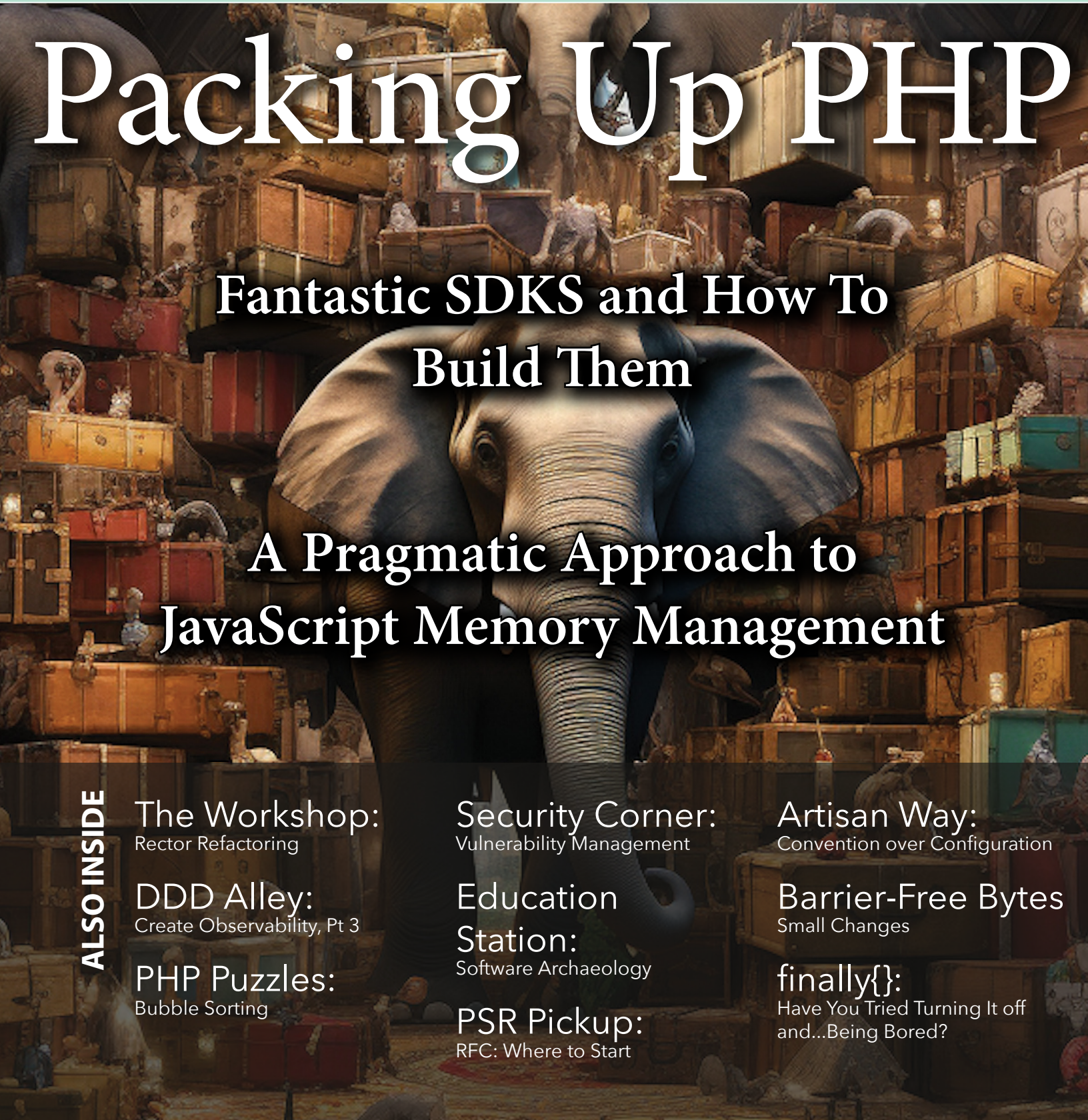
www.phparch.com

August 2023
Volume 22 - Issue 8

php[architect]

The Magazine For PHP Professionals



Packing Up PHP

Fantastic SDKS and How To Build Them

A Pragmatic Approach to JavaScript Memory Management

ALSO INSIDE

Edited with bleary eyes

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

Refresh

Jaclyn Congdon

August: the end of summer, back to school for some, harvest month for others, an ending and a beginning all in one month. It's a good time to reflect on your overall PHP development—habits you've worked to break and new ones you're trying to adopt. PHP Architect's goals are to provide the latest information from professionals in the field. Doing so allows our readers to constantly refresh their skills to weed out the things that aren't working and 'try on' some new approaches. It's a literal, personal update that goes beyond the screen.

