) and turn off power. Remove the micro SD card, leaving MyBook connected. Power the Pi back on. It will hang for 30 seconds, waiting for the micro SD

card to show itself. Then it checks each of the four USB slots in turn, looking for something to respond which contains a boot block. In my case, the screen shows 34 seconds of uptime when it finally jumps into action, proceeding to start up. That is, it appears there's a 30-second timeout for the micro USB reader and a 3-second timeout for each USB slot. This means I could probably move the MyBook to the other USB-3 slot and save myself three seconds per day.

Meanwhile... that boot record on the MyBook is a bit of a problem. The following figures are photos of the screen, rather than screenshots, because I was just about to reformat the hard drive. I'd lose the screenshots!

Figure 3 shows the "Disks" utility showing MyBook. It shows the partitioning style is Master Boot Record (MBR). That's an old format (Windows 95, if I recall correctly) limited to 2GB. This means formatting the unused space fails with the message, "Message recipient disconnected from message bus without replying (g-dbus-error-quark, 4)". I tried formatting the whole thing as ext4, the standard Linux format, but that attempt also failed with the same error message. The disk utility still uses the existing MBR partition map.

Figure 3.



Next, use the command-line partitioning utility to change the partition map. See How to use parted on Linux⁸. The "parted" utility accepts its own commands:

- "mklabel gpt" to change the partition map
- "unit TB" to change units to terabytes
- "mkpart" to create the partition (I used MyBookA as partition name, ext4 as partition type, 0 as start, 4 as end)
- "quit" to execute the above and exit

Now re-image the operating system with the Raspberry Pi Imager.

There's another utility, gparted. See How to use GParted on Ubuntu⁹. This utility shows we are back to the MBR partition map, shown as "fat32" in Figure 4.

⁴ The Official Raspberry Pi Beginner's Guide: How to use your new computer: <https://www.amazon.com/dp/191204773X>

⁵ free PDF: <https://phpa.me/raspberry-pi4-beginners>

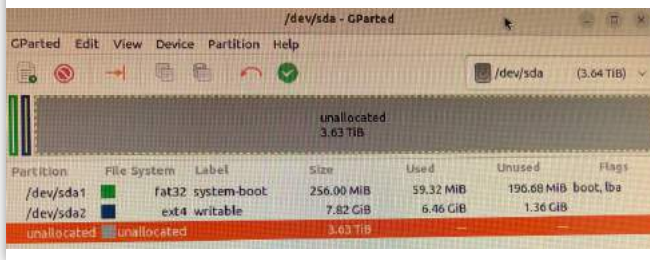
⁶ How to get started with Ubuntu Desktop on Raspberry Pi: <https://www.youtube.com/watch?v=ZWBe2E1Sgi0>

⁷ Ubuntu Desktop on Raspberry Pi: <https://www.youtube.com/watch?v=OpT4-RcTERU>

⁸ How to use parted on Linux: https://linuxhint.com/parted_linux/

⁹ How to use GParted on Ubuntu: https://linuxhint.com/gparted_ubuntu/

Figure 4.



Could we have skipped a step? I suspect so. We needed to format the hard drive in ext4 format before using the Raspberry Pi imager. It came out of the box formatted NTFS if I recall. Knowing what I know now, I probably could have reformatted to ext4 before the first round of imaging the MyBook.

Either way, the Raspberry Pi Imager returns the partition map to MBR format. That means we're limited to 2 TB partitions and a maximum of four partitions.

- The first partition, /dev/sda1, is the boot block. It's in fat32 format with MBR partition map. That's what limits us to 2 TB partitions and a maximum of four partitions.
- The second partition, /dev/sda2, is ext4 format. That's a good thing. It's 7.82 GB, which is a problem.

That second partition, called "writable", is the root filesystem. I quickly found that its 1.36 GiB of free space overflows the first time I run software updates! I found that out when I used the unallocated space to create two more partitions.

Instead, I increased the root filesystem allocation to 2 TB. See Figure 5. Remember, I'm not going for optimal server configuration; I just need something where I can experiment with infrastructure components and visualizations.

Figure 5.



