www.phparch.com

September 2023
Volume 22 - Issue 9

php[architect]

The Magazine For PHP Professionals

The Spectrum of PHP

Getting Modern with our Monolith

Asynchronous PHP Without Libraries

An Overnight COVID Ticketing System

ALSO INSIDE

Education Station:
We are Losing the Browser War

PHP Puzzles:
Comb Sorting

Security Corner:
The Apocolypse is Now

DDD Alley:
Simple Queue System

Barrier-Free Bytes:
Unseeable Colors

PSR Pickup:
Exploring Real World Applications

Artisan Way:
Popular Tools for Robust Laravel Development

Finally□□:
It□ Time to Reinvent the Wheel

Honeybadger provides all the context you need to understand what is causing an exception

Honeybadger.io

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.

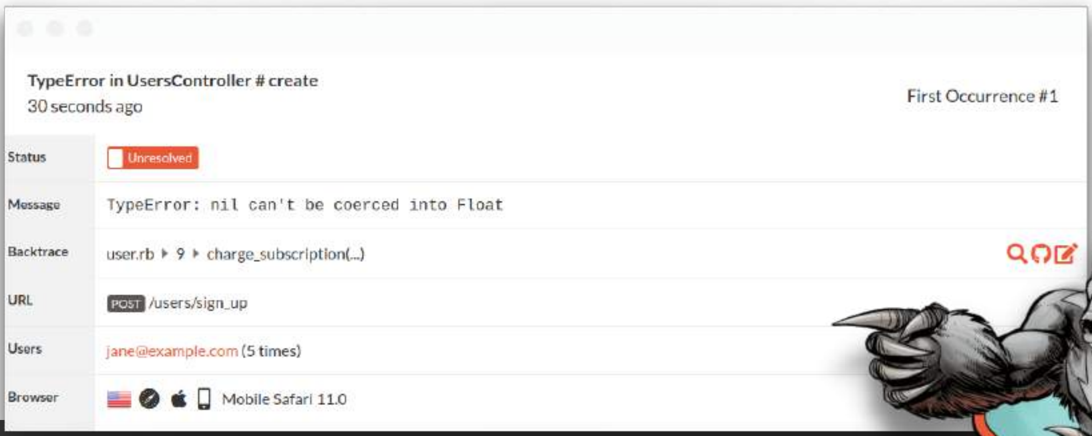
DevOps monitoring, for developers. *gasp!*

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.



TypeError in UsersController # create
30 seconds ago

First Occurrence #1

Status	Unresolved
Message	TypeError: nil can't be coerced into Float
Backtrace	user.rb ▸ 9 ▸ charge_subscription(...)
URL	POST /users/sign_up
Users	jane@example.com (5 times)
Browser	Mobile Safari 11.0

Start Your Free Trial Today

https://www.honeybadger.io/

SEPTEMBER 2023
Volume 22 - Issue 9

Edited way too late in the month

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

An Ever Changing System

When it comes to PHP development, one thing remains clear: we are living in a time of exciting changes and rapid innovation. The constant tug-of-war between modernization and classic approaches, the necessity for speed without sacrificing quality, and the search for optimal tools are the battles PHP developers face daily. Just as we marvel at the sheer diversity of libraries and frameworks available to us, we are equally terrified of the growing threats from AI and the shifting landscape. This month, we dive into some of this and much more.

To start things off, Steve McDougall is back with *Getting Modern with Our Monolith*. Steve discusses the marriage of front-end and back-end code to create modern monolithic applications using InertiaJS, VueJS, and Laravel.

Next, Vinicius Dias brings us *Asynchronous PHP without external libraries*, where he demystifies asynchronous programming in PHP and how to achieve it without external libraries. In a bonus third feature article, Raja Renga shares *An overnight Covid Ticketing System for Bangalore Metropolitan of India with Fibenis,* a compelling narrative that focuses on the urgent development of a Covid Ticketing System for the Bangalore Metropolitan area during India's second Covid wave in 2021.

In Frank Wallen's PSR Pickup column, he brings us *Exploring Real-World Applications of PHP-FIG's PSRs through Popular Libraries*. He explores a package built using the guidelines of PHP-FIG's PSRs (PHP Standards Recommendations).

Edward Barnard continues his series, *Create Observability, by bringing us Part 4, Simply Queue System* over in DDD Alley. Ed will walk you through creating a simple queue system based on MySQL tables. And in keeping with making things observable, you will also see how to create a flow chart documenting your work.

Maxwell Ivey discusses the frustrations of online shopping, focusing on the importance of accessibility and the need for more inclusive web designs in this month's Barrier-Free Bytes column, *Unseeable Colors*.

Matt Lantz speaks highly of the Laravel community for consistently pushing the envelope on what can be achieved in Artisan Way's column *Popular Tools for Robust Laravel Development*. Matt discusses the importance of frameworks and how they help overall development. He will share various paid products and CLI tools that may help with your entire project or just be useful to resolve simple issues.

In Security Corner's column, *The Apocalypse is Now*, Eric Mann probes the ominous but frequently misunderstood concept of an AI Apocalypse, shedding light on the state of AI and its current threat level, which is considerably less dramatic than what Hollywood portrays but still significant.

Netscape Navigator, Internet Explorer, Mozilla Firebird (or later renamed, Firefox), and of course Chrome. We all know them and have used them at some point. In Education Station, Chris Tankersly warns, *We are Losing the Browser War*, where he outlines the concerning trend of browser monoculture, with Google Chrome now dominating the scene. We'll also learn more about how browsers shape our entire experience with the internet. Chris further expands on the effects of various browsers on developers as well.

Oscar Merida continues to help sharpen our minds and critical thinking with *Comb Sort* in PHP Puzzles. Last month, Oscar discussed an easy-to-understand algorithm called Bubble Sort. We were left with the challenge of writing an implementation of the Comb Sort. You'll want to see how Oscar breaks down the differences between the Bubble and Comb Sort and which one performed better overall.

Finally, Beth Tucker-Long shares her journey through the life cycle of a monolithic framework that has become too cumbersome to manage in part because of the bulk added by frameworks and packages. She thinks *It's Time to Reinvent the Wheel.*

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 announcments. Make sure you're connected with us.

- Subscribe to our list: https://phpa.me/sub-to-updates
- Mastodon: @editor@phparch.social
- Twitter: @phparch
- Facebook: http://facebook.com/phparch

Download the Code Archive:

https://phpa.me/September2023_code

Getting Modern with Our Monolith

Steve McDougall

Figure 1.

In June 2021, we heard a rumor about how we could merge our cobbled-together front and backend code and merge it into something beautiful. Something that we call modern. Something we could be proud to work on. Fast forward to today, and we are getting closer to that dream. I am, of course, talking about InertiaJS.

We may have yet to reach a stable 1.0.0 release, but hundreds, if not thousands, of applications already use this in production. We can now stop trying to load a SPA application as soon as Laravel has served its first route, and instead, we can build genuinely modern monolithic applications where both frontend and backend code are as one.

I have always been a fan of using VueJS on the frontend of my applications. It's simplicity and beautiful syntax made me feel like I could achieve what I needed to - while having a familiar feeling of what Laravel gave me as a backend framework. Since alpha-rc1, I have been trying to find a way to use this fantastic library to manage all of my frontend code with my Laravel applications to have the best of both worlds.

At some point, however, NuxtJS came along and stole my heart. I could build a complete SPA application that lived separately from my back-end code - and was connected through an API - another topic I love very much. While a perfect solution, this felt like extra steps to get what I wanted - no matter the cost. Nuxt was, and still is, a fantastic framework for building large VueJS applications. But we have all suffered the overload of maintaining two repos: one front and one back, while trying to synchronise releases and keep momentum equal.

InertiaJS came about at the right time. I was about to swear off JavaScript for good - which some might say is a good thing, and others might say I shouldn't be allowed to write it anyway. It was the promise we were all hoping for, the hero we didn't know we needed. All we needed to do was install two packages, and we had the ultimate monolith that would work exactly how the tech gods intended.

For those unaware of Inertia, it is a middleware application. Nothing more, nothing less. It sits between your Laravel application and whichever front-end framework you choose. It allows you to render components on the front end using JavaScript magic, just like you would in your NextJS, NuxtJS, or even SvelteKit application. Then, as a request comes in to go to a new route, it will fetch the new component requested over XHR, of course, re-render the DOM, and push the browser's history API to the new location.

It is, and I believe this, how modern JavaScript applications should be built. There is much to say about the plethora of services a company like Vercel can launch to make your full-stack JavaScript application fantastic. But often, we want a solid frontend framework to talk to a mature and easy-to-use ecosystem on the backend to extend our complexity in how we choose. The biggest problem with full-stack JavaScript, besides the severe lack of elephants, is that it takes the approach of "everything is a SaaS." This makes building anything a series of subscriptions, connections, and widgets … we have all seen where this leads before. Early 2000s WordPress/Joomla/insert the long lost forgotten CMS here.

Back to Inertia, however, as I am sure you don't want to hear my rants about the JavaScript ecosystem. Or do you?

Inertia was originally the brainchild of Jonathan Reinink, who now works full-time at Tailwind Labs and is maintained mainly by the community these days. But that shouldn't stop you, especially in the true spirit of open source.

Let's look at how quickly, especially if we utilize the entire Laravel ecosystem, we can get a full-stack modern monolith up and running.

Firstly, let's install Laravel using the Laravel Installer.

```
laravel new github-browser --git --pest
```

I always use this approach: initialize a standard Laravel project with version control and pestPHP ready to go. From here, I can track the changes as I start adding layers of the Laravel ecosystem - starting with authentication.

```
composer require --dev laravel/breeze
```

I like to do this separately from the install step because by doing so, it has a template for VueJS with TypeScript already set up for me. For those who don't know me, I hate TypeScript but love types - it is a contradiction, I know. This will boilerplate an InertiaJS project for you - with VueJS and set everything up to work flawlessly - even Server Side Rendering for your VueJS code if you want to!

The key differences you will see from a standard Laravel application are mostly kept in the `/resources/js` directory - which we'll get to. But you will have an additional view file, now called `app.blade.php`, in the root of your views directory. Inertia uses this as the HTML/PHP template to load everything it needs. You can extend or customize this as much as you need to - as long as you leave the `@routes` and `@inertia` tags in.

You will have a nice, shiny new middleware class called `HandlesInertiaRequests`, which is added as global middleware to all your registered `web` routes. This is an important file, as it is how you register global props for all of your frontend pages. These are merged into any passed-through props - allowing you to have a set of persistent properties you always share.

Typically, in this file, I would share things like the authenticated user - and any "meta" models you might want to have available. A "meta" model is something like countries - that you might have in a dropdown and be database managed. I typically cache this heavily in this middleware so that I am not performing the same query on every request. Let's look at an example.

The demo application I will build is a GitHub Issue Browser that uses the GitHub REST API to allow me to manage issues and notifications in a way that works for me.

Listing 1.

```php
public function share(Request $request): array
{
  $auth = Auth::check();

  return array_merge(parent::share($request), [
    'auth' => [
      'user' => $auth ?
        new UserResource(
          resource: Auth::user(),
        ) : null,
      ],
      'projects' => $auth ?
        ProjectResource::collection(
          resource: Cache::remember(
            key: Auth::id() . '-projects',
            ttl: CacheTime::HOUR->value * 5,
            callback: static fn () =>
              Project::query()
                ->where('user_id', Auth::id())->get(),
          )
        ) : null,
      'ziggy' => function () use ($request) {
        return array_merge((new Ziggy)->toArray(), [
          'location' => $request->URL(),
        ]);
      },
  ]);
}
```

In Listing 1 we are checking the authenticated state on each request - to conditionally load the required data. Because this is called on each component render - a session could have been invalidated by the time the next request comes through. So, it makes sense to check this. Then, we conditionally load the projects for the authenticated user. I like to use Laravels API Resources for passing data back to Inertia - this allows me to do any backend processing that I might need or want to do, keeping things consistent. I am caching the projects for 5 hours using a PHP 8.2 Enum that I typically use for common TTLs - which is a nice little developer experience add.

This is the main change that you will see in your Inertia Application, a middleware that handles passing data. Quite simple compared to the old SPA or NuxtJS way, am I right? Controllers are another place where you will see a difference. Instead of seeing the `view` helper or something similar, you will typically see something like:

```php
public function __invoke(Request $request): Response
{
    return Inertia::render('PageName/Component');
}
```

This uses the Inertia Facade to load the response factory to return the component with any passed-in props in a format the frontend Inertia helper will understand.

Now, I consider myself to be a somewhat picky developer. I have strong opinions that I have formulated over time, especially when it comes to writing Laravel code. With that being said, don't take my word as gospel; what works for me may not work for you.

In every Inertia application I build, I create a trait allowing me to load and return Inertia responses in the way that works for me.

```php
trait HasInertiaResponse
{
    public function __construct(
        protected readonly ResponseFactory $response,
    ) {}
}
```

This pulls the Response Factory from Inertia into the constructor. My main reasoning for this is that the helpers and facade will do a look-up in the container - to load the build instance of the Inertia Response Factory. This feels like wasted effort when you know you will need it, so I might as well use my DI container properly and inject it into the constructor.

From here, I can refactor my controller as shown in Listing 2.

There are times that I may want to perform additional operations and pull additional things into the constructor, but a lot of the time - I stick to this pattern. It keeps my code clean and predictable. If we create a new resource and submit a form, our controller will look exactly the same as if we were using standard Laravel without Inertia.

Listing 2.

```php
final class IndexController
{
  use HasInertiaResponse;

  public function __invoke(Request $request): Response
  {
    return $this->response
                ->render('PageName/Component');
  }
}
```

Speaking of forms, let's look into how they work in Inertia. The documentation for this doesn't quite do it justice, but it's easily the most powerful feature in Inertia right now. For these examples, I will use VueJS in inertia, as shown in Listing 3, so that you can see an example, but they work similarly, no matter the frontend framework.

Listing 3.

```
<script setup lang="ts">
import { useForm } from '@inertiajs/inertia-vue3'

const form = useForm({
  email: '',
  password: ''
})

const submit = async () => {
  form.post(route('login'), {
    preserveState: false,
    preserveScroll: true
  })
}
</script>
```

The code above uses the form helper from Inertia, allowing us to quickly and easily create a reactive form. From here we can use `v-model` on input elements to have the two way binding. We add `v-on:submit.prevent="submit"` on our form element to submit our form to a specified route asynchronously. We have two main options I like to use here: 1) Preserve state will preserve the forms state after it has finished - so the email and password won't be cleared. 2) Then we have `preserveScroll`, which is great when you have a form part way down your page - and don't want the page to scroll back to the top after submission.