We also explore the updating theme in this month's Education Station column by Chris Tankersley. Chris is taking us through understanding and improving legacy code in the first part of his series, *Software Archaeology*. Working with legacy code can be filled with frustration; however, Chris points out the benefits of solving the legacy code puzzle, such as building resilience and sharpening your analytical thinking.

Maxwell Ivey's column, Barrier Free Bytes, gets in on the updating/refreshing theme in his article, *Small Changes*. Maxwell explains how much learning/memorization goes into visiting a website for a person with visual impairments and the impacts of even little tweaks.

From Edward Barnard, DDD Alley brings us to the next article, *Create Observability, Part 3: Rewrite Business Process*. Again, a refresh is front and center in this article as Ed takes us through fixing/rewriting the business process to fix the code effectively. He also points out the importance of documenting ideas/decisions in a flow chart to prevent 'tribal knowledge'.

Have you wanted to help shape the future of PHP? Frank Wallen will break down the process of submitting comments and joining those PHP shaping discussions in this month's PSR Pickup, *RFC: Where to Start*. It's an opportunity to freshen up the coding language and enhance the community.

Matt Lantz is tackling *Convention over Configuration: Anti-Patterns in Laravel* in this month's Artisan Way column. This read provides an opportunity to refresh your use of the conventional framework to avoid using common anti-patterns.

Refreshing your approach to security is always a good idea. *Vulnerability Management 101* in our Security Corner column by Eric Mann will help you do just that. Eric steps us through identifying and patching any security vulnerabilities your team discovers.

If you've been following the Puzzles column by Oscar Merida, you know we've been navigating our way through mazes. This month, Puzzles gets a refresh of its own as Oscar brings us *Bubble Sorting*. Oscar will share why it's called bubble sorting and how the algorithm works with an array.

Next up, we're updating our knowledge about software development kits (SDKs) in one of this month's feature articles, *Fantastic SDKS and How To Build Them* by Steve McDougall. Steve brings us a refreshing take on SDKs by pointing out how they can be fun to build and sharing common challenges.

Rahul Kumar brings us one of this month's feature articles, *How I Learned to Stop Worrying: A Pragmatic Approach to JavaScript Memory Management*.

Rector Refactoring is our next article in The Workshop by Joe Ferguson. This article will share ways to refresh your understanding of refactoring with Rector. Joe takes us through how to configure and use Rector to improve code quality, remove dead code, and enforce coding style.

The column, Finally brings us *Have you tried turning it off and…being bored?*, by Beth Tucker Long. This article provides the ultimate refresher on how you approach your work. It's so easy to stay busy all of the time, even without realizing we're doing it. In this article, Beth shares the importance of creating downtime for yourself and, ultimately, how your work will benefit from doing so.

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/August2023_code

Fantastic SDKS and How To Build Them

Steve McDougall

I have been integrating with external APIs for longer than I can remember, and I often end up frustrated and want to look for alternative solutions. Stick with me as we look at some challenges and even explore how they can be fun to build.

So many companies auto-generate their client SDKs, leading to poor developer experience and relatively quick abandonment of the service. We have all seen these auto-generated SDKs, such as the AWS or Google SDKs. Let's look at a quick example. (See Listing 1)

So this is better than it used to be, but it still needs to be the fluid approach we are used to in our applications. Let's walk through the example above and see how we might improve it.

Instantiating the client to start with should be a simple approach, maybe something we should be able to lean on our Dependency Injection Container for.

```
// The original code
$client = new Google\\Client();
$client->setApplicationName("Client_Library_Examples");
$client->setDeveloperKey("YOUR_APP_KEY");
```

```
// Our new implementation
$client = new Google\Client(
  name: 'Client_Library_Examples',
  key: 'YOUR_APP_KEY',
);
```

We are using named parameters, a more modern language feature, and simplifying from 3 lines of logical code to what could be on one line.

Next, look at how the service is created using the Google SDK. We instantiate another class to pass the class into so that we can call an endpoint. We pass our client into the service class, so we can call a method on the service and make an HTTP request. You have to wonder why we didn't inject the client into the service using our Container or why we can't just call a method on the client to proxy through to the service itself.