Finally, I created a third partition from the remaining unallocated disk space. See Figure 6. Now the Raspberry Pi is able to boot from the USB drive, use all 4 TB of space, and stay within the four-partition limit. If I am using the Raspberry Pi Imager, this means that any space beyond 6 TB is unusable on the boot drive. Incidentally, I'll experiment with making the boot block considerably larger next time I create a boot drive using this procedure. The boot image can grow over time.

Figure 6.



Virtualbox

VirtualBox does not run on the Raspberry Pi. I took far too long to figure that out. I'm glad I didn't ask on Stack Exchange because the reason is so obvious—and I missed it. I found various tutorials online showing how to install Virtual Box 7 on Ubuntu Desktop 22.10. That was my objective.

I was not distinguishing between "amd" and "arm". I was seeing the error messages but not *thinking*. I don't normally check hardware support, and I need to get back into that habit. VirtualBox does not run on ARM hardware.

Summary

Depending on what you need to explore, the Raspberry Pi might be a good route to creating a safe "test bed". It's also viable for production use when having your own physical hardware makes sense. In my case, we're considering the Pi for displaying leaderboard information in remote venues without reliable Wi-Fi connections. Given this added experience with Raspberry Pi, I expect we'll also use them in-house for dashboard displays.

I do seem to get better performance with the root filesystem on a USB-3 connection rather than on the micro SD card. Since the Raspberry Pi Installer writes an old-style MBR partition map, I found a way to partition and use the full capacity of a 4 or 6 TB USB drive.

The performance difference probably doesn't matter as much when using the Pi as a headless production device. You're not sitting at a keyboard waiting for it to catch up. But as a test bed, when I am sitting at a keyboard, the added responsiveness does help.



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](#)



PSR-15: HTTP Server Request Handlers

Frank Wallen

This month we're introducing PSR-15¹, which proposes standards for handling and responding to HTTP requests through request handlers and middleware. A request handler receives an HTTP message (see PSR-7², processes the request, and returns a response. A middleware component processes incoming requests, often in combination with other middleware, and will perform specific actions before the receiving application handles the request. There are two github repositories for this PSR: Server Request Handlers for Middleware ([http-server-handler](https://github.com/php-fig/http-server-handler))³ and HTTP Server Middleware ([http-server-middleware](https://github.com/php-fig/http-server-middleware))⁴.

The concept of middleware has been around for a long time. According to Wikipedia⁵, the term has been used since 1968 and was later made popular in the 1980s to connect newer applications to legacy applications. In modern frameworks today, middleware is often used to handle responsibilities that include:

- Authentication and Authorization
- Rate-limiting requests
- Logging
- Handling errors in the request
- Cookie encryption
- Sanitizing data
- Tagging

Middleware sits between the transport layer and the application layer. Think of it as a set of passageways you must pass through to reach your destination, with some wide open and others with doors closed. An authentication door requires your key to get through; an authorization door may require a certain kind of key and opens to different passageways depending on the shape of your key. In some cases, a passageway may attach some extra metadata to you, such as timestamps or flags for another passageway to understand you better.

Middleware implementation can certainly vary. According to the PSR-15 metadocument⁶, there are generally two approaches: Double pass and Single pass. The double pass is named so because of the signature `fn(request, response, next): response` where `request` and `response` implement `ServerRequestInterface` and `ResponseInterface`, and `next` is a callable that will send the request and response to the

next middleware. Sending **requests** and **responses** is why it's called a 'double pass' and is currently the more common and newer approach. A strong feature of a double pass is the ability for the middleware to change or add to the response object before it passes it forward. On the other hand, the 'single pass' approach will provide the **request** object and also the **request handler** for delegating the creation of an HTTP response. Eventually, it was decided that the single pass was the best choice for several reasons that exist in the double pass approach:

- If an empty response is passed, there is no guarantee that a usable response will be returned. Additionally, because the middleware can change the response, there is no way to ensure it has not been written to, which could lead to many issues, primarily a corrupted body.
- While passing a response object is considered by many to ensure dependency inversion, injecting factories into the middleware would also solve that problem, ensuring the response object is safe.
- The **next** argument cannot be strongly typed as it's expected to be a callable or closure.