I have tried to replicate the same user and developer experience combining various solutions in the past, and nothing quite hits the mark the way that InertiaJS does. It brings the developer experience of VueJS to the developer experience of Laravel in a way that works really well - the much needed glue to hold it all together.

When working with responses in Inertia, you have a number of options available to you. You can respond directly with the data you want to return, which will **always** be included on the first visit. It will **optionally** be included on partial reloads and will **always** be evaluated as shown in Listing 4.

Listing 4.

```php
final class IndexController
{
  use HasInertiaResponse;

  public function __invoke(Request $request): Response
  {
    return $this->response->render(
      component: 'PageName/Component',
      props: [
        'articles' => Article::query()->all(),
      ],
    );
  }
}
```

You can take this a step further so that the returning data is **always** included on the first visit, **optionally** included on partial reloads, and will **only** be evaluated when needed, see Listing 5.

Listing 5.

```php
final class IndexController
{
  use HasInertiaResponse;

  public function __invoke(Request $request): Response
  {
    return $this->response->render(
      component: 'PageName/Component',
      props: [
        'articles' => fn () => Article::query()->all(),
      ],
    );
  }
}
```

The final option is the ultimate lazy loading approach. Response data will **never** be loaded on the first visit, **optionally** included on partial reloads, and **only** evaluated when needed as shown in Listing 6 on the next page.

The beauty of these approaches is that they enable you to implement your own state management pattern on the VueJS side. You can implement loading states and skeleton UI components to speed up the loading of components - and backfill the data when you need it. The way you might do this is shown in Listing 7 on the next page.

Listing 6.

```
final class IndexController
{
  use HasInertiaResponse;

  public function __invoke(Request $request): Response
  {
    return $this->response->render(
      component: 'PageName/Component',
      props: [
        'articles' => $this->response->lazy(
          callback: fn () => Article::query()->all(),
        ),
      ],
    );
  }
}
```

Listing 7.

```
<script setup lang="ts">
import { ref, onMounted } from 'vue'
import { Inertia } from '@inertiajs/inertia'

const loaded = ref(false)

onMounted(() => {
   Inertia.reload();
   loaded.value = true;
})
</script>
```

Listing 8.

```
<script setup lang="ts">
import { useRemember } from '@inertiajs/inertia-vue3'

const form = useRemember({
  title: '',
  name: '',
  email: '',
  more_fields: ''
}, 'optional-unique-id-if-mult-components-use-it')
</script>
```

Listing 9.

```
it('renders timeline component', function (): void {
  get(
    uri: action(InvokableController::class),
  )->assertOk()
   ->assertInertia(
     fn (AssertableInertia $page) =>
       $page->component('Timeline/View')
            ->has('posts',
                  15,
                  fn (AssertableInertia $page) =>
                    $page->where('topic', 'Foo Bar')
                         ->etc()
            )
   );
});
```

This will call a partial reload on the page once the component mounts; it will call an inertia reload method to lazy fetch the data from your controller. Then, update the loaded reference to true so you can switch your skeleton UI to your actual UI. This JavaScript may not be 100% perfect, and there could be more adequate ways to achieve this - after all, I am a backend engineer at heart.

There is also a handy method shown in Listing 8 that you can use called `rememberState`, which will allow you to remember the components state within the history API. So, if a user navigates away and comes back, the state will be preserved. This is a great way to keep form state in case of accidental user navigation. I think we can all agree we have used super long and frustrating forms, then accidentally navigated away, only to have to go through the whole painful process all over again. Perhaps not something you would use often, but it's definitely a great feature to remember.

Alongside all of these handy features (Example shown in Listing 9), the testing helpers that are available really make building applications in Inertia such a pleasant experience - especially when paired with something such as pestPHP.

Of course you can choose your own adventure when it comes to building a full-stack Laravel application, but if you do fancy it - why not give Inertia a try? I have used *all of the options* so far, and my favorite experience has been building an Inertia application. Livewire is fantastic, and there is no doubt about it. However, I find it is best used sparingly in your application where you need reactive components - instead of going all in with reactive pages. This is just a personal opinion, though - you may have better luck. I will point out that if you are building internal dashboards, don't try anything other than Filament. It is a beautifully written package that will enable you to start building admin interfaces quickly using Livewire.

Steve McDougall is a conference speaker, technical writer, and YouTube livestreamer. During the day he works on building API tools for Treblle, and in the evenings spends most of his time writing content, or contributing to the PHP open source community. Whatever you do, don't ask him his opinion on twitter/X `@JustSteveKing` *@JustSteveKing*

Asynchronous PHP without external libraries

Vinicius Dias

Asynchronous programming can be daunting at first, especially in PHP, but we will understand how to achieve it without using any external libraries in PHP. This will make the world of non-blocking I/O much clearer and prepare you to use tools like ReactPHP or Swoole with a greater understanding of what is happening under the hood.

Performing I/O operations, such as accessing streams by reading and writing to files or making network calls, is a relatively common task for PHP devs. This type of operation is costly and takes time to execute. When dealing with multiple I/O operations, an effective technique to enhance application performance is taking advantage of asynchronous execution.

Stream Access

Before delving into techniques and functions for asynchronous programming, let's understand how we usually handle I/O.

Accessing files, HTTP requests, sockets, and more can be done using PHP streams[1]. A stream is essentially the representation of a flow of data. When we use functions like `file_get_contents`, `fopen`, `fgets`, and others, we're working with PHP streams, and we can call those functions to handle a lot more than just plain old text files.

The function call `file_get_contents('<https://example.com>')` will perform a valid HTTP request, and the return will be the response body. By using stream contexts, it is even possible to send request body or headers using the same function, even though it has `file` on its name. But this is **not** a text about PHP streams, so let's focus on the topic of this article: asynchronous programming.

Asynchronous Programming

There are multiple definitions for "Asynchronous Programming", which can mean completely different practices and techniques. Using a message queue can be considered asynchronous programming, but that's also not what we will discuss here. So, let's get some clear definitions and decide on what we mean by asynchronous programming in this text.

Whenever asynchronous programming is mentioned, it means non-blocking I/O, e.g., the possibility of handling messages when it's possible instead of immediately.

This definition seems quite vague and not clear at all, right? Let's try to use some practical examples using a common language in the web context other than PHP: JS. The HTTP request will be performed when we use the `fetch` function in JavaScript, but the response won't be handled immediately. The browser JS engine will execute the code that treats this response as soon as possible, but it will not be synchronous or immediate.

What does that mean? When we receive the response, the browser JS engine will put the function that handles the response in some sort of line/queue, and as soon as possible, that function will be executed. While the response isn't available or even after it arrives (but before it's time for that function to be executed), other pieces of code can be run.

In the following example, we use Promises to inform what function will be added in line for when the response is ready. Up until that moment, other lines of code will be executed.

```
fetch('<https://example.com>')
  .then(function (res) {
    // This function can handle the response
  });

// Here, other lines of code can be executed even
// before the function that handles the response
console.log("Other functions");
```

This "line of functions" and delegating the execution for the future is possible thanks to a pattern called *Event Loop*. Using an event loop allows you to have asynchronous programming, or a less confusing term, non-blocking I/O.

1 https://php.net/streams

Events

The name *Event Loop* brings up something that we use extensively in web development, especially in the front-end: events. We use event-driven development to respond to actions like button clicks or text input in JavaScript. Events can occur at any time, making their nature asynchronous. But can we achieve something similar on the PHP side?

Before diving into PHP itself, it's worth noting that the operating system already deals with asynchronous calls. Many events occur during system execution, and there's code reacting to these events all the time. Using PHP, we can access some of these operating system functionalities.

The Stream_select Function

There is a function not widely known by PHP developers called stream_select[2]. This function allows us to "observe" changes that may occur in streams. It's not the easiest function to grasp, so I will try my best to make it less painful to understand in the following chapters.

Scenario

Imagine you need to read the contents of 5 different files and then perform some processing. Traditionally in PHP, you might do something like: (See Listing 1)

Listing 1.

```php
<?php

$fileContent1 = file_get_contents('file1.txt');
$fileContent2 = file_get_contents('file2.txt');
$fileContent3 = file_get_contents('file3.txt');
$fileContent4 = file_get_contents('file4.txt');
$fileContent5 = file_get_contents('file5.txt');

// Process the 5 files
```

The problem here is clear: Before reading `file2.txt`, you need to finish reading the whole `file1.txt`. While the computer waits, you could already be reading and processing the other files individually.

When performing I/O operations (file access, network, etc.) synchronously, the CPU remains in a state called **idle** while the operation is pending. That's something we want to avoid. We want to ensure that the processor keeps working while the file is being prepared for reading or while some HTTP request is being performed.

Asynchronous Solution

To address this situation, we can use the `stream_select` function, which monitors changes in the status of a list of streams. In other words, when any of the files are ready for reading, this function notifies us. It's important to note that the operation of reading each file does not necessarily happen in order. The operating system might open `file3.txt` faster than `file1.txt`, for example. In this scenario, we process it right away without waiting for the previous files.

Listing 2 contains the full code for asynchronously reading the files and dealing with each file individually when they are ready to be read.

Listing 2.

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

$fileStreamList = [
    fopen(__DIR__ . '/text_files/file1.txt', 'r'),
    fopen(__DIR__ . '/text_files/file2.txt', 'r'),
    fopen(__DIR__ . '/text_files/file3.txt', 'r'),
    fopen(__DIR__ . '/text_files/file4.txt', 'r'),
    fopen(__DIR__ . '/text_files/file5.txt', 'r'),
];

foreach ($fileStreamList as $fileStream) {
    stream_set_blocking($fileStream, false);
}

do {
    $streamsToRead = $fileStreamList;
    $streamsWithUpdates = stream_select(
        $streamsToRead,
        $write,
        $except,
        seconds: 1,
        microseconds: 0
    );

    if ($streamsWithUpdates === false) {
        echo 'Unexpected error';
        exit(1);
    }

    if ($streamsWithUpdates === 0) {
        continue;
    }

    foreach ($streamsToRead as $index => $fileStream) {
        $content = stream_get_contents($fileStream);
        // process file content
        echo $content . PHP_EOL;
        if (feof($fileStream)) {
            fclose($fileStream);
            unset($fileStreamList[$index]);
        }
    }
} while (!empty($fileStreamList));
```

Explanation

Setup

As mentioned, it's not straightforward, so let's break it down. First, we open all the files we want to read using `fopen`. So far, so good. Then, we use the `stream_set_blocking` function to

2 *https://php.net/stream_select*

inform that each stream should be handled in non-blocking mode. This way, reading the files will not block the CPU, i.e., it will prevent it from waiting for the files to be ready for reading.

At this point, we have an array with 5 open resources, ready for stream operations. Let's take a look at a snippet of code:

Listing 3.

```
$fileStreamList = [
    fopen('file1.txt', 'r'),
    fopen('file2.txt', 'r'),
    // ...
];
foreach ($fileStreamList as $fileStream) {
    stream_set_blocking($fileStream, false);
}
```

(See Listing 3)

If we would `var_dump` the contents of `$fileStreamList`, the output would be something like the following:

```
array(5) {
  [0]=>
  resource(5) of type (stream)
  [1]=>
  resource(6) of type (stream)
  ...
}
```

Now, the challenging part begins. The call to the `stream_select` function must be within a loop since some streams may not be ready for reading immediately.

The Stream_select Function Parameters

Here, we enter one of the most important details: the `stream_select` function parameters. The first three parameters of this function are passed by reference. That's why the second and third parameters are variables we haven't defined yet. The first parameter is the list of streams to observe for reading. The second is for writing, and the third one, less common, is for exceptional data with higher priority. The last two parameters are about timeout.

The number `1` passed as a parameter indicates the number of seconds the function should wait for notifications of changes in the streams before "giving up", i.e., the timeout. This is why it needs to be within a loop. If this time passes and no updates arrive, we try again. The function returns if any (not necessarily all) streams are ready before that 1 second. The last parameter, which we set as 0, indicates the timeout in microseconds.

Return and Reading

Now, let's discuss the return value. This function returns the number of streams that have updates. In our case, it tells us how many files are ready for reading. However, in case of an error, the function returns `false`.

Additionally, since the parameters are passed by reference, they can be modified by the function. That is the reason why a copy of the original list of streams was made in the following line: `$filesToRead = $fileStreamList;`. After the function call, the variable we passed as a parameter will contain only the streams with updates, i.e., in our case, the files that are ready for reading.

To illustrate, let's assume that all the files are available in the first loop iteration. In this case, by analyzing the variable values, we can understand what's happening. Looking at the value of `$streamsWithUpdates`, it would be int(5) in this scenario.

Looking at both arrays of streams (`$fileStreamList` and `$streamsToRead`), they would be the same:

`$fileStreamList`:

```
array(5) {
    [0]=>
    resource(5) of type (stream)
    [1]=>
    resource(6) of type (stream)
    ...
}
```

`$streamsToRead`:

```
array(5) {
    [0]=>
    resource(5) of type (stream)
    [1]=>
    resource(6) of type (stream)
    ...
}
```

Now, if in the first iteration, only 3 out of 5 files are ready, and we analyze the values of the three variables, we'd have something like:

`$streamsWithUpdates`:

```
int(3)
```

`$fileStreamList`:

```
array(5) {
    [0]=>
    resource(5) of type (stream)
    [1]=>
    resource(6) of type (stream)
    ...
}
```

`$streamsToRead`:

```
array(3) {
    [0]=>
    resource(5) of type (stream)
    [2]=>
    resource(7) of type (stream)
    [4]=>
    resource(9) of type (stream)
}
```

Note the sizes of both arrays. The first array (`$fileStreamList`) is the original stream list with 5 elements, while the second one (`$streamsToRead`) contains only the 3 files that are ready for reading.

This way, we can iterate through this array knowing there is data to read and process. However, there's no guarantee that the entire file content will be there, so we should check whether the end of the file was reached by using `feof`. For small files, typically, all content arrives at once, but don't rely on this and prepare your code to read even just a single byte at a time.

After reading the entire file, we must close it. Another important detail is removing the file we just closed from the original list of streams we want to read. This is done so that in case of a new iteration in the loop, i.e., if not all files have been read, we don't attempt to reread the files that have already been read and closed.

With this approach, we have our exit condition for the loop. When all files have been removed, and the list is empty, we know that all the files have been processed.

Concepts

This code can certainly be refactored to make it more readable, but I believe the logic behind it is clear. In this article, we have created an event loop rudimentarily. Of course, this implementation isn't complete or optimized, but it serves our purpose: reading files as they become ready for reading. This way, we ask the operating system to prepare all the files for reading at once.

The advantages of this asynchronous approach become more apparent when dealing with slower operations like HTTP calls, for example, but there is a small problem: the `stream_set_blocking` function only works with plain files and with sockets. Other wrappers, such as `http://`, are not supported and cannot be handled in a non-blocking manner. But, of course, I wouldn't go through all this trouble to tell you now there's no way to perform asynchronous HTTP requests with PHP using this so-called event loop.

Http

As mentioned in the previous chapter, the `stream_set_blocking` function works with files, as we have worked already, and with sockets. Using sockets, we can do any network operation, including HTTP requests. So, in Listing 4 (see on the next page), an extended example shows how you can use sockets to perform an HTTP request and then fetch the response.

Note that in the example, even though the socket to the web server is the third stream to be open, it is the last one to have the contents displayed because it is the stream that takes the longest to be available for reading. If instead of five files and one HTTP request, we had multiple HTTP requests, all of them would be sent, and the first responses would be treated first, not necessarily in the order the requests were

Listing 4.