Listing 1.

```
1. $client = new Google\Client();
2. $client->setApplicationName("Client_Library_Examples");
3. $client->setDeveloperKey("YOUR_APP_KEY");
4.
5. $service = new Google\Service\Books($client);
6. $query = 'Henry David Thoreau';
7. $optParams = [
8.   'filter' => 'free-ebooks',
9. ];
10. $results = $service->volumes->listVolumes(
11.     $query,
12.     $optParams
13. );
14.
15. foreach ($results->getItems() as $item) {
16.   echo $item['volumeInfo']['title'], "<br /> \\n";
17. }
```

Figure 1.

```
// The original code
$service = new Google\Service\Books($client);
$query = 'Henry David Thoreau';
$optParams = [
  'filter' => 'free-ebooks',
];
$results = $service
              ->volumes
              ->listVolumes($query, $optParams);
```

```
// Our new implementation
$results = $client->books()->listVolumes(
  query: 'Henry David Thoreau',
  parameters: ['filter' => 'free-ebooks'],
);
```

A lot cleaner and easier to use, as you can imagine. This is what is painful about auto-generated SDKs. They are

designed to take an API specification and build something easy to throw away.

Let's now walk through how we should be approaching building an SDK. For this example, we will use a fictional API to focus more on the SDK and less on the API itself.

Our example will be a simple documentation API or a note-taking API. We can create folders to organize notes and attach tags for better taxonomy and searchability. Let's look at

Listing 2.

```
GET `/folders`
POST `/folders`
DELETE `/folders`

GET `/notes`
POST `/notes`
GET `/notes/{note}`
PUT `/notes/{note}`
DELETE `/notes/{note}`

PUT `/notes/{notes}/tags`
```

a simplified view of the API structure, designed for education purposes, not for a real-life scenario. (See Listing 2)

Looking at the above, we have two main entry points that we want to be able to access from our SDK. Folders and Notes. Each performs a specific function to allow the integrator to manage their notes and folders in a way that will work best.

The main entry point should be our SDK client so that, as a user, I don't have to instantiate multiple classes to perform actions if I don't want to. Let's dive into the make-up of our client and what is essential.

```
final class Client
{
    public function __construct(
        private readonly PendingRequest $request,
    ) {}
}
```

We are building an SDK that will work with the native Laravel implementation using Guzzle. We do this because we know our API calls all go to the same URL and want to be authenticated with our accounts' API token. Let's look at how this might be bound in the Laravel container using a service provider. (See Listing 3)

At this point, we have registered a set of instructions into our Container that lets our framework know how to build our Client SDK. This is also registered as a singleton, as we want only to create one instance with a default state that we know we will want. One thing I like to do here, too, is to make this Deferrable—so that we only ever build this client when injected, meaning that our SDK does not slow down the booting of the application we are installed into.

Now that we understand how our SDK is built in the context of usage, we can think about how we want to access resources

Listing 3.

```
final class IntegrationServiceProvider
        extends ServiceProvider
        implements DeferableServiceProvider
{
    public function register(): void
    {
        $this->app->singleton(
            abstract: Client::class,
            concrete: fn () => new Client(
                request: Http::baseUrl(
                    config('service.name.url')
                )->asJson()
                 ->withToken(
                        config('service.name.token')
                ),
            ),
        );
    }

    public function provides(): array
    {
        return [
            Client::class,
        ];
    }
}
```

or services from our SDK. We want to do something similar to how the Google SDK registers its service classes.

```
$service = new Google\Service\Books($client);
```

Listing 4.

```
final class Client
{
    public function __construct(
        private readonly PendingRequest $request,
    ) {}

    public function folders(): FolderService
    {
        return new FolderService(
            client: $this,
        );
    }

    public function notes(): NoteService
    {
        return new NoteService(
            client: $this,
        );
    }
}
```

Ideally, however, we would use a proxy call through our SDK—or we want to inject this directly into our application using the Container again. Let's expand our example of the client class to see how this might work. (See Listing 4)

So we have methods on our SDK that create service classes—passing through the client to the constructor of each one. This allows us to easily inject our service class into our application—as our Container knows how to create this, or inject the client and proxy the call through the client itself.

Listing 5.

```
it('will do some test that we need our SDK for',
    function (): void {
        Http::fake([
            '*' => Http::response(
                    ['response' => 'body'],
                    Status::CODE,
                    ['headers' => 'here']
            ),
        ]);

        $sdk = app()->make(Client::class);

       $sdk->folders();
    }
);
```

Regarding testing, we can inject a mock into our service class that allows us to mock sending the request. Or the way I like to do it is to use the static fake method on the HTTP Facade so that I can do it more globally. (See Listing 5)

At this point, we can ensure we are testing things correctly using the HTTP facade we set up in our service provider.