Now let's talk about the request handler, an object that is used to process a server request. The request handler works with the middleware before it processes the request. The `RequestHandlerInterface` in PSR-15⁷ appears very simple:

Listing 1.

```
1. namespace Psr\Http\Server;
2.
3. use Psr\Http\Message\ResponseInterface;
4. use Psr\Http\Message\ServerRequestInterface;
5.
6. interface RequestHandlerInterface
7. {
8.     public function handle(
9.         ServerRequestInterface $request
10.     ): ResponseInterface;
11. }
```

1 PSR-15: <https://www.php-fig.org/psr/psr-15/>

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

3 `http-server-handler`: <https://github.com/php-fig/http-server-handler>

4 `http-server-middleware`: <https://phpa.me/http-server-middleware>

5 Wikipedia: <https://en.wikipedia.org/wiki/Middleware>

6 PSR-15 metadocument: <https://www.php-fig.org/psr/psr-15/metadata>

7 PSR-15: <https://www.php-fig.org/psr/psr-15/#2-interfaces>



It's simple so that it doesn't force an opinionated design, allowing the implementer to make their own decisions on how it should handle a request. One might wonder, too, where is the middleware? This could be handled in many ways, more than we have room for in this article. Still, a list of middleware can be passed to the constructor, or perhaps a repository, that returns a list that might be filtered by conditions or criteria in the request. Following is an example based on an example in the PSR-15 metadocument:

Listing 2.

```
1. use Psr\Http\Message\ResponseInterface;
2. use Psr\Http\Message\ServerRequestInterface;
3. use Psr\Http\Server\MiddlewareInterface;
4. use Psr\Http\Server\RequestHandlerInterface;
5.
6. class RequestHandler implements RequestHandlerInterface
7. {
8.     public function __construct(
9.         protected MiddlewareInterface $middleware,
10.        protected RequestHandlerInterface $nextHandler)
11.    {}
12.
13.    public function handle(
14.        ServerRequestInterface $request
15.    ): ResponseInterface {
16.        return $this->middleware
17.            ->process($request, $this->nextHandler);
18.    }
19. }
20.
21. $initialHandler =
22.     new class() implements RequestHandlerInterface {
23.         public function __construct(
24.             protected ResponseFactory $responseFactory
25.         ) {}
26.
27.         public function handle(
28.             ServerRequestInterface $request
29.         ): ResponseInterface {
30.             // handle the request to return a
31.             // response using $responseFactory
32.         }
33.     };
34.
35. $initialResponse = new RequestHandler(
36.     new RoutingMiddleware(), $initialHandler
37. );
38.
39. $nextResponse = new RequestHandler(
40.     new AuthorizationMiddle(),
41.     $initialResponse
42. );
```

We have two request handlers here. The first requires middleware and another request handler (for *next*). The second is an anonymous class (it doesn't have to be anonymous, it could be a concrete class, too) that only accepts a `ResponseFactory`. We use the anonymous class to begin our chain of request handling (resulting in `$initialResponse`), which starts with the `RoutingMiddleware`. If the route is not found, then the response object will have a 404 status set on it and returned. Otherwise, with the route located, we instantiate a new request handler, inject the `AuthorizationMiddleware`, and the last response (`$initialResponse`). If access to the route is not authorized, we should return the response object with a proper 401 Unauthorized status or 403 Forbidden status (dependent on the application).

Conclusion

There are many ways that middleware can be implemented into an application. Modern frameworks usually provide architecture and support them and include built-in middleware, but they also allow for customizing them and adding custom middleware. Using middleware provides the developer with an excellent method for separating code and supporting greater flexibility. By following standards like PSR-15, your middleware code can be compatible and reusable in other applications.

Related Reading

- *PSR Pickup: PSR-7 HTTP Message Interface* by Frank Wallen, July 2022.
<https://phpa.me/psr-jul-2022>
- *PSR Pickup: PSR-6 Caching Interface* by Frank Wallen, September 2022.
<https://phpa.me/psr-sept-2022>
- *PSR Pickup: PSR-11: Container Interface* by Frank Wallen, November 2022.
<https://phpa.me/psr-nov-2022>
- *PSR Pickup: PSR-13: Link Definition Interfaces* by Frank Wallen, December 2022.
<https://phpa.me/psr-dec-2022>



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)

php[tek]

Early Bird Tickets Now on Sale



<https://tek.phparch.com>

May 16-18, 2023

Sheraton Suites Chicago O'Hare

Brought to you by the good people at





Events, Listeners, Jobs, and Queues Oh my!