```php
declare(strict_types=1);
// Opening files and sockets, those two can be non-blocking
$fileStreamList = [
    fopen(__DIR__ . '/text_files/file1.txt', 'r'),
    fopen(__DIR__ . '/text_files/file2.txt', 'r'),
    stream_socket_client(
        "tcp://example.com:80", $errno, $errstr
    ),
    fopen(__DIR__ . '/text_files/file3.txt', 'r'),
    fopen(__DIR__ . '/text_files/file4.txt', 'r'),
    fopen(__DIR__ . '/text_files/file5.txt', 'r'),
];
// Setting all of them as non-blocking
foreach ($fileStreamList as $fileStream) {
    stream_set_blocking($fileStream, false);
}

// Performing the HTTP request manually via socket.
// $fileStreamList[2] has the socket connection
$httpRequest = 'GET / HTTP/1.1' . "\r\n";
$httpRequest .= 'Host: example.com' . "\r\n";
// Without this header, the socket will hang open.
$httpRequest .= 'Connection: close' . "\r\n";

fwrite($fileStreamList[2], $httpRequest . "\r\n");

// Event loop
do {
    $streamsToRead = $fileStreamList;
    $streamsWithUpdates = stream_select(
        $streamsToRead,
        $write, $except, seconds: 1, microseconds: 0
    );

    if ($streamsWithUpdates === false) {
        echo 'Unexpected error';
        exit(1);
    }
    if ($streamsWithUpdates === 0) { continue; }

    foreach ($streamsToRead as $index => $fileStream) {
        $content = stream_get_contents($fileStream);
      // Fetch the offset of the body ignoring the headers
        $bodyOffset = strpos($content, "\r\n\r\n") + 4;

        // If there is the 2 line breaks,
        // we treat it as HTTP and ignore the headers
        if ($bodyOffset !== false) {
            echo substr($content, $bodyOffset);
        } else {
          // Otherwise, it's a file read the whole content
            echo $content;
        }

        if (feof($fileStream)) {
            fclose($fileStream);
            unset($fileStreamList[$index]);
        }
    }
} while (!empty($fileStreamList));
```

performed, and without the need for one request having to wait for the previous one to be completed. That can bring huge performance improvements to routines that perform multiple I/O operations.

Conclusion

This manual work isn't straightforward to understand, but this is the exact intention of this post: showing how PHP can be async without any external components. There are multiple components that make our lives easier when trying to perform asynchronous operations. If we just want an HTTP client, `curl` can perform multiple requests simultaneously, and *Guzzle* takes advantage of that. For any type of stream, as we did in our example, *ReactPHP* is a wonderful solution. And using `stream_select` is one of the options *ReactPHP* gives you when you create an Event Loop. It's the worst one, but it is an option.

The reason for us to know how to achieve non-blocking I/O with "bare metal" PHP is to understand how tools like ReactPHP and even Swoole work behind the scenes. To understand how an Event Loop works and how I can take advantage of it, and most importantly, in which cases asynchronous programming can be useful in comparison to other approaches such as parallel programming. As Richard Feynman, Nobel laureate in physics, once said: "What I cannot create, I do not understand." So, in order to understand how tools that give us asynchronous powers work, we must be able to create something similar to them.

The techniques shown here, such as non-blocking I/O and event loop, are used not only when you want to perform I/O operations as the client, i.e., fetch data. In fact, the biggest results are yielded when you apply these techniques as the server. Non-blocking I/O, or as they like to call it, I/O multiplexing, is what made Nginx famous. Same for Node.js. And the exact same techniques are used to create overpowered servers in PHP using tools like AMPHP, ReactPHP, and Swoole.

With a degree in IT and being a Zend Certified Engineer, I always try to follow the good boy scout rule: "Always leave the code cleaner than when you found it". I am currently a Software Engineer at SOCi Inc, instructor at the biggest online tech school of Brazil, called Alura, and a YouTuber (https://youtube.com/`@DiasDeDev`*). I am one of the administrators of Brazilian PHP user groups such as PHP Rio and PHP Brasil, having spoken in multiple conferences.* @cviniciussdias

LONGHORN PHP 2023

Schedule is Live!

3 Days, 3 Tracks

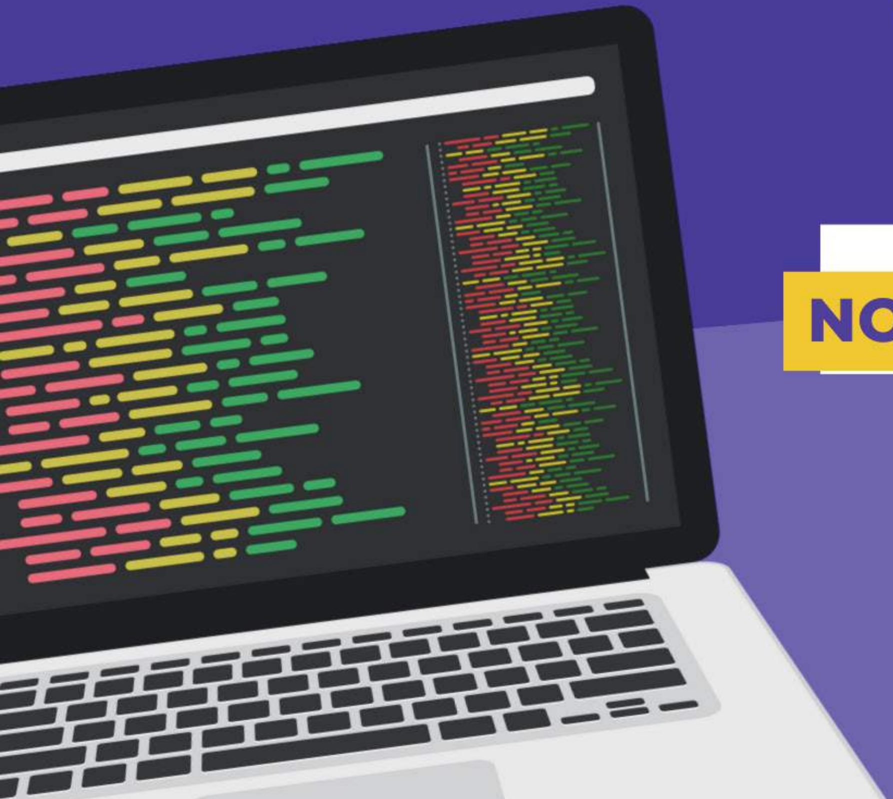
- **Thursday: Tutorials**
- **Friday & Saturday: Talks**

Awesome Speaker Lineup!

- **3 Keynote Speakers**
- **31 Talk & Tutorial Presenters**
- **42 Tutorials & Talks**
- **Open Spaces Discussions**

Plus:

- **Networking Opportunities**
- **Lunch Provided**



AUSTIN TX
NOVEMBER 2-4, 2023

Holiday Inn Austin Midtown
6000 Middle Fiskville Road

Buy tickets @
longhornphp.com

An Overnight Covid Ticketing System

Raja Renga

The year was 2021. It was a typical hot Indian summer day. Along with the heat wave, we were also facing the devastating Covid second wave. The country's public & health systems were running around the clock to save lives, with the immense personal sacrifice of health workers at every level. As IT professionals, we had the comfort of "Working From Home", which gave us economic and health safety. After hearing the hurdles of society every other day, the inner voice provokes us to do something useful. Apart from minor fund contributions, I could not serve society in any meaningful way. The healthcare workers were bearing the extent of the burden, and I felt guilty about it while also feeling useless.

Covid Challenged the World Biasedly

During the pandemic, I received a call from one of our earlier clients, whom I had not heard from in a long time and who worked in healthcare knowledge management. Within a few seconds of the conversation, I was happy that we had a chance to contribute by helping out with the proposed work we discussed. Here's how our conversation went: he said, "We need to implement a Covid ticketing system for Bengaluru Metropolitan and have a working system by tomorrow for a Government server. Will you be able to do it?" And I said, "We can!"

We received the requirements & earlier implementation details within the next half an hour. Their team tried internally for a few days, but due to the time limit and existing commitments, they looked for someone to carry it. But the time limit constrained the other takers, SAAS ticketing platforms seem quick to start, but the high customization needs and the per-user license rate make it unsuitable for their needs. We earlier worked on developing a custom reporting solution for the Lime survey tool. Based on the reverse engineering capability, they approached us.

It's a usual ticketing system with additional information that includes a patient's medical & demographic details. It has nearly 50 attributes with 24 masters, 5 conditional informational collection fields, a few decision-making fields, and a dozen departments with nearly 500 users to follow up on the ticket. The simple flow is:

- A toll-free number is given to the public to report on COVID-related things.
- A call agent will attend the call and capture the information from the caller.
- Based on the reason & current status of the patient, it will be directed to different experts & public departments like (Doctors, Ambulances, and Patients, residing Zone Officers, etc.)
- The concerned departments will follow up on the ticket.
- A central war room will monitor the ticket movements on a daily basis.

By the following day, the emergency system prototype had worked and was installed in a Government server on the same evening. The ticketing system gets into use by the third day with full process flow. The system ran for the next two months and handled nearly 55,000 tickets. From the initial usage of Bengaluru Metropolitan, the toll-free number further extended to the entire Karnataka state of India with flow changes to meet post-COVID challenges. The ticketing system was updated to align with new needs and reached 15,000 after two months of active usage. After the vaccination drive and medical improvements, it has slowly reduced the need for it. Now, it's serving as a ticketing system for the internal needs of the implementing agency in a new avatar.

The Unknown Word

Once the ticketing system got into full-fledged use of handling 2000 plus tickets, the experts from the Government asked one common question with an expected answer. "Which stack is it implemented in, either Django, Laravel, or Spring?" Our answer seemed to be a new word for them, perhaps not just them, but also for those of you reading this now. And that mysterious word is Fibenis. Surely, it's also the first time for you. It is a homegrown information system of Webstars Compugram, consolidated from a decade of improvements & refinements by a small team to overcome big challenges continuously.

Fibenis

Fiben Information System (Fibenis) is an adaptive full stack base development system, evolved by communication patterns and natural language principles. It's rooted in component-based development and continuous improvement.

The Base Line

Any information process cycle can be limited to BREAD: Browse, Read, Edit, Add, and Delete functions. Apart from the size of data and scale of operations, the underlying communication concept remains the same in most cases. It may have variations in data access points like tables and data delivery, like HTML, JSON, PDF, and more. However, most of the operations were finally transformed into SQL queries or structured interactions with data storage or access points.

The Communication Nature

In real-time human communication, repeated conversations in closed environments naturally become more concise and efficient over time. Our everyday interactions within family and office settings often require only a few words to convey a message. Such communication tends to rely on verbs and collective nouns, while using fewer subjects and proper nouns. This is due to the shared information embedded within the environment and context.

For instance, if we ask a friend “Takeout?” at the end of lunchtime near the canteen, it implies whether they have had their lunch or not. However, the same word can carry different meanings when used in front of a bank/ATM or a clothes shop. In these situations, our subconscious understanding of the subject is guided by the surrounding environment and context.

Conversely, communication can work in reverse as well. When a verb is commonly understood and repeatable within a specific context, we often communicate using only the subject or proper noun. For example, in an event ticket queue or a bank lobby, we typically use names, token numbers, or the word “Next” to convey the message. This is because the context itself encompasses the verb, making it unnecessary to explicitly mention it.

Structured Communication & Solution Pattern

Fibenis has taken this approach in building forms and desks. It carries the communication in a structured way that is simply a key and value term in computing. When a process is well-defined and standard, we usually communicate only the subjects being handled. The functional re-usability of a code block also lies in the same principle. The only variation comes in understanding the repeated procedures and solution patterns. We felt a pattern of repetitiveness in our solutions, mostly ending in a form and desk. So our communication has been restricted to only two possibilities, either a form or a desk. The process contains:

- type of communication (either form or desk)
- content for the conversation (information on which table and columns in key & value way)
- the response (reply of the other end)

Minimal Input for a Form

```
$F_SERIES = [
    'title'  => 'Flat DB',
    'table'  => 'demo',
    'key_id' => 'id',
    'fields' => [
        1 => [
            'label'   => 'Text',
            'id'      => 'text_flat',
            'type'    => 'text',
            'is_must' => 1,
            'allow'   => 'x30',
        ],
    ],
];
```

Minimal Input for a Desk

```
$D_SERIES = [
    'title'      => 'Demo Flat',
    'table_name' => 'demo',
    'key_id'     => 'id',
    'fields'      => [
        'tx' => [
            'label'   => 'Text ',
            'id'      => "text_flat",
            'head'    => ' width="80%"',
            'attr'    => [
                'class' => 'fbn-h5',
            ],
            'is_sort' => 1,
        ],
    ],
    'is_add'     => 1,
    'is_del'     => 1,
    'is_edit'    => 1,
];
```

The array based inputs are called as (def) definitions in Fibenis. A group of definitions will frame a module. An example module for the contact:

Sample Module Structure

```
+ contact
  |- fx.php
  |- dx.php
  +- customer
    |- fx.php
    |- x.php
```

Sample Calls

- Form- `localhost/fibenis/?f=contact`
- Desk- `localhost/fibenis/?d=contact`

Channelized Request & Response

- The URL call will call the respective engine processor and
- The processor will load definitions based on their location information
- The processed output will be sent as the response

The centralized process method improves the existing and integrates new things on the go. Fibenis has primary and secondary engines for different needs of information processing. The horses-for-courses approach differentiates the process, simplifies development, and supports a modular approach. It gives high reusability, scalability, and maintenance.

Engines & Definitions

In fibenis, the functional processors are called engines. In initial times, definitions were grouped by their process type. The form definitions and desk definitions are grouped in a folder. Due to the series of definitions in place, form and desk definitions are called Form Series (F_Series) and Desk Series (Desk). Due to the repeated use of definitions, it became def. Communication patterns play here. After a few years of usage, additional engines evolved for special purposes of content dissemination and detached BREAD operations. The secondary engines are:

Template Series(t_series)

The t_series has two inputs. A data picker def and a template for content generation.