Moving onto the service classes themselves, these are classes that we use to start calling endpoints on the API. The constructor should be a little obvious.

```
final readonly class FolderService
{
    public function __construct(
        private Client $client,
    ) {}
}
```

Listing 6.

```
$query = 'Henry David Thoreau';
$optParams = [
  'filter' => 'free-ebooks',
];
$results = $service->volumes->listVolumes(
        $query,
        $optParams
);

foreach ($results->getItems() as $item) {
  echo $item['volumeInfo']['title'], "<br /> \n";
}
```

The next step we take is to start calling endpoints on our API, which, if we examine the way the Google SDK works, is a little too cumbersome for my taste. It works, don't get me wrong. But it isn't an SDK experience I would like to use. (See Listing 6)

Something about how these larger organizations publish their PHP SDK examples feels wrong. Everything is echo this or var_dump that. There is never any real-life context, which is sad. Ideally, once we have called the endpoint, we have an iterable result instead of calling another method on another resource. This feels sloppy, and only giving an array back is lazy. We have PHP objects. Let's use them!

Listing 7.

```
$volumes = $service->volumes->list(
    query: 'Henry David Thoreau',
    parameters: [
        'filter' => 'free-ebooks',
    ]
);

foreach ($volumes as $volume) {
  // At this point we have a Volume class and
  // can call $volume->title;
}
```

Listing 8.

```
final class Client
{
    public function __construct(
        private readonly PendingRequest $request,
    ) {}

    public function folders(): FolderService
    {
        return new FolderService(
            client: $this,
        );
    }

    public function notes(): NoteService
    {
        return new NoteService(
            client: $this,
        );
    }

    public function get(string $endpoint): Response
    {
        return $this->request->get($endpoint);
    }
}
```

This is how I would have liked to see that example written: (See Listing 7)

Let's dive into our Folder Service and see what we can do to list all our folders. Firstly, we need to add one method to our Client class. (See Listing 8)

Listing 9.

```
1. final readonly class FolderService
2. {
3.     public function __construct(
4.         private Client $client,
5.     ) {}
6.
7.     public function all()
8.     {
9.         $response = $this->client->get('/folders');
10.
11.        if ($response->failed()) {
12.            throw new FailedToFetchFolders(
13.                response: $response,
14.            );
15.        }
16.
17.        // Process payload and turn it into objects.
18.    }
19. }
```

Because our request on the client is private, we cannot access it through the service class itself. This is by design, though, as the request has data we do not want to be passing around our application, such as the API key. We keep that where it is, immutable to the Client only. (See Listing 9)

We are handling the potential exception and passing back a contextual exception—allowing us to know what happened. The next step after this is to map over the response and create objects that represent what we expect. Passing back an array of data at this point is ok, but it isn't what I call a pleasant experience for an SDK—which is what we aim for.

I have written about this part before—mapping responses to objects. Like when building an API, you want to transform the database values to something consistent and slightly abstracted from your database for the response. It is a great topic, and there are plenty of opportunities for optimizations here using Object Hydration—but I will stick to a simple mapping approach.

Imagine a payload coming back from our API for our folders. These folders have a name, an icon, a boolean flag to mark it as a parent folder, and an identifier for its parent folder if it is a sub-folder. (See Listing 10)

This is a somewhat simple structure, but something we could utilize with Value Objects easily. So, the first thing to remember is that with the Laravel HTTP client, we have access to the `json` and `collect` methods, depending on what we need to do. In this instance, the `collect` method will help us to map over the response data and return a collection of Value Objects instead of simple arrays. Let's look at our Value Object quickly. (See Listing 11)

Listing 10.

```
1. {
2.   "data": [
3.     {
4.       "id": "1234",
5.       "type": "folders",
6.       "attributes": {
7.         "name": "Folder name",
8.         "icon": "x-icon-name",
9.         "parent": false,
10.        "parent_id": "4321"
11.      }
12.    }
13.  ]
14. }
```

Listing 11.

```
1. final readonly class Folder
2. {
3.     public function __construct(
4.         public string $id,
5.         public string $name,
6.         public string $icon,
7.         public bool $parent,
8.         public string $parentId,
9.     ) {}
10.
11.    public static function fromResponse(
12.        array $data
13.    ): Folder {
14.        return new Folder(
15.            id: $data['id'],
16.            name: $data['attributes']['name'],
17.            icon: $data['attributes']['icon'],
18.            parent: $data['attributes']['parent'],
19.            parentId: $data['attributes']['parent_id'],
20.        );
21.    }
22. }
```

Later on, we can start working with this Value Object to add more to it, such as pre-loading in the parent if we have a parent, loading in a component name for our icon, etc.