Matt Lantz

Harnessing the power of Laravel's queues, events, listeners, observers, and jobs will significantly improve response times and provide your application with a clear separation of coding concerns enabling you to deliver some seriously efficient code.

In 2008 you could write a PHP application that did little more than a simple database transaction when handling a Request and providing a Response. However, as user experiences, datasets and expectations began to evolve, the demands within each Request did too. As PHP's complexities and capabilities grew, so did our users' expectations. Now, in a single request, you no longer update the database; you pull from external APIs, write audit logs, send emails, push to Slack and trigger a Latte order from Starbucks.

The sheer volume of expectations per Request often created slow response times and poor code architecture. Laravel overhauled this limitation by providing an elegant queue-based solution using Events, Listeners, Observers, and Jobs. By no means did Laravel pioneer this approach, but the answer it offered us was undoubtedly well beyond anything else the community had available. Harnessing the power of Laravel's queues, events, listeners, observers, and jobs will significantly improve response times and provide your application with a clear separation of coding concerns enabling you to deliver some seriously efficient code.

Events & Listeners

Events and listeners are not exclusive to Laravel or PHP; they are a fundamental component of many languages and frameworks. Laravel's implementation of its EventServiceProvider as a central manager and its ease of processing Listeners in its queue is what made its implementation of Events and Listeners so notable. Events are not queue-driven, and Listeners do not need to be either. Simply put, when you call an event, a listener is processed. The significant value in response times and code separation is that the queue workers can process the Listener.

The EventServiceProvider also enables you to have Events trigger multiple Listeners or have numerous Events initiate the same Listener.

In this example, we can clearly see that a new Latte is ordered every time a new user registers, and some notifications are sent as well.

```
protected $listen = [
    Registered::class => [
        SendEmailVerificationNotification::class,
        SendNewUserRegistrationNotification::class,
        OrderStarbucksLatte::class,
    ],
];
```

Since Events are class instances you call at any time in your application, Listeners is where we configure the use of the queue or persist with an in-sync action. You can also set which queue they are executed in, meaning you can specify if a Listener runs in the slow or fast queue.

```
public $queue = 'fast';
```

What are Observers Observing?

Ultimately Observers are just a particular way of listening for Model-based Events rather than making an Event and Listener per Model Event.

```
protected $observers = [
    Bookmark::class => [BookmarkObserver::class],
];
```

Configuring an Observer lets you unify all the actions related to Model Events into one file. This is great for applications where you need to perform actions surrounding creating or updating Model data sets.

Queues

Is this like a long line to get popcorn? No. Queues are a powerful tool for any developer. They enable you to defer code processing to a later time and sometimes to other resources. For example: Say we want to send an email to our users when they log in, or perhaps they generated a report, and we need to email them the results. Rather than delay the Request's Response and ruin the User Experience, we issue a Job or Event/Listener in the queue and have it processed in the background by a queue worker. The queue worker is a PHP command being run constantly in the background of your server; with Laravel Forge, this is handled through Supervisor. The queue worker's job is to monitor queue tasks coming in and process them.

When using queues, you need to be aware of some critical things. Unless you do some unique hacking to your Laravel app, queues that use Redis, Database, or Sync will always run in the order of First-In-First-Out (FIFO), and each of these engines also supports task delays. Delays are useful when you need to queue a task to be run, say 24 hours after an action in your application. However, be cautious when exploring the use of Cloud provider tools like SQS as they can be non-sequential due to their architecture. Though, recently AWS did provide SQS with a FIFO option. Though again, because we're



mindful of Cloud provider options, SQS limits your delays to a maximum of 15 minutes, and FIFO tasks cannot be delayed at all.

Any time a task fails in the queue, it is logged in the `failed_jobs` table, which is added to your app with the command `$ artisan queue:failed-table`. Having run this migration, you can run commands to rerun single tasks or the whole stack of failed tasks. You can also quickly review the corresponding trace log of why a task failed in the first place.