```
// Data source file
$$T_SERIES = [
    'title'    => 'Demo Template',
    'table'    => 'demo',
    'fields'   => ['text' => ['id' => 'text_flat']],
    'template' => dirname(__FILE__) . "/tx.html",
];
```

```
<!--- Template Content--->
<ul>
    <TMPL_LOOP CONTENT>
       <li><TMPL_VAR TEXT></li>
    </TMPL_LOOP>
</ul>
```

Arrow Series(a_series)

Arrow series are meant for pinpoint BREAD operations. It will be a code block triggered by a structured request. The definitions can have multiple functions with access tokens.

- A simple a_series call `/fibenis/lite/?series=ax&action=demo&token='OTP'&data`
- a_series definitions

```
$A_SERIES = [
    'OPT' => function ($param) {
        return //process;
    },
];
```

Def's Locations & Access

Group by Engine

Initially, the defs are grouped by engine type. All the form `defs` in a f_series folder and all the desk `defs` in a `d_series` folder. For a simple contact case, a `contact.php def` is in both folders.

```
+ d_series
  |- contact.php
  |- customer.php
+ f_series
  |- contact.php
  |- customer.php
```

Group by Context

```
+ contact
  |- f.php
  |- d.php
  +- customer
    |- d.php
    |- f.php
```

Group by Usage

By the nature of usage, application-independent generic modules like contact user_info are needed in every installation. To avoid duplication and maintain the integrity of reusable modules, separate locations are kept for core modules and application-specific external modules. The core modules are kept inside the library. The external modules are kept out of the library and carry an additional x (fx.php/dx.php/tx.php/ax.php) to differentiate it.

Caller Heads

By Engine	Core Modules	Customer Modules
?d_series=contact	?d=contact	?dx=customer

The caller heads pick the modules from different locations. The divide helped manage the modules independently and consistently carry the modules' inter-dependence.

Flat & EAV Storage:

Fibenis was initially set up to handle forms with a flat relational table. Creating a table for every entity of an application increased the size of the table. Later, it became a challenge to manage the relational integrity of tables. It has both waste tables and scalability challenges. In most of our scenarios, master tables with minimal rows and a fixed number of columns like (id, code, short_name, long_name) seem consistent in most applications. So, we first checked for normalization of the same structure tables with a type code. Consider this case for three master tables:

Master_a

id	code	short_name	long_name
1	A	Name A	Lone Name A

Master_b

id	code	short_name	long_name
1	B	Name B	Lone Name B

Master_c

id	code	short_name	long_name
1	C	Name C	Lone Name C

Normalization by Structure

Master_type

id	type
1*	A
2	B
3	C

Master_base

id	master_type_id
5	1*
6	2
7	3
8	1*

The relation of `master_type_id` is used to differentiate the master. It helped to reduce the table count. However, there may be cases with uncommon fields for the entity. The custom fields are managed by an additional add-on.

Master_detail

id	master_base_id	code	short_name	long_name
1	5**	A1 Code	A1 Short Name	A1 Long Name
2	6	B1 Code	B1 Short Name	B1 Long Name
3	7	C1 Code	C1 Short Name	C1 Long Name
4	8	A2 Code	A2 Short Name	A2 Long Name

Master_addon

id	master_base_id	col_x1_addon	col_x2_addon
1	5**	A1 Addon x1 Value	A1 Addon x2 Value
1	8**	A2 Addon x1 Value	A2 Addon x2 Value

The Scaling Challenge

Though it helped to manage special cases, the base structure remains flat. Challenges come through situations like the frequent addition of columns in primary tables. It can be manageable if the rows are in thousands. When the table has hundreds of thousands and millions of records, it becomes a mounting task with indexes and constraints. Another concern is the memory waste from unused optional columns, which becomes critical during high record counts. We looked for a way to scale the column information dynamically and reduce the memory wastage for unused cases. At those times, document databases like MongoDB had just started to penetrate. We have been using MySQL/MariaDB/PostgreSQL for a long time, and changing the solution stack is a huge challenge to our team size. Also, we have been running applications for a long time with the LAMP stack. So, we looked to adapt a dynamic solution with the current tools. With the hint from Magento, we got into using EAV to solve the challenges.

EAV

EAV (Entity Attribute Value) is a modeling innovation where we can achieve dynamic scalability on top of

RDBMS—the EAV way tables for the earlier Master case.

Entity

id	code	name
1	MT	Master Types
2	MX	Master

Entity_attribute

id	entity_ code	code	short_ name
1	MT	MTCD	Code
2	MT	MTSN	Short Name
3	MX	MXTY	Master Type Code
4	MX	MXCD	Code

Entity_child

id	entity_code
1*	MT (Master Type A)
2**	MT (Master Type B)
3#	MX (a child of Master Type A)

Entity_attribute_value

id	entity_ child_ id	attri- bute_ code	value
1	1*	MTCD	==MA==
2	1*	MTSN	Master A
3	2**	MTCD	MB
4	2**	MTCD	Master B
5	3#	MXTY	==MA==
6	3#	MXCD	A1
7	3#	MXSN	A1 Short Name

In case there is a need for a detail column, reflection is in entity_attribute

id	entity_ code	code	short_ name
5	MX	MXDT	Detail

Reflection in entity_attribute_value

id	entity_ child_ id	attri- bute_ code	value
8	3#	MXDT	A1 Detail Content

EAV's adoption gives freedom on column scalability without changing the table structure; we can make column additions smoothly. At the least, we stored the values in entity_attribute_value. In real-time, each data type will have different value tables like `eav_attr_value_varchar`, `eav_attr_value_date`, `eav_attr_value_decimal`, etc. It will be solving the indexing and relation needs. But unlike a flat table communication of TCV(table, column, value), we need to communicate with multiple tables here. The pseudo queries for the EAV case:

Queries & Reflections for Adding a Master Type Child

Entity_child

- `INSERT INTO enity_child (entity_code) VALUES('MT')`
- Get the id of the insertion(1)

Entity_attribute_value

- `INSERT INTO enity_attribute_value(entity_child_id,attribute_code,value) VALUES (1,'MTCD','A'),(1,'MTSN,'Master A')...`

In Case of Multiple Value Tables for Different Types

The values will be stored in different tables based on the attribute type

- `INSERT INTO enity_attribute_char (entity_child_id,attribute_code,value) VALUES (1, 'MTCD, 'A')`
- `INSERT INTO enity_attribute_varchar (entity_child_id,attribute_code,value) VALUES (1,'MTSN,'Master A')`

Unlike a single insert query of a flat table, the multiple staged queries of an EAV design have overhead. Since fibenis already has structured communication, we checked how to adapt it based on communication without disturbing the current dialect. After some initial setbacks, the structured communication way was the given path to handle additional keys for handling EAV columns. Based on that, the engine will take care of an EAV process to manage it.

A Pseudo Definition for Master_a

```
$D_SERIES = [
    'title'      => 'Demo Flat',
    'table_name' => 'demo',
    'key_id'     => 'id',
    'fields'      => [
        'tx' => [
            'label'   => 'Text ',
            'id'      => "text_flat",
          'head'   => ' width="80%"',
            'attr'    => [
                'class' => 'fbn-h5',
            ],
            'is_sort' => 1,
        ],
    ],
    'is_add'     => 1,
    'is_del'     => 1,
    'is_edit'    => 1,
];
```

A Pseudo Definition for Master_a EAV Case