In our service class, we can now utilize our Value Objects and add a return type to our method. (See Listing 12 on the next page)

This is a somewhat simple approach to achieving what we need, assuming the response returns in a specific format. However, if this is our API, then we can guarantee this with more certainty than we could if it weren't. What I would typically do at this stage is to define the array shape on the Value Object so that I get static analysis errors when running my checks. This would look like the following: (See Listing 13 on the next page)

This is not something you need to do—however, it will help with code auto-completion in the IDE when integrating with the SDK. Where possible, always add as many types as

Listing 12.

```
final readonly class FolderService
{
    public function __construct(
        private Client $client,
    ) {}

    public function all(): Collection
    {
        $response = $this->client->get('/folders');

        if ($response->failed()) {
            throw new FailedToFetchFolders(
                response: $response,
            );
        }

        return $response->collect('data')->map(
            fn (array $folder) =>
                        Folder::fromRequest($folder)
        );
    }
}
```

Listing 13.

```
/**
  * @param array $data{
  *       id:string,
  *       type:string,
  *       attributes:array{
  *           name:string,
  *           icon:string,
  *           parent:bool,
  *           parent_id:string
  *       }
  *  } The incoming data from the request
  *  @return Folder
  **/
public static function fromResponse(
    array $data
): Folder
```

you can—it really does help with developer experience when working with third-party libraries.

Now that we know how we can make a get request, we want to look at the other side of the coin. The beauty of building an SDK specific for something like Laravel is that you can take advantage of Laravel's amazing Validation library without pulling your own in. Let's add a post method to our client now, and look at how we might work with validating data before sending an API request. (See Listing 14)

We can add these "convenience" methods to our client, allowing the person using our SDK to abstract and implement their approach should they need to. Let's look at creating a new folder next. We will scaffold this without the rest of the class to narrow our focus.

Listing 14.

```
final class Client
{
    public function __construct(
        private readonly PendingRequest $request,
    ) {}

    public function folders(): FolderService
    {
        return new FolderService(
            client: $this,
        );
    }

    public function notes(): NoteService
    {
        return new NoteService(
            client: $this,
        );
    }

    public function get(string $endpoint): Response
    {
        return $this->request->get($endpoint);
    }

    public function post(
        string $endpoint,
        array $data
    ): Response {
        return $this->request->post($endpoint, $data);
    }
}
```

```
public function create(NewFolder $payload): Folder
{
    // first we want to validate our value object
    // then we want to look at transforming our
    // value object into an array finally we can
    // send the request and work with the response.
}
```

Looking at the above code, it is clear that we will have a separate Value Object to create our Folder. Let's create this, with the additional methods we will require. (See Listing 15 on the next page)

As you can see in the validate method, we return whether or not the validation fails. We could flip this boolean here or rename the method, but this is more down to you and what makes sense in your library. The beauty of using the Laravel validation library here is that we can tap into the validation using custom validation error messages and multi-lingual support in our SDK if we want to. The possibilities are somewhat endless here.

Let's return to our service class now and see how we might use this code. (See Listing 16 on the next page)

As you can see above we validate the payload and throw an exception that makes sense at the time—even if it is something like an Invalid Argument Exception at this stage. We

Listing 15.

```php
final readonly class NewFolder
{
    public function __construct(
        private string $name,
        private string $icon = 'x-folder-icon',
        private null|string $parentId = null,
    ) {}

    public function toArray(): array
    {
        return [
            'name' => $this->name,
            'icon' => $this->icon,
            'parent_id' => $this->parentId,
        ];
    }

    public function validate(): bool
    {
        return Validator::make($this->toArray(), [
            'name' => [
                'required',
                'string',
                'min:2',
                'max:255',
            ],
            'icon' => ['required', 'string'],
            'parent_id' => ['nullable', 'string']
        ])->fails();
    }
}
```

Listing 16.

```php
public function create(NewFolder $payload): Folder
{
  if ($payload->validate()) {
        // throw a validation exception here or a
        // custom domain-specific exception.
  }

  $response = $this->client->post(
        '/folders',
        $payload->toArray()
  );

  if ($response->failed()) {
    throw new FailedToFetchFolders(
      response: $response,
    );
  }

  return Folder::fromRequest($response->json('data'));
}
```

then send the request, passing in the Value Object being turned into an array. You could use serialization or many other clever ways to turn a value object into a request payload here; however, keeping it simple like this gives me the most control when working with the data. I can ensure that the fields align with the API's expectations, and treat it almost like a transformer for my outgoing API requests.