If you're focused on Redis as your queue engine, look at Horizon, another Laravel product that lets you manage your Redis-based queues directly from your app config file. Otherwise, you're likely creating queue workers on your server to run; if you use Laravel Forge, these workers (in the Queue section) are configured with a supervisor config, so the workers run even if your server crashes and reboots. Be mindful that the number of processes is the number of threads you wish to run for a queue worker. If you have a consistently large number of jobs in your queue, you could quickly suck all your server's power to handle queue-related tasks vs. actual web requests.

Jobs

Jobs are, for lack of a better term, Events and Listeners all in one. Rather than creating an event and the corresponding Listener for a single action, you can create a job. For example, `GenerateReport` is a great job, which we can easily call by writing something similar to this:

```
GenerateReport::dispatch($bookmark);
```

Where our actual `GenerateReport` job, which is configured to run the queue, looks like Listing 1.

There is only a single moment the user can generate a report, so there are not multiple Events triggering the Listener; we also don't need to have numerous Listeners on the single generated report Event. Instead, we create a single job that is entered into the queue and processed. Jobs can be configured to run without the queue and can be set to run immediately in an application that enables them to perform like Actions. However, doing so will not improve response times, though it can sometimes be helpful when debugging in your local environment.

In short, Laravel provides an incredible set of tools for handling complex code requests in an elegant responsive manner. Should your users request complex database reports or query third-party APIs, the use of events, listeners, jobs, and queues can dramatically improve your user's experience. At the very least, utilizing these tools can ensure your users are not waiting 45 seconds for a Request's Response. In most cases, you can initially implement Observers and Jobs in your application and phase in Events and Listeners as complexity or repetition grows within your application. Similarly, with queues, the database engine will often satisfy nearly all your queue needs. But, if your application's complexity is scaling,

as is its user base, then shifting to Redis can greatly improve your application's overall performance without significantly increasing its maintenance.



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

Listing 1.

```
1. <?php
2.
3. namespace App\Jobs;
4.
5. use App\Models\Bookmark;
6. use Illuminate\Bus\Queueable;
7. use Illuminate\Queue\SerializesModels;
8. use Illuminate\Queue\InteractsWithQueue;
9. use Illuminate\Contracts\Queue\ShouldQueue;
10. use Illuminate\Foundation\Bus\Dispatchable;
11.
12. class GenerateReport implements ShouldQueue
13. {
14.     use Dispatchable;
15.     use InteractsWithQueue;
16.     use Queueable;
17.     use SerializesModels;
18.
19.     public function __construct(
20.         public Bookmark $bookmark
21.     ) {}
22.
23.     public function handle()
24.     {
25.         // do something here
26.     }
27. }
```



Making the Cut

Beth Tucker Long

Between a major shortage of programmers, rampant burnout, and a workforce disrupted by the pandemic, employment in the tech industry is in turmoil. At the same time, our hiring practices continue on as if nothing has changed. Why are we letting this happen?

International Data Corporation (IDC), which provides market research on the technology field, released a market perspective document in September of 2021 titled “Quantifying the Worldwide Shortage of Full-Time Developers” (<https://phpa.me/idc>). Their research shows that “the global shortage of full-time developers will increase from 1.4 million in 2021 to 4.0 million in 2025”. This is a major shortage.

The U.S. Bureau of Labor Statistics predicts, “Overall employment of software developers, quality assurance analysts, and testers is projected to grow 25 percent from 2021 to 2031, much faster than the average for all occupations.” This is really fast growth.

The tech industry is starting from a major shortage of workers and growing at a much faster-than-average rate. We are already working our existing workforce too hard, leading to high rates of burnout in our field. This is not sustainable, and it's getting worse. We need to make some drastic changes, and I believe this can start with the hiring process.

Right now, job listings ask for professional experience in every language the company has ever considered using. If applicants can get beyond that gate and land an interview, they will be faced with intensive logic puzzles, highly-academic coding exercises, and build-at-home development projects. This is intimidating, exhausting, and time-consuming. It also, in my opinion, completely misses the mark for what we really need.

These technical interview practices will find you someone good at memorizing syntax, someone who has a lot of practice coding in an academic environment, and someone who has the time (and resources) in their off hours to afford coding for you for free. These are not the skills that define a good developer.