```
$F_SERIES = [
  'title'   => 'Masters',
  'table'   => 'entity_child',
  'key_id'  => 'id',
  'default' => ['entity_code' => 'MT'],
  'fields'  => [
    // for code column
    1 => [
      'label'                => 'Code',
      'child_table'          => 'entity_attribute_char',
      'parent_field_id'      => 'entity_child_id',
      'child_attr_field_id' => 'attribute_code',
      'child_attr_code'      => 'MTCD',
      'id'                   => 'value',
      'type'                 => 'text',
      'is_must'              => 1,
      'allow'                => 'x2',
    ],
  ],
];
```

The pseudo process will be:

- The engine will segregate the primary table and child table columns
- Handle the primary table first. The default table column is added to the primary query. `INSERT INTO enity_child (entity_code) VALUES('MT')`
- Child table columns will be handled in the second stage `INSERT INTO enity_attributechar (entity_child_ id,attribute_code,value) VALUES (1, 'MTCD, 'A')`

The EAV integration was given an orbital change in our solutions approach; the number of tables becomes a constant. The fine-tuning of indexing and constraints synced with every other application's needs. Due to the continuing usage, more functions are implemented with handling EAV wherever there is room for improvement.

Though Entities and Attributes are managed through an interface, the attributes are free in nature. The attributes can be stored in any attribute value table — it can also be bound to any input type. A challenge with that is that attributes become a mystery over time. We have to check the code to know the exact nature of the attribute. The more the form fields ended, the more the challenge to manage.

Base System & EAV Modeler

For a long time, the Fibenis base system evolved with the management of contact information with multiple groups, user role permissions, status logs, master panels for configurations, basic EAV management, and a few more modules for managing masters. Though it's helped in the basic setup of application systems with rapid speed and standardization, the development forms and desk phase had inconsistent time needs. Despite structured communication, remembering the growing number of keys became a challenge. The need for additional keys in EAV made the definitions lengthier and harder to manage with big forms.

Communication pattern theory helped us again. Entity attributes are bound with 15 plus input types of Fibenis form series. The addition of attributes has basic information fields and input type-specific additional fields. Based on that form, series definitions are generated by a form add-on sub-engine. It solved the challenges of inconsistency and scale. The EAV modeler allows us to use semi-skilled resources in the development process through minimal inputs. On the go, desk-specific information fields are added for implementing desks through a user interface. Now the modeler comprises of:

- Attribute Builder (From and Desk definitions)
- Entity Child Management Auto (Form and Desk Generator)
- Entity Child Base (For minimal and standard attributes)
- Entity Key Value (To manage one-time configuration and quick storage with reference)
- Entity Status Map (Manage status flows)

It helped to manage the entity-related things through a single window. Like the separation of core modules and application modules, the entities are also separated into core library and user-defined application modules. Also, both core and external modules have different storage tables to maintain the separation of concerns.

Moving Around

Improvements made in the EAV modeler have taken entities as super- structures, which have complete information on Form and Desk behavior. However, the next challenge arises in porting the entities between applications. Unlike a simple table export and import, EAV modeling has its primary challenge in transporting the structure since each field needs a separate process to handle it. Lessons from the OPC-UA modeling hint at a way to solve it—an xml structure-based data model defined for export and import. Using metadata for field attributes solved the basic communication issue during import. It's sped up the portability of entity structure. Specifically, the append mode helped update live applications where it was a complex thing earlier.

Steadily, the EAV modeler becomes a core development junction point. It has every piece of information on form fields and desk columns—the quantitative information on entities and attributes is given more granularity on ground-level implementation. For a few team members, it's given an orbital shift in implementing applications in a rapid, reliable, and standard way.

A Face-off with a Challenge

The moment we accepted the overnight implementation of the ticketing system, we immediately felt that the day had arrived we had longed for. With every implementation, we may face two things: The existing things on one side and the new things on another side. The ratio of these two things defines the seriousness of the implementation. On average, it will fall 60:40 between existing and new. Repeating the new things a few times naturally generates a way to generalize the process and make a reusable solution for similar needs. The little aggregations made the fibenis to scale today. Additionally, we have two more benchmarks, critical and safe for process certainty. By these classifications, we will get a character matrix for solution availability and process certainty. The four quadrant matrix will have:

Availability/Nature	Safe	Critical
Existing	Existing & Safe	Existing & Critical
New	New & Safe	New & Critical

We will separate the concerns by way of the character matrix and handle them progressively. The implementation will be by way of Ticketing System for the first five days.

Availability/ Nature	Safe	Critical
Existing	User Role & User Management / Master Configuration / Ticketing full status flow / Ticketing Dashboard (Basic) / Ticket form & desk formatting	EAV modeling for the ticketing system basic* / Attributes & Masters* / Base User roles* / Basic ticket flow* / Full EAV modeling with live attributes
New	Functions for conditional show & hide of elements / Functions for set & remove validation in bulk level / Importing users for EAV attributes through Perl script /Custom Dashboards & Reports	Installation of application environment in National Information Centre(NIC) infrastructure through remote connectivity* / Implementation of basic prototype in NIC covid ticketing server* / Addition of department attribute to the contact / Role & Department Based Ticket Desks / Call reason-based ticket diversion / Importing Area, Zone mapping for EAV attributes through Perl script

*Mentions the Day 1 tasks

Critical Moments

There were a few critical moments during implementation saved in an unorthodox way. The real testimony of software lies in meeting the reporting requirements. The top management sees only the information abstraction. After the days of phase-i launch, a report on complete ticket information was requested for showcasing to higher authorities on short notice of only a few hours. Usually, fibenis has the feature to export the desk as csv. Since the desk is customized to have minimal critical information, the custom configuration needs to define all the fields in a PHP associative array format. It had more than 70 fields; if we go by manual typing, it may take more than half an hour. But due to the tight time target, carrying manual work will become an overhead. Somehow, the Fibenis template series(`t_series`) capability produced the ticket attributes in the needed way. It's incorporated in the final moments before the high-authority meeting and saved the day for the ticketing team.

Raja Renga Bashyam is a Web App Architect,Developer & Designer from Indian Sub-content. Nearly 2 decades of continuous involvement to Web App Development in varied levels with wide range of implementations. Developed a Full-stack Adaptive Base Development System named Fibenis based on communication patterns to standardize development way to achieve re-usability and reduce development time.%[1]

We are Losing the Browser War

Chris Tankersley

In the vast digital landscape, browsers serve as our windows to the world, shaping our experiences, interactions, and perceptions of the internet. Over the years, this landscape has seen intense battles for dominance, often termed the 'browser wars'. These wars have profound implications for developers, businesses, and everyday users alike.

The first browser war between Netscape Navigator and Internet Explorer was a testament to the power struggles in defining the web's direction. Fast forward to today, and we find ourselves amidst another era of dominance, with Google Chrome standing tall as the undisputed leader. On the surface, this might appear as a victory for seamless web experiences, given Chrome's efficiency and robustness. Beneath this facade lies a web of concerns, ranging from monopolistic control, stifled innovation, and potential privacy breaches.

But why should developers, the architects of the web, be concerned? Would it not be easier to develop for a single, dominant browser? As we delve deeper into the annals of browser history and the implications of a monolithic web, it becomes evident that the stakes are high. The choices developers make today will shape the internet's future, determining whether it remains a diverse, innovative playground or transforms into a walled garden controlled by a few.

The Original Browser War: Rise of Ie

The turn of the millennium was a transformative period for the internet. As the digital world began to unfold, the tools we used to navigate it became the epicenter of fierce competition. At the heart of this battle were Netscape Navigator and Microsoft's Internet Explorer, two browsers vying for supremacy in an evolving landscape.

In the mid-1990s, Netscape Navigator was the shining star of the web. It had defeated Mosaic, one of the first popular web browsers. Its user-centric design and innovative features made it the browser of choice for many. By 1995, Netscape seemed almost invincible, commanding a significant portion of the market. I remember one of the first things I did once we got internet at home was open Internet Explorer… to download Netscape.

One of Microsoft's pivotal strategies was the integration of Internet Explorer with its Windows operating system. This move, though controversial, ensured that every Windows user had Internet Explorer available, giving it an unparalleled advantage over its competitors. By the dawn of the 2000s, this strategy, combined with aggressive development and astute marketing, catapulted Internet Explorer to a dominant position, capturing a staggering majority of the browser market.

But with this dominance came a series of repercussions. Microsoft's influence over the web began to grow. They introduced a slew of proprietary features and extensions in Internet Explorer. While some of these were considered groundbreaking, they often didn't adhere to open standards. This meant that websites optimized for Internet Explorer sometimes malfunctioned or displayed differently on other browsers, leading to a fragmented web experience.

Furthermore, with Internet Explorer's unparalleled market share, there was a noticeable slowdown in the evolution of web standards. The drive for innovation seemed to wane as Microsoft became comfortable with its market share, and the pace at which the web grew and transformed began to stagnate as Internet Explorer itself stagnated.

This period also saw Internet Explorer grappling with many security vulnerabilities. Its deep ties with the Windows operating system made it a lucrative target for malware and cyberattacks, eroding user trust and compelling developers and businesses to invest in security patches and workarounds. As the world moved to a more internet-centric model, having a browser embedded in the operating system turned out to be a poor idea.

For developers, this era presented a unique challenge. With the vast majority of users on Internet Explorer, there was a palpable pressure to focus on it, often at the expense of other browsers. Businesses became entrenched with Internet Explorer on their machines, forcing developers to design around Internet Explorer's failings. This not only led to a less inclusive web but also stifled the potential for diverse innovations and approaches.

Looking back, the rise and dominance of Internet Explorer, which was originally built from a licensed commercial version of Mosaic that Netscape had helped dethrone, serves as a reminder of the complexities and challenges of the digital realm. While a single dominant player might offer short-term benefits, the long-term implications for innovation, security, and the very ethos of an open web can be profound.

Parallels with Chrome's Rise

While Netscape Navigator turned into a bloated mess, out of its ashes rose the Phoenix browser, then renamed Mozilla Firebird, and then further renamed Mozilla Firefox. It was a great alternative to Internet Explorer for many developers and web-savvy users, but the web moves fast. Google was not happy with Mozilla or Microsoft controlling the web.

Google Chrome, introduced in 2008, wasn't merely another browser—it was a clarion call from Google, signaling its intent to redefine our online experiences.

When Chrome debuted, the browser landscape was still lacking in choices. Firefox had carved out its space as a formidable alternative to Internet Explorer, while Safari was the preferred choice for the Apple aficionados. Chrome's entry was more than a ripple in the pond of the internet. Its minimalist design, coupled with the promise of unmatched speed and stability, set it apart. Beyond its technical prowess, Chrome represented Google's holistic vision for the web, one that seamlessly brought together search, apps, and services.

One of the driving forces behind Chrome's rise was its deep integration with Google's suite of services. Whether it was Gmail, Drive, or YouTube, using Chrome felt like stepping into a cohesive ecosystem where everything was interconnected. This relationship between the browser and services made the user experience fluid, binding users ever closer to Google's digital walled garden.

Performance was another feature touted by Chrome. Chrome was not just fast—it was blazingly fast. The V8 JavaScript engine under its hood ensured web pages loaded at lightning speed and web apps ran without a hitch. This performance edge was coupled with Chrome's commitment to regular updates, ensuring users always had access to the latest features and security enhancements.

It wasn't only technical excellence that propelled Chrome to the forefront. Google's marketing machinery played its part to perfection. From online advertisements to billboards and even television commercials, Chrome was omnipresent. Users were already

visiting Google sites every day, so it was easy for users to want to try out Chrome. Developers wanted to use cutting-edge features, so Chrome was an easy sell.

Today, as we gaze upon the vast expanse of the internet, Chrome's influence and presence is unmistakable. Its dominance evokes memories of Internet Explorer's reign, and with it come similar concerns. With such a significant share of the browser market, Google wields considerable influence over the trajectory of the web. While this dominance has led to standardization and consistency, it also raises questions about centralization of power, potential stagnation in innovation, and concerns over user privacy.

Drawing these parallels between the past and present, it becomes evident that while browsers may change, the challenges and opportunities they present remain similar. It is up to the digital community to ensure that history's lessons are heeded and that the web remains a diverse and open frontier.

Why a Chrome-centric Web is Bad

It can be easy to see why Chrome has gained such a huge market share. Its sleek design, rapid performance, and how it melded with Google services have made it a darling of the digital age. But beneath this polished surface lies a potential pitfall: the dangers of a web dominated by a single browser.

Centralization of power is one of the most significant concerns. When one entity, in this case, Google, holds the reins to the web's direction, it can inadvertently stifle alternative viewpoints and innovations. The web thrives on diversity, and a singular influence can lead to a homogenized experience.

With Chrome's dominance, Google's voice in web standards becomes overwhelming. While they've often been proponents of open standards, the potential for bias or decisions that put their interests over the broader community's cannot be ignored. Even today, Google launched new features into Chrome, which developers can use and get used to before they are officially standards, or can add a superset of features onto a standard that developers and users start to expect.

Then (and probably even more important), there's the looming shadow of privacy. Google's revenue stream is tied to advertising, a model that feasts on user data. With Chrome's widespread use, Google has an unparalleled view into users' online habits, raising eyebrows about surveillance and potential misuse of personal information. And while users might appreciate Chrome's features, they might also feel cornered into using it, given its omnipresence, even if they harbor reservations about Google's data practices.

Let's take a look at the new Privacy Sandbox feature that is currently rolling out to Chrome users. On its face, it's an admirable goal to stop the amount of information third-party cookies allow companies (including Google) to gather about you. The only problem is that Privacy Sandbox trades the existing broken structure for a new standard controlled by Google that now lets advertisers directly access APIs in the browser to decide what ads to show you.

Sure, you can get some more control over what is shown or shared, but at the end of the day, instead of blocking third-party cookies like Firefox or Safari, they just changed how advertisers can get that info and baked it directly into your browser. Now, your information is just shared differently.

Innovation could also take a hit. History is full of examples where market dominance leads to complacency. Chrome, despite its current prowess, is not immune to this. With a large share of the market, the drive to innovate might wane, and new entrants with fresh ideas might find the barriers to entry insurmountably high.

For developers, a Chrome-centric world presents its own set of challenges. The temptation to primarily optimize for Chrome is real. Have you ever visited a Firefox or Safari site only to be told, "This site works best in Chrome"? We now live in an internet where sites focus on one browser, Chrome, to the detriment of users of other browsers. This not only reduces user choice but also risks creating a fragmented and less inclusive digital landscape.

For users, nothing is more annoying than being told you need to use a different browser. I recently tried to sign up for an account through a bank, and naturally, they wanted to verify my phone number through SMS. That failed, so I tried e-mail. That also failed. Do you know what worked? Using Chrome. It turns out their system did not support Safari, which their technical support knew about. When I mentioned I was using a Mac, they immediately told me to use Chrome and not the major, built-in browser that the OS ships with. They only supported Chrome-based browsers.

While Chrome's rise in the browser world is a testament to its quality and Google's vision, an internet landscape dominated by it is fraught with challenges. For the web to remain open and innovative, it's essential to push for standards with multiple implementations, not a single implementation that drives and controls the standard.

Alternatives to Chrome

While Chrome stands as a towering giant, it's by no means the only option for users. Several browsers offer unique features, privacy controls, and visions of what the web can be, challenging Chrome's dominance and providing users with a plethora of choices.

Mozilla's Firefox, for instance, has long been a beacon for those who prioritize an open web. Born from the non-profit Mozilla Foundation, Firefox isn't just a browser. It is a statement of intent, a call that an open web is a better web. It champions user privacy, open web standards, and has defaulted to privacy first, introducing features like container tabs that allow users to separate their online identities. Firefox's commitment to a user-centric web makes it a compelling choice for those wary of corporate influence.

Then there's Apple's Safari, the browser deeply integrated into the macOS and iOS ecosystems, much like Internet Explorer did so many years ago. While it has some great performance numbers and battery efficiency, Apple has been doubling down on privacy features, introducing intelligent tracking prevention to shield users from prying eyes. For those embedded in the Apple ecosystem, Safari offers a seamless browsing experience across devices. Sorry Windows users.

And, realistically, those are your alternative options in 2023. Browsers like Microsoft Edge, Brave, Opera, and others tend to be based on the Chromium engine, which is the same engine that powers Google Chrome. This means that under the hood, these browsers still adhere to Google standards even if they attempt to change Chromium to be "less Google."

The reason for this is that at the heart of all browsers is something called a Rendering Engine. We currently have three major rendering engines:

1. Quantum, the rendering engine used in Firefox that replaced the older Gecko engine.
2. WebKit, the rendering engine of Safari.
3. Blink, the rendering engine used by Chromium and Chromium-based browsers and based on earlier versions of WebKit.

These rendering engines are then paired with a JavaScript engine:

1. Spidermonkey, used in Quantum.
2. JavascriptCore (or Nitro under Apple's branding), used in WebKit.
3. V8, used in Chromium.

It takes a lot to make a good rendering engine and JavaScript engine, so many browsers are based on one of the above three options. The problem is that almost all alternative browsers use Blink/Chromium as a base. As much as I can tell, there are currently no alternative browsers based on Quantum, short of repackages of Firefox itself like IceWeasel and Librewolf, and the only "major" alternative browser using WebKit is GNOME Web, which runs only on Linux.

Browser developers like Brave and Opera are taking steps to "de-Google" their browsers, but at the end of the day, they are still using technology built and controlled by Google. They are beholden to the standards and technologies that Google wants to encourage, even if they take out the various unwanted parts that exist in Chrome.

Navigating the Web's Crossroads

The digital tapestry of our world, woven through the vast expanse of the internet, is as diverse and intricate as the cultures that birthed it. At the heart of this tapestry lies the browser, our window into the digital realm. Over the years, this window has seen its frame shift and change, from the early days of Netscape Navigator to the towering presence of Chrome today.

Reflecting on the journey, it's evident that while a dominant player like Chrome brings uniformity and consistency, it also casts a long shadow, obscuring the vibrant diversity of the web. The lessons from Internet Explorer's reign remind us of the risks of placing too much power in the hands of one entity. It's not about technical superiority or user experience; it's about the very ethos of the web — a decentralized, open platform where innovation and diversity thrive.

The alternatives to Chrome, each with its unique vision and strengths, underscore the richness of this ethos even if they do not have the market share of Chromium. They remind us that the web's strength doesn't lie in homogeneity but in its plurality. Whether it's Firefox's commitment to an open web, Safari's seamless integration, Edge's rebirth, Brave's challenge to traditional monetization, or Opera's consistent innovation, each browser offers a distinct path through the digital forest.

For developers, this diversity is both a challenge and an opportunity. It's a call to design and build for an inclusive web, where applications and sites shine regardless of the window through which they're viewed. For users, it's an invitation to explore, step beyond the familiar confines of one browser, and experience the web's myriad possibilities.

In the end, the future of the web hinges on choices—choices made by developers, businesses, and everyday users. By championing diversity and resisting the allure of a monolithic web, we can ensure that the digital tapestry remains vibrant, innovative, and representative of the global community it serves.

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

The Apocalypse is Now

Eric Mann

The world's leading experts on artificial intelligence have warned us of a coming "AI Apocalypse". How real is this threat, and when will we see it?

Thankfully, none of us are standing at a fence yelling at families to run while a rogue cyborg drops a nuke on a neighboring city. Apologies to Sarah Connor, but that's not the kind of apocalypse we need to worry about with the state of AI in today's world.

Instead, we're faced with a society eager to offer up their blogs, reading habits, purchasing behavior, or even retinal scans[1] for a quick buck. The downside here is that the folks scraping, downloading, and hoarding this data are using it to train their own artificial intelligence models to write more content, sell more goods, or even corner cryptocurrency.

The *current* AI apocalypse is around data, its ownership, and the endgame of those who control it. But it's not all bad news. Those who understand the differences between the fields involved—AI, ML, and DS—are well-armed to survive the apocalypse and keep their data for themselves.

AI, ML, and DS

Having worked professionally in this space for quite a while, there are three particular acronyms that keep coming up and are often confused for one another. To really understand whether or not Skynet is coming to kill us, you must first recognize which acronym refers to which part of the apocalypse. In the reverse of the order in which you'd normally see them:

- **DS - Data Science** is the general field of study of information and metadata. It's one of the hottest professions in today's economy and usually revolves around building deep subject matter expertise so you can tease out patterns from huge sets of data. Identifying that there's a pattern or some sort of predictability in a dataset is the first step towards building a model, and this kind of **exploratory data analysis** is done by humans before any smart machines are ever involved.
- **ML - Machine Learning** is the more specialized application of specific algorithms and relatively hard math to instruct a machine to identify patterns and make them more visible. A relatively straightforward example would be regressions - you use data science to identify the most important features of a house, then build a regression (linear, logarithmic, or otherwise) to leverage those features to predict a price per square foot for real estate investors. The Zillow Zestimate[2] is a fairly accurate predictor of home values that uses exactly this approach.
- **AI - Artificial Intelligence** is similar to machine learning but asks a computer to produce inferences about data (often taking action on those inferences) without training it in advance. In the media, you might hear about **deep learning** or unsupervised training - these are examples of artificial *narrow* intelligence where a machine can do one thing well. Skynet (or Legion or any of the other apocalyptic killer AIs) would leverage artificial *general* intelligence at levels at or above that naturally represented by humans.

Today, we are witnessing the birth of artificial narrow intelligences almost daily—each one causing tectonic shifts in whichever field into which they're launched. Photoshop's content aware fill/remove features[3] empower even novice photographers to quickly and professionally edit photos in ways that used to take high levels of niche expertise and significant investments of time.

Generative tools, like ChatGPT and Stable Diffusion, create believable blocks of text or even artwork that looks like it was created by a human. Such artificial narrow intelligence is fascinating, potentially improving any industry that can use them[4], and it exposes us to a strange world where our own content and metadata have been used to build the tools without our consent[5].

Who Owns Your Data?

In March[6], video conferencing leader Zoom updated its terms of service to require customers to consent to the user of their data for "training and tuning of algorithms and models". While the company has tried to explain nuance around this change, fundamentally, it gives them the right to repurpose any of your data to train an AI.

Originally, experts feared this would include any of the content exchanged through a Zoom call—including text chat or even video/audio transcripts. Zoom explained last month that their terms are explicitly targeting "service generated data", like how you use the product itself.

Regardless, the new terms of service present Zoom with an amazing amount of permission to use data about *you* to

1 https://phpa.me/worldcoin

2 https://www.zillow.com/z/zestimate/

3 https://phpa.me/remove-object

4 https://github.com/features/copilot

5 https://phpa.me/banned-chatgpt

6 https://phpa.me/zoom-terms

train their AI models. Ideally, they'll use this to target future product features to make their system more attractive. But there's no real guarantee that's what they'll do and no real limit on whether or not they can sell this data to third parties.

At the end of the day, ownership of your data can be murky at best. You might own your data, or your habits, usage, and even produced content could be owned by and sold to the highest bidder—often without your knowledge!

What Can You Do?

Despite early (and even ongoing) pushback by regulators, organizations like OpenAI are trying to make it easier to opt out of having your content scraped in order to train their next iteration of Skynet new tools. Similar to how you can block Google from indexing specific pages or even your entire site, you can add a user-agent block to your site's `robots.txt` file to block the system.