A good developer is excited about solving problems and thinking creatively, so they can navigate whatever new technologies and challenges come their way. A good developer knows where to quickly find the syntax specifications they need to know because things change too quickly in our industry to memorize everything. A good developer has a solid work-life balance, so they won't burn out. A good developer has good listening and communication skills to work with clients and teammates effectively. We need to start looking for these skills when hiring.

Instead of giving them an obscure riddle to solve, set up a mock meeting with someone in the marketing department and figure out the best way to create an application for their

next campaign. Listening to the project requirements, asking questions to see if they are the right requirements, and then choosing the best way to meet those requirements are absolutely critical skills for a good developer.

Instead of asking them to traverse a binary tree, ask them something practical, like how they would validate and secure a contact form or set up a blog for rapid scaling.

Instead of asking them to build you a useless project for free at home, hire them for a few hours to code for an open-source project that you use. They are compensated for their work, and even if you don't hire them, you benefit from the work they have done on the open-source project.

Above all else, find out how they handle stressful situations and conflict with co-workers. It's easy to teach someone syntax. It's really hard to teach them to work well with others. These are the skills we need to focus on—the day-to-day practical rather than the high-level academic.

And please, everyone, start hiring junior devs. They are tomorrow's senior devs if we can just get them started today.

Related URLs:

- “Quantifying the Worldwide Shortage of Full-Time Developers” (IDC) - <https://phpa.me/idc>
- “Occupational Outlook Handbook: Software Developers, Quality Assurance Analysts, and Testers” (U.S. Bureau of Labor Statistics) - <https://phpa.me/labor-stats>
- “What You Need to Know to Ace Your Technical Interview” (Glassdoor) - <https://www.glassdoor.com/blog/technical-interview-tips/>
- “The technical interview practice gap, and how it keeps underrepresented groups out of software engineering” (interviewing.io) - <https://interviewing.io/blog/technical-interview-practice-gap>



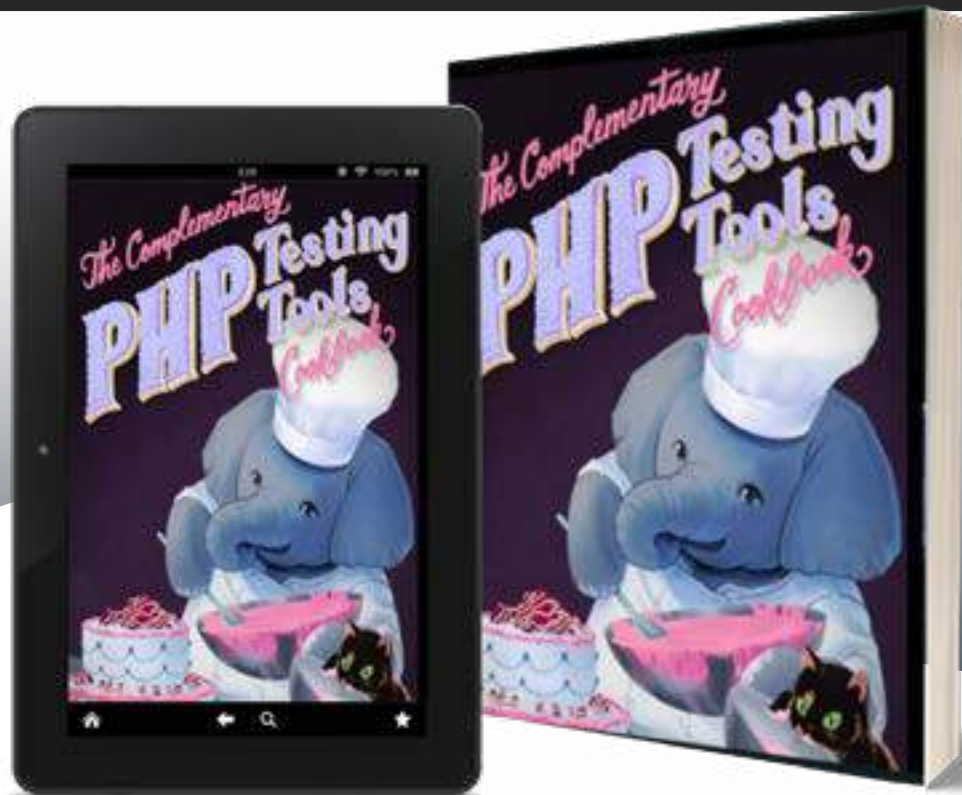
Beth Tucker Long is a developer and owner at Treeline Design¹, a web development company, and runs Exploricon², a gaming convention, along with her husband, Chris. She leads the Madison Web Design & Development³ and Full Stack Madison⁴ user groups. You can find her on her blog (<http://www.alittleofboth.com>)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