```
User-agent: GPTBot
Disallow: /
```

You're still relying on the honesty of OpenAI that their tools will respect this flag, but it's a start. Understanding who is accessing your data (created content or just usage metadata) is critical to ensuring you're not eventually victimized by the AI-based tools purportedly built to help us.

The easiest way to avoid becoming another set of features used to train an ML model or AI system is to fight back at the data science (DS) level by keeping your data out of the system entirely! Take time to understand what tools people are using to mine insights from your data and then invest the effort necessary to keep your private info exactly that: private.

Related Reading

- *Security Corner: Vulnerability Management 101* by Eric Mann, August 2023.
https://phpa.me/2023-08-security
- *Security Corner: Prisoner's Dilemma* by Eric Mann, July 2023.
https://phpa.me/security-jul-23
- *Security Corner: Types of Tokens* by Eric Mann, June 2023.
https://phpa.me/security-jun-23

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

Unseeable Colors

Maxwell Ivey

I recently spent time on an e-commerce site trying to buy new professional clothes. I received a fee for a recent talk on overcoming adversity from the Wyoming Governor's Council on Developmental Disabilities. Afterward, I decided to invest some of that money to maintain my appearance.

I am a rather large man at 6'4" and over 275 pounds. So, my natural go-to is Kingsize Direct, now dXl. I went with them because I have built up confidence in the quality of their products and the likelihood that their clothes will fit me when they arrive.

Having confidence in my decisions about what to purchase and from whom when buying online is extremely important to someone who can't see the images on the screen. This is especially true if you want to avoid asking a sighted person to invest lots of their time while surfing the net.

Beyond clothing style, sizing, and delivery options; knowing what color something is can be really challenging.

In their efforts to distinguish their products from their competitors, most businesses give their products unusual, cutesy names.

Now, I did have perfect vision up until the age of five or six. And I used to be able to see colors. But there were only 24 colors in my box of Crayolas.

I can't tell you how often I have been on the phone with a customer support person and had to ask them to explain what a particular color looks like.

So, the other night, I had a crazy idea. Or maybe it wasn't so crazy, given how much we depend on our wireless home devices. I decided to ask Siri and Alexa to tell me what some of those unfamiliar color names looked like.

Here are some examples of what I found.

While shopping for polos, I ran into the following:

Deep shade, which I assumed would be a dark green color. But, I was later told that, in fact, it has a purple hue even though some people may see it as a very deep blue or indigo.

Then, there was purple heather, which would seem to be self-explanatory. But I was told it was similar to lavender. I had to refresh my memory of what was lavender. I found out later that it is a pinkish shade of what looks like purple.

Then, there was blue atoll. This is the description I got for it: A blue that has a range of shades between light blue and turquoise. It is the color of the sky and the sea.

Then, while shopping for dress shirts:

I ran into one called tibetan red. I found out it is red mixed with pink. I decided to take a chance on it anyway, figuring that I could always wear it for business casual if it didn't work with one of my jackets.

I was surprised to find that both Alexa and Siri could tell me every time what a color looked like. They would sometimes also tell me the manufacturer's name and what website I could go to to verify the description.

At first, I was confused. I asked myself how could that be happening. Then, I decided these companies must register their color names somewhere to ensure no one else uses them. And Alexa and Siri must pull their information from those websites.

I was excited by my discovery, and I started telling my blind friends right away about this great new workaround.

Then, a friend told me that it may work with clothes but never with cosmetics. She further explained to me that the names given to lipstick, nail polish, eyeliner, etc., are often given really crazy names.

When I asked her for an example, she said there was a nail polish called "I am not a waitress".

So, I put it to the test and asked my Alexa for a description. My Alexa device told me that the color in question was a deep red wine with pink accents. I confirmed this to be true with my friend.

Now, you might be wondering how this affects you. How do I use this information to create more inclusive websites?

The way I see it, the first thing is to realize that people with vision loss or other disabilities see the world differently. If your world is filled with color, you may not think about these things as you navigate websites. Now that your awareness has been raised, you have the opportunity to share this information with site owners and fellow developers.

The second thing is to make color names understandable to all users.

I think this could be done in two ways.

Option one, the alt text tags for images should include a more detailed color description, perhaps by adding data entry fields to remind staff to enter them.

This is a challenge as even large e-commerce sites routinely leave out image descriptions or only offer the bare minimum, which tells the visually impaired consumer very little about the product.

Or option two, you could code in a button that fetches color descriptions when a user needs them.

I don't know how difficult that latter option would be to build in, but I routinely run across buttons to produce a size chart.

You may have an even better solution for how to solve this problem. My top goal for my column is to build relationships and start conversations that lead to better solutions and broader implementation of the best options for creating a more inclusive online world.

Building confidence in business is an important part of their success. However, creating a site that helps the buyer have more confidence in their choices could have an even more significant impact on a company's customer base and their related bottom line.

After all, no one wants to have to return merchandise. It's expensive and a hassle. E-commerce sites are interested in selling their products. They do not want visitors to opt out of purchasing because they aren't sure what they read in the descriptions matches the color.

I look forward to your questions, comments, and suggestions. Thanks, Max

Maxwell Ivey is known around the world as The Blind Blogger. He has gone from failed carnival owner to respected amusement equipment broker to award-winning author, motivational speaker, online media publicist, and host of the What's Your Excuse podcast. *@maxwellivey*

Comb Sort

Oscar Merida

The Bubble Sort is an easy-to-understand algorithm to order array elements from smallest to largest. One drawback of the bubble sort is that we compare one element to an adjacent element. We'll see how the Comb sort addresses the limitation and compare its performance with the Bubble sort.

Recap

> *Write an implementation of the Comb sort. Generate an array of N random numbers. Pick as large a range as you want to work with, but don't make the range too small, and use your comb sort function to order it.*
>
> *Again, generate many such arrays and collect and present data on how long it takes to sort. Show the shortest, longest, and mean times.*

Turtles and Rabbits

It takes a lot of swaps for elements that have to move a large distance. Consider a scenario where the smallest element in your array is in the last position. On the first pass, it'll move one slot closer to the beginning. You'll need N loops just to move that element to the start of the array. These are "turtles". You have to make N-1 comparisons to move it to the end of the array. Rabbits are large values near the start of the array. We could move them to the end of an array in fewer swaps if we don't compare adjacent elements. However, they don't require as many operations to move to the correct position at the end of the array.

The Comb Sort addresses this limitation by comparing non-adjacent elements. For example, we can compare an element with an element five slots away on the first pass. On the second pass, we compare each element with one four slots away, then three away, until we get to comparing slots that are adjacent. This is known as "the gap." As in the bubble sort, we stop when we are comparing adjacent elements and don't swap any of them.

PHP Implementation

I started with the code for last month's solution. The bubble sort is a case of the comb sort where the gap is always set to one. We'll need a variable to define the starting gap and then narrow it as we loop through the array.

Listing 1 shows how to update the bubble sort to work like a comb sort. For starters, I chose a shrink factor of two. That makes the first gap half the size of our array. Shrink factors of

Listing 1.

```php
/**
 * We're assuming sequential integer keys
 * @param array<int, scalar> $list
 */
function comb_sort(array &$list): void
{
    $max = count($list);
    $shrinkFactor = 2;
    $gap = ceil($max / $shrinkFactor);
    do {
        echo "\nGap: " . $gap;
        // stop two elements before max so we have two
        // elements to swap
        $swapped = false;
        for ($i = 0; $i < $max - $gap; $i++) {
            $target = $i + $gap;

            if ($list[$i] > $list[$target]) {
                $swapped = true;
                swap_elements($list, $i, $target);
            }
        }
        // reduce the gap again
        $gap = ceil($gap / $shrinkFactor);
    } while ($swapped);
}
```

less than two would compare elements that are farther away, while larger values compare items that are closer together (since this is a denominator when we calculate the gap.)

Wherever we had hardcoded `1` to compare elements or keep our loops within the bounds of our array, we can instead use `$target`.

Another thing to watch out for is letting the gap be set to zero. That happened to me when I used `floor()` to make the gap value an integer. If it falls to zero, the algorithm loops through the array, compares each element with itself, makes no swaps, and exits, leaving a partially sorted array.

Best Shrink Factor

I picked `2` as a shrink factor arbitrarily. Let's use the benchmarking tools to see what it should be, knowing it has to be greater than 1 and less than the size of our array. It doesn't have to be an integer value, either. The table below summarizes the mean time to sort an array of 5000 items. Shrink factors larger than 2 are inefficient, while 1.3 is the sweet spot. This matches up with the expected optimal value, according to Wikipedia[1].

> *The shrink factor has a great effect on the efficiency of comb sort. k = 1.3 has been suggested as an ideal shrink factor by the authors of the original article after empirical testing on over 200,000 random lists. A value too small slows the algorithm down by making unnecessarily many comparisons, whereas a value too large fails to effectively deal with turtles, making it require many passes with a gap of 1.*

Shrink Factor	5,000 items
8	1.0361
4	0.7492
2	0.4758
1.8	0.1352
1.7	0.1146
1.6	0.08339
1.5	0.03946
1.4	0.00805
1.3	0.00638
1.2	0.00750
1.1	0.01098

Testing Performance

Now that we know the "best" shrink factor to use, we can benchmark and compare performance to the bubble sort.

Comb Sort	1000	5000	10000
Fastest	0.0009739	0.005993	0.01325
Slowest	0.0017241	0.015620	0.02577
Mean	0.0010537	0.006340	0.01420

Bubble Sort	1000	5000	10000
Fastest	0.0301	0.8095	3.2070
Slowest	0.0370	0.8780	3.4395
Mean	0.0321	0.8414	3.3420

We made one straightforward change to deal with our turtles and, to a lesser extent, the rabbits. How much faster is this? The table below summarizes how many times faster the comb sort is compared to the bubble sort for various array sizes.

Times Faster	1000	5000	10000
Fastest	30.91	135.07	242.04
Slowest	21.46	56.21	133.47
Mean	30.46	132.71	235.35

One thing that jumps out is that the comb sort becomes relatively much faster as the array gets larger. Intuitively, this makes sense since turtles must impose a huge performance tax on large arrays—the number of comparisons and swaps required to move them toward their correct place in the array compounds. We can see how slightly tweaking an algorithm can lead to drastic performance improvements.

1 https://en.wikipedia.org/wiki/Comb_sort

Insertion Sort

For next month, write an implementation of the Insertion Sort, one of the fastest algorithms for sorting "small" arrays. Generate an array of N random numbers. Pick as large a range as you want to work with, but don't make the range too small, and use your insertion sort function to order it.

Again, generate many such arrays, and collect and present data on how long it takes to sort. Show the shortest, longest, and mean times.

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 PsySH[2] 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.*

2 https://psysh.org

Create Observability, Part 4: Simple Queue System

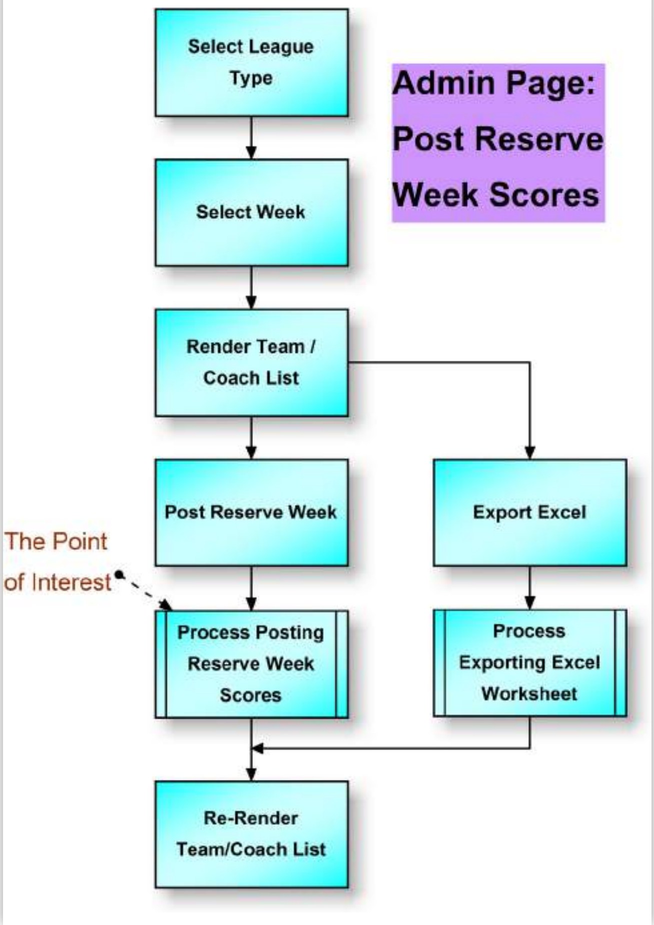
Edward Barnard

This month, we are identifying a business process that needs to move offline for asynchronous processing. To support the move, we'll design a simple queue system based on MySQL database tables. Next month, we'll design observability into our business process as we move it offline.

Last month, we created a flowchart to show our rewrite of a broken business process. This month, we'll be preparing to rewrite the business flow for a different process, which we call "post Reserve Week scores". See Figure 1. Note the step marked "The Point of Interest". We'll be rewriting that step.

First, we need to look at the context and understand the problem to be solved.

Figure 1.


Select League Type
Admin Page: Post Reserve Week Scores
Select Week
Render Team / Coach List
Post Reserve Week
Export Excel
The Point of Interest
Process Posting Reserve Week Scores
Process Exporting Excel Worksheet
Re-Render Team/Coach List

Reserve Week

At the USA Clay Target League, a competition season consists of six weeks. Coaches post athlete scores each week. At the close of each week, we calculate athlete and team standings. Competition is outdoors, which means some teams might be unable to compete due to snow storms, hurricanes, etc. That's why we have an optional "Reserve Week", which is the week prior to the first week of competition.

For each team choosing to take advantage of their Reserve Week, athletes compete, and coaches enter scores just like they would for a regular week of competition. Then, if the team is prevented from competing due to weather, the Head Coach for that team may opt to use the Reserve Week scores as their scores for that week.

For the team, this is a "yes or no" situation. Either the entire team uses their Reserve Week scores, or nobody on that team does. When using the Reserve Week scores, that team does not enter any scores because we already have the scores in our database.

Any particular team might compete in multiple disciplines (trap, skeet, 5-stand, and/or sporting clays). The Reserve Week scores are for a specific discipline. It's possible a team was able to complete their Trap competition but was barred from their Skeet competition. For example, a snowstorm might have appeared mid-week!

Part of our processing flow is to make a list of teams that still need to submit scores. Our staff contacts the affected teams, reminding them to get all scores entered before our calculation deadline.

Post Reserve Week

Ultimately, at some point, we need to make the weekly score calculation. At this point, we list teams with *no* scores entered for this week. It's a list per discipline per team; therefore, the list is called "team-discipline" in the diagrams. Then, for each team-discipline with no scores entered, we post a copy of that team-discipline's Reserve Week scores to become that team's scores for the current week for that discipline.

That's the process, and the process does not need to change. The problem is with sending email.

We send a score summary to each affected coach. This could include several coaches for any given team. Therefore, we could send out dozens or hundreds of email messages. That fact, in turn, means a processing time of several minutes.

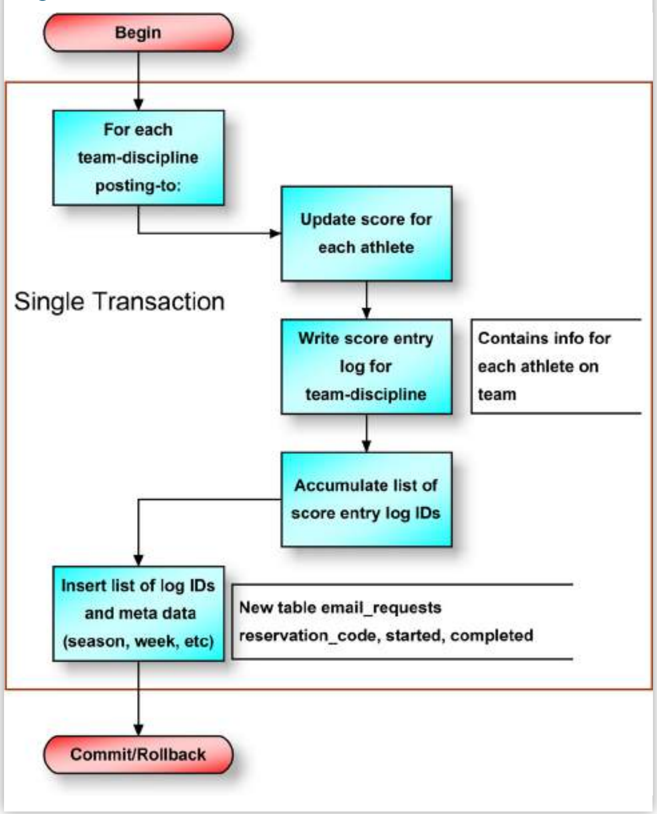
The administrator must wait several minutes for the webpage to reload.

Instead, we will move email sending to an offline process.

Single Transaction

Our first step is to rewrite Post Reserve Week to run inside a single transaction. See Figure 2.

Figure 2.


Begin
For each team-discipline posting-to:
Update score for each athlete
Single Transaction
Write score entry log for team-discipline
Contains info for each athlete on team
Accumulate list of score entry log IDs
Insert list of log IDs and meta data (season, week, etc)
New table email_requests reservation_code, started, completed
Commit/Rollback

What happened to sending email? That's handled by the bottom blue node of the flowchart in Figure 2. The new table `email_requests` contains pointers to the information we'll need to generate and send email messages to the various coaches affected.

However, before we go further, we need to look at how we intend to handle queue processing.

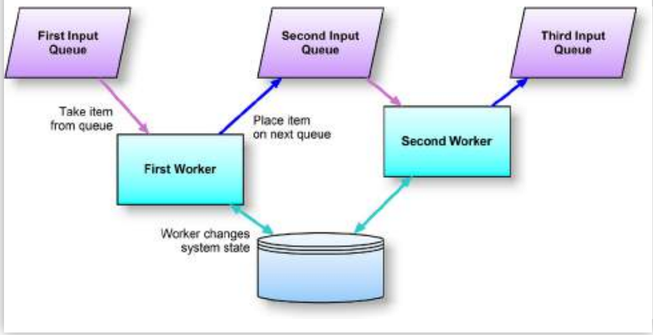
Simple Worker

We'll be taking an "assembly line" approach to offline processing. See Figure 3.

Picture a worker who has one, and only one, task or responsibility. This worker takes an item of work from its input queue, processes the item, and passes one or more new items along to the next worker (by creating items on the next queue).

How does this work in practice? In our example, the webpage posted reserve week scores all in one transaction (Figure 2). That transaction also inserted a single record into the `email_requests` table. That table is the "first input queue" at the upper left of Figure 3.

Figure 3.



Our next step, which will be the "First Worker" of Figure 3, will be to read that single record from `email_requests`. This first worker will split that information into one request per email recipient. This first worker places each request into the second input queue.

For example, if we post reserve week scores and need to send emails to 60 recipients, the first input queue will have one record before the first worker runs. Once the first worker completes, the second input queue should have 60 records, one for each recipient.

Simple Queue

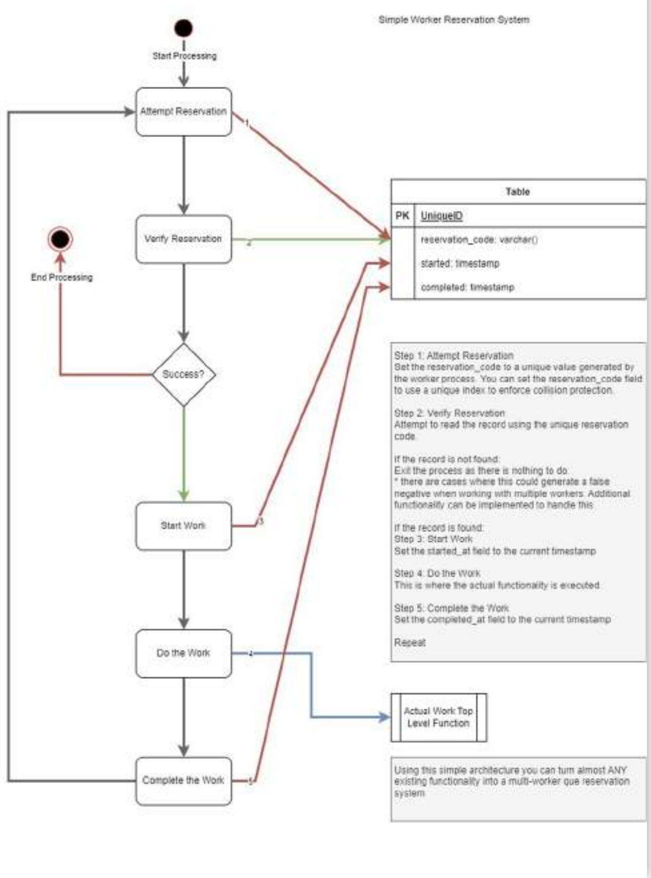
We'll be using Dave Pomeroy's "simple queue" design, allowing workers to automatically scale as workload increases. See Figure 4 (on the next page). Dave drew this diagram for me from scratch in about 15 minutes. That's a good measure of how clean-and-simple this design is!

Listing 1 shows the first input queue, `email_requests`. I changed the reservation code slightly because I prefer to use a 128-bit random number. I generate the number with `bin2hex-(random_bytes(16))` and store it in the table as `char(32)`.

Listing 1.

```sql
CREATE TABLE `email_requests`
(
    `id` int unsigned NOT NULL AUTO_INCREMENT,
    `reservation_code` char(32) DEFAULT NULL,
    `reserved_at` timestamp(6) NULL DEFAULT NULL,
    `started` timestamp(6) NULL DEFAULT NULL,
    `completed` timestamp(6) NULL DEFAULT NULL,
    `queue_parameters` mediumtext NOT NULL
          COMMENT 'PHP serialize()',
    `created` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP,
    `modified` datetime NOT NULL DEFAULT CURRENT_TIMESTAMP
          ON UPDATE CURRENT_TIMESTAMP,
    PRIMARY KEY (`id`),
    UNIQUE KEY `reservation_code` (`reservation_code`)
) ENGINE = InnoDB DEFAULT CHARSET = utf8mb4
  COLLATE = utf8mb4_0900_ai_ci;
```

Figure 4.



Step 1: Attempt Reservation
Set the reservation_code to a unique value generated by the worker process. You can set the reservation_code field to use a unique index to enforce collision protection.

Step 2: Verify Reservation
Attempt to read the record using the unique reservation code.

If the record is not found:
Exit the process as there is nothing to do.
* there are cases where this could generate a false negative when working with multiple workers. Additional functionality can be implemented to handle this.

If the record is found:
Step 3: Start Work
Set the started_at field to the current timestamp

Step 4: Do the Work
This is where the actual functionality is executed.

Step 5: Complete the Work
Set the completed_at field to the current timestamp

Repeat

Using this simple architecture you can turn almost ANY existing functionality into a multi-worker que reservation system

Here's how it works.

Step 1: Attempt Reservation. Set `reservation_code` to a unique value generated by the worker, and set `reserved_at = NOW(6)`, where `reservation_code IS NULL`. The table schema enforces the reservation code's uniqueness. The WHERE clause ensures we only reserve a row if it's available for reservation.

Step 2: Verify Reservation. Read the record using the unique reservation code. If the record is not found, exit the worker process because there is no more work to perform. If the record is found, continue to Step 3.

Step 3: Start Work. Set the `started` field to `NOW(6)`, i.e., the current timestamp.

Step 4: Do the Work. In the case of the first worker in Figure 3, the first worker splits the request into multiple recipients, creating multiple records in the second queue.

Step 5: Complete the Work. Mark the request as complete by setting `completed` to `NOW(6)`.

The worker now repeats by returning to Step 1.

Why are "reserve" and "start work" separate steps? Step 1, Attempt Reservation, could reserve a block of reservations and then work through one request at a time by looping around Step 2 through Step 5.

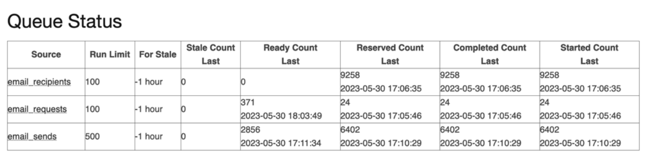
Scaling for Heavy Workload

Suppose the first worker is scheduled to run every five minutes (for example, via `crontab`). If the worker is able to finish in less than five minutes, great! If the queue remains empty five minutes later, the job will start, fail to reserve any input records, and quit.

On the other hand, if there is a heavy workload, the number of workers will ramp up. Each worker runs until the queue is empty. A new worker begins every five minutes (in this example). Thus, the system has more and more workers running simultaneously over time. Each worker is working on a separate input record.

What happens if something goes wrong? That's why it's important to design for observability. I created a simple webpage to show the email queue status. See Figure 5. I set a limit of 100 requests to be processed per worker. If a request starts but does not complete within 1 hour, I report that as a "stale" request. The "ready count" is the number of input items available for processing. When the system is idle, "reserved", "completed", and "started" should all match.

Figure 5.



Queue Status

Source	Run Limit	For Stale	Stale Count Last	Ready Count Last	Reserved Count Last	Completed Count Last	Started Count Last
email_recipients	100	-1 hour	0	0	9258 2023-05-30 17:06:35	9258 2023-05-30 17:06:35	9258 2023-05-30 17:06:35
email_requests	100	-1 hour	0	371 2023-05-30 18:03:49	24 2023-05-30 17:05:46	24 2023-05-30 17:05:46	24 2023-05-30 17:05:46
email_sends	500	-1 hour	0	2856 2023-05-30 17:11:34	6402 2023-05-30 17:10:29	6402 2023-05-30 17:10:29	6402 2023-05-30 17:10:29

Summary

We created a simple queue system using database tables. In order to provide a working example, we described (and flow-chart-ed) a business process that needs to be rewritten to take advantage of this queue system.

Next month, we'll complete the redesign, including deciding what events should be emitted for observability.

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

Exploring Real-World Applications of PHP-FIG’s PSRs

Frank Wallen

In the realm of modern web development, adherence to standards has guided developers through the complex world of code. The PHP-FIG (PHP Framework Interop Group) has taken up the mantle of shaping these standards, known as PSRs (PHP Standards Recommendations) and PERs (PHP Evolving Recommendations), with the intention of fostering interoperability and consistency within the PHP ecosystem and community. These PSRs have found practical grounding in a multitude of popular and utilitarian PHP libraries.

In this series of articles, we will take a detailed look at the PSRs, explore their practical implementations within the world of PHP development in popular libraries, and examine the benefits of adhering to these standards and how they can help developers write maintainable and scalable code.

There are many libraries and frameworks that adhere very closely to PSRs and PERs. We could spend a great amount of time on articles about some of the top projects, but we’re going to start with something a little more grounded: the Container[1] from The League of Extraordinary Packages[2]. The League of Extraordinary Packages is a group of PHP developers committed to creating powerful, useful, and re-usable libraries that adhere to standardizations put forth by PSRs, PERs, and practical approaches.

The Container

This package, installable by way of the Github repository[3] or through Composer at Packagist[4], follows PSR-11[5], the *Container Interface*. PSR-11 states, “The goal set by `ContainerInterface` is to standardize how frameworks and libraries make use of a container to obtain objects and parameters….” The minimum requirement is that it has two methods: `get` and `has`. These methods require the identifying string to represent an object (the string does not necessarily convey any semantic meaning other than being unique), value, string, an array, etc. The `get` method will return the value (throwing an exception if not found), and the `has` method will return `true` or `false` depending on whether it has been registered. PSR-11 does not define *how* dependency injection works in the developer’s application, only that the *Container* instance can register the concrete definition and return it when called to. In this case, the Container library goes well beyond the `get` and `has` by including a rich set of methods for configuring and retrieving your dependency definitions. Let’s start with a simple example of a service object with a dependency on a repository.

Repository.php