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

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

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

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

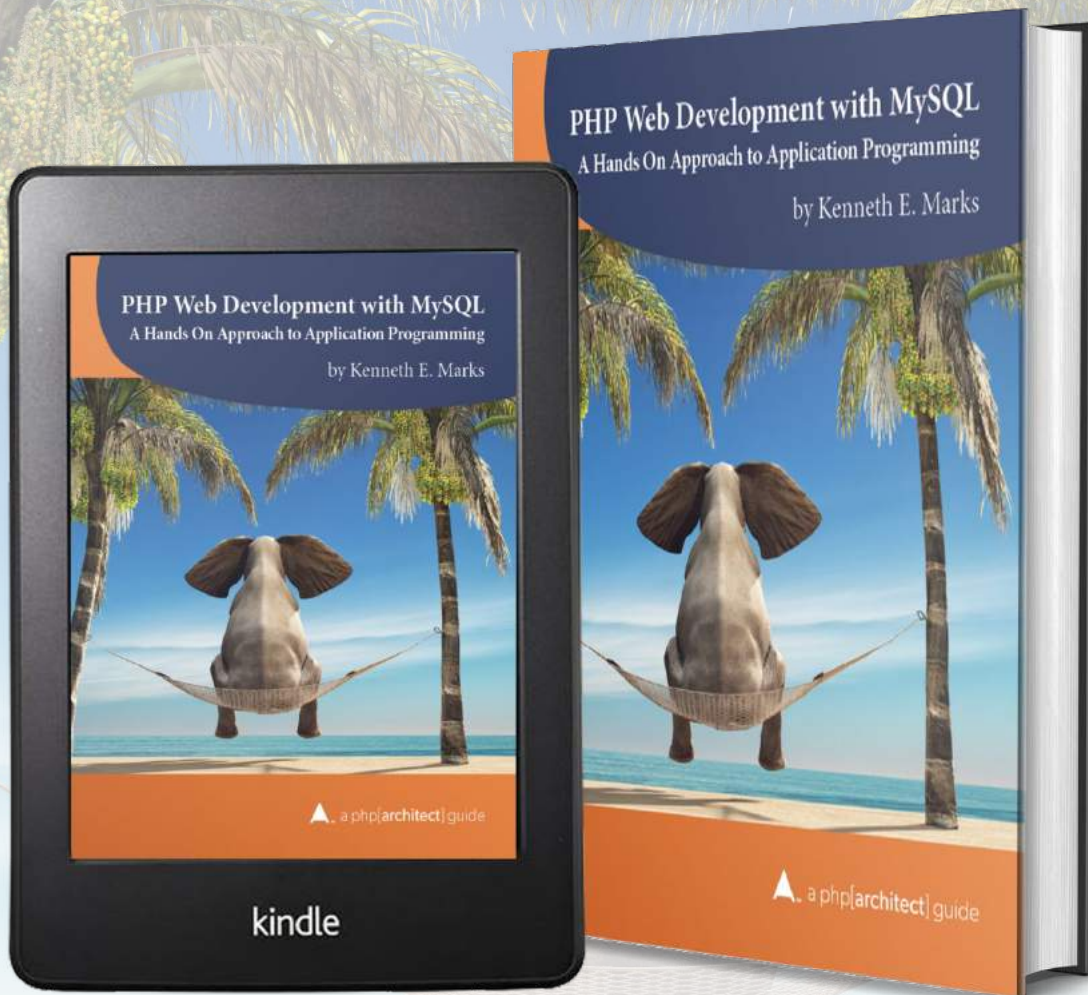


Learn how a Grumpy Programmer approaches improving his own codebase, including all of the tools used and why.

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

Available in Print+Digital and Digital Editions.

Order Your Copy
phpa.me/grumpy-cookbook



Learn how to build dynamic and secure websites.

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

Available in Print+Digital and Digital Editions.

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