```php
namespace PsrsInAction;

class Repository implements RepositoryInterface {}
```

TheService.php

Listing 1.

```php
namespace PsrsInAction;

class TheService implements ServiceInterface
{
    protected RepositoryInterface $repository;

    public function __construct(
        RepositoryInterface $repository
    ) {
        $this->repository = $repository;
    }

    public function getRepository(): RepositoryInterface {
        return $this->repository;
    }
}
```

The following example assumes we’re following PSR-4[6] for autoloading, where we will register the two objects we need: `TheService` and the `Repository` it requires. Note the order is not necessary here, as the objects are not immediately instantiated.

```php
require 'vendor/autoload.php';

$container = new League\Container\Container();

$container->add(\PsrsInAction\TheService::class)
    ->addArgument(\PsrsInAction\Repository::class);

$container->add(\PsrsInAction\Repository::class);
```

1 https://container.thephpleague.com/
2 https://thephpleague.com/
3 https://github.com/thephpleague/container
4 https://packagist.org/packages/league/container
5 https://www.php-fig.org/psr/psr-11/
6 https://www.php-fig.org/psr/psr-4/

Now that we have registered our objects, we can retrieve them as easily as:

```
use \PsrsInAction\Repository;
use \PsrsInAction\TheService;
$repository = $container->get(Repository::class);
$service = $container->get(TheService::class);
```

When retrieving `TheService` object, it has already been instantiated with the `Repository` object due to the `addArgument` configuration we used when registering. In these examples, we used the class names as the entry id, and Container understands that the id is also the name of the concrete object we expect to get. This is just fine for dependency injection, but if you want to use dependency inversion to abstract your actual classes further, you can easily do so by defining the actual concrete object:

```
$container->add(
    ServiceInterface::class,
    TheService::class
)->addArgument(RepositoryInterface::class);
$container->add(
    RepositoryInterface::class,
    Repository::class
);
var_dump(get_class(
    $container->get(ServiceInterface::class)
));
//"PsrsInAction\TheService"
```

This way, the developer can change the concrete class that is returned, and as long as it implements the correct interfaces, it can be used without refactoring. For some, using a fully qualified class name for the id can feel cumbersome. Remember one of the qualifiers for id: "…The string does not necessarily convey any semantic meaning other than being unique…?" That means you can use a simpler string as the id:

```
$container->add('service', TheService::class)
    ->addArgument('repository');
$container->add('repository', Repository::class);
var_dump(get_class($container->get('service')));
//"PsrsInAction\TheService"
```

Notice that even the argument ('repository') we configured is using an id string. While this is readable and certainly functional, if the fully qualified class names are used, you would not need to create additional registrations in the event an `addArgument` uses it instead of a string id. For example, the following would fail:

```
$container->add('service', TheService::class)
    ->addArgument(RepositoryInterface::class);

$container->add('repository', Repository::class);

var_dump(get_class($container->get('service')));
//results in TypeError exception
```

While the repository was defined as 'repository', we never registered it to be located by the interface using the interface name. If we add an additional registration line:

```
use \PsrsInAction\Repository;
use \PsrsInAction\RepositoryInterface;

$container->add(
    RepositoryInterface::class,
    Repository::class
);
```

Our service can now be instantiated without the `TypeError` exception.

Normally, the Container will create a new object for every `get` call. If necessary for saving resources or ensuring unique instances (such as only one `TheService` can be instantiated), you would use `addShared`:

```
$container->addShared('repository', Repository::class);
$repository_1 = $container->get('repository');
$repository_2 = $container->get('repository');
var_dump($repository_1 === $repository_2);
//true
```

Remember that a container does not only return objects but can also return other values:

```
$container->add('properties', [
    'prop1' => 'foo',
    'prop2' => 'bar',
]);
$properties = $container->get('properties');
var_dump(is_array($properties)); //true
var_dump($properties['prop1']); //foo
var_dump($properties['prop2']); //bar
```

The PHP League's Container library offers much more than what we have reviewed today, as we are staying close within the scope of PSR-11. Like other modern frameworks and libraries, it offers Auto-Wiring, container delegation (for backup resolution of services), inflectors (for manipulating the object before returning it), etc. In upcoming articles in this series, we will be continuing to look at The PHP League's other excellent packages and how they support and implement PSRs.

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

Popular Tools for Robust Laravel Development

Matt Lantz

Developers rarely build anything from scratch that isn't neatly wrapped in various packages or frameworks. Developers also rarely configure a replica of a production environment to write some code. Developers have for decades built tools for other developers to help streamline their development process. Some communities can get this right where various members push the envelope and help remove hurdles from their fellow developers. Other communities have fallen very short. Let's take a look at how Laravel stacks up.

Laravel has an active community that constantly shares the latest tools and tricks for local development, among many other things. In my experience with Laravel over the last eight years, there have been some tools added to the Laravel developer ecosystem that I use daily, and I cannot imagine developing any code without them anymore. Some are paid products; some are simple CLI tools that strip out many headaches when building things quickly in a local environment.

Cli Tools

Valet

Valet has been around for a while, and most Laravel developers I have met working on a Mac use it. It seamlessly integrates with Homebrew and handles all local dependencies to get your development environment up and running quickly. Some naysayers push for Docker systems and others, but Valet enables developers working on multiple applications to develop code efficiently. If you focus on a single application, Docker tools may be a better fit, but Valet is indispensable for the developer working on numerous projects weekly.

Minio

Many, many developers work with projects deployed on AWS. You can easily configure a dummy S3 bucket to confirm that your code is working well, but that also requires you to have an active internet connection to work with it. However, if you're working for a larger firm, numerous DevOps employees are probably blocking you from simply creating dummy S3 buckets. Minio provides an incredibly powerful engine to run your own S3 buckets locally. If it fits your interests, you can deploy it on a Kubernetes environment and run your own S3 iteration on production.

In-app Tools

Some of the most handy tools for developers are the ones we can add directly to an application. Sometimes, they are community-driven, and others are part of more prominent vendors. Those handy packages you install to streamline all the development processes you can.

Debugbar

Arguably, it is the second most important tool for every developer's set. When looking for something new to work on in your application, dig deeper into DB queries, views, payloads, response times, and more. When it's time to focus on performance rather than new features, Laravel DebugBar will be your best friend. It even has a means of checking Livewire components. Discovering duplicate queries and overloading of Models in my app sections takes seconds, reducing debugging time by numerous hours.

Laravel Pint

Whether you were a previous fan of php-cs-fixer or not, the benefits of using Laravel Pint far outweigh any negatives you could come up with. Adding some simple composer script commands to your application lets you run style checks efficiently in GitHub Actions and run your style fixes when needed. The best part is that all your code styles align with standard Laravel Code components, which makes for effortless browsing through the codebase.

Desktop Applications

Helo

Created by a popular member of the Laravel community, Helo is another essential app for any Laravel developer. Rather than sending off local emails to external systems you have to visit on the web or just dump emails into logs, Helo lets you send emails to a basic local email client. You get a real-time preview of your email with deeper specs on what templates, etc., generated this email. It's a critical tool for testing email links and auth URLs before getting these elements into a staging environment. I use Helo for client work at least twice weekly; I cannot recommend it highly enough.

Tinkerwell

Most Laravel developers have heard of or have Tinkerwell. It's been around for a few years. It is a compelling desktop application that enables developers to tap into existing applications to tinker with them rather than force inject code into a random Controller and run dd. Furthermore, Tinkerwell lets you tap into deployed environments via SSH or even tap into projects deployed through Laravel Forge. All too often, it's a handy tool to help debug code problems or test various ideas without tampering directly with a project's code.

Postman or Insomnia

There are great tools like Tinkerwell, which can let you access internal code for tinkering. However, having a simple UI that can hit a variety of API endpoints quickly rather than relying solely on TDD is a must for every developer. Postman and Insomnia are very similar platforms that enable developers to hit API endpoints locally or externally and tinker with responses. I do not know any API developers who do not have one of these apps installed on their local machine.

Tableplus

There will never be enough good things to say about TablePlus. Before TablePlus, a popular database management tool was SequelPro; however, its last update, as of this writing, was in February 2016. Since then, TablePlus has become a go-to tool for developers who need a simple UI for database management or reading locally. TablePlus provides an elegant experience to explore databases locally and externally quickly. It's a robust application that gives every developer clear insights into their application data. TablePlus is often the third application I open every day.

Newest Additions:

Overall, many tools and community packages outlined above help developers with a well-running local environment. They are often icing on the cake, minus Valet. At Laracon this year, we were introduced to a new toolset for developers not keen on spending time with DevOps problems or Docker madness.

Herd

Herd is a standalone NativePHP application that can perform Valet tasks behind the scenes in many ways. As a Laravel developer, you can ignore the command line, step away from Homebrew, and let Herd handle your developer space. It may be in an early state, but I would be surprised if it didn't get significant attention from the Laravel community over the next couple of years.

Not all of these tools will fit every developer's or every project's needs. However, they can be valuable tools to resolve simple issues in a developer environment. Many developers at various stages of their careers choose to develop their own tools, which can lead to community growth. Sometimes, those tools end up on a back burner, half-cooked and forgotten. Before you invest too much time or money into tools you build yourself, take a peek at some that community members have already developed and maintain with passion.

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. @MattyLantz

It’s Time to Reinvent the Wheel

Beth Tucker Long

I hate frameworks.

Now, before the religious war breaks out, hear me out.

I love reusable code. I love the ease at which you can run ‘composer install’, and suddenly, you have a mostly working app, complete with an authentication layer and basic user CRUD. I love that there is a community of developers all over the world working with and developing for the same codebase. I love that libraries are built to enhance this shared codebase and that everyone involved follows an expected set of coding standards so the code is familiar and comfortable.

At the beginning of every new project, I love frameworks.

A few years in, though, I remember that I hate frameworks. I hate trying to force standardized code to do something non-standard. I hate spending hours and hours trying to figure out which part of the massively interconnected system is broken so I can figure out why the “magic” isn’t working anymore. I hate spending 8 hours on the 30-minute task of getting a simple form to work because there are 15 different libraries involved in making this simple form “easy” to create.

It’s at this point that my framework has become a monolith. It is so heavy and complicated that any development is painful. It has so many layers and inscrutable moving parts that debugging is a nightmare. It has become reliant on so many outside libraries and open source developers that keeping it on the latest versions of the language, platform, and software is nothing but a distant dream.

And yet, every efficiency article out there is telling us not to reinvent the wheel:

- Don’t write code that someone else has already written - save time by reusing code!
- Your application will be better if the code it uses has been vetted and debugged via a diverse open source community!
- Think about all of the new features you can leverage having a global community developing for your tech stack!

Yes, but think of all the time you could save by writing only the code needed for this specific situation. Instead of trying to force a broad solution into your narrow requirements, think of how clean, customized, and elegant your solution could be. Think how easy it would be to debug your code if you actually knew what all of the pieces were doing. Think of how much time you could spend on UX and Accessibility if you didn’t have to try to get five libraries to play nicely together to make a simple help request form.

Next time you are looking at adding a framework or library into your stack, remember: while you should not be quick to reinvent the wheel, be mindful of the long-term consequences of adding each layer of complexity.

Don’t reinvent the monolith.

Beth Tucker Long is a developer and owner at Treeline Design[1], a web development company, and runs Exploricon[2], a gaming convention, along with her husband, Chris. She leads the Madison Web Design & Development[3] and Full Stack Madison[4] user groups. You can find her on her blog (http://www.alittleofboth.com) or on Twitter @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*