Let's recap. We started with manual instantiation from the Google PHP SDK and considered why this doesn't work. I may have moaned several times about auto-generated Client SDKs, especially in the PHP space. We looked at what we would do if we approached this as an actual person, how we might improve the Developer Experience, and potentially the developer engagement. We then looked at a fake API we could use as an example, and looked to see what we could do if we built an SDK.

The only thing left to cover at this stage is sub-resources. Luckily, we only have one sub-resource available in our fake API—tags. So, we want to create another service class that can understand the context of its use—without making the usability terrible.

Listing 17.

```php
final class SyncTagsCommand extends Command
{
    protected $signature =
   'tags:sync { note : The Identifier for the note. }';
    protected $description =
                    'Sync tags attached to this note.';

    public function handle(Client $client): int
    {
        // work on collecting the tags to sync.
        // The tags will be stored as $tags
        // We have the argument from the CLI as
        //      $note for the identifier
        try {
            $client->notes()->tags($note)->sync($tags);
        } catch (Throwable $e) {
            $this->components->error($e->getMessage());

            return Command::FAILURE;
        }

        return Command::SUCCESS;
    }
}
```

What I like to do when building SDKs, is typically to design the API I want first and see how I can make that possible without the code being terrible. So for this last part, let's do that. Imagine we are injecting this SDK into a CLI command. (See Listing 17)

Listing 18.

```php
final readonly class TagService
{
    public function __construct(
        private Client $client,
        private null|string $note = null,
    ) {}

    public function sync(array $tags): Collection
    {
        // Here we would send the request through to
        // the client as we usually would do.
    }
}
```

Listing 19.

```php
final readonly class NoteService
{
    public function __construct(
        private Client $client,
    ) {}

    public function tags(string $note): TagService
    {
        return new TagService(
            client: $this->client,
            note: $note,
        );
    }
}
```

So the behaviour I want to achieve is:

```php
$client->notes()->tags($note)->sync($tags);
```

This means our Tags service class needs another parameter, which we want to be optional in case we ever want to be able to resolve it directly in the future. (See Listing 18 on the next page)

Not too different from our other service classes. So we can instantiate this in our Note Service easily. (See Listing 19)

The beauty of this approach is that if we want to use tags as a stand-alone service class, we just don't pass in the note—it just means that we cannot use the sync method without adapting it at this stage.

SDKs do not have to be difficult, in-fact, they can be quite fun to build and present some interesting challenges along the way. While it is easy to auto-generate them, you really shouldn't be. Thanks for reading.

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*

a php[architect] guide

Learn how to build dynamic and secure websites.

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

Available in Print+Digital and Digital Editions.

Purchase Your Copy

https://phpa.me/php-development-book

How I Learned to Stop Worrying: A Pragmatic Approach to JavaScript Memory Management

Rahul Kumar

Have you ever felt overwhelmed by the complexity of memory management in JavaScript? Between closures, event handlers, and variable scoping, it can feel like there are a million ways your app might start leaking memory without you realizing it. But memory management in JavaScript doesn't have to be complicated. In fact, if you follow a few simple, pragmatic principles, you'll be well on your way to building stable apps with clean memory usage.

This guide will walk you through the core concepts of JavaScript memory management in a straightforward, hands-on way.

Let's dive in!

My Struggles with Memory Management As a Beginner

When I first started learning JavaScript, memory management confused the heck out of me. I just couldn't grasp all the rules around scope, closures, and garbage collection. My code was plagued with memory leaks and unexpected behavior.

If this sounds like you, don't worry—you're not alone. Memory management in JavaScript can be tricky to pick up. Here are a few of the struggles I faced as a beginner and the lessons I learned along the way:

Forgetting to remove event listeners

This was a big one for me. If you attach an event listener to an element in the DOM but never remove it, that function will persist in memory even after you remove the element. Make sure to call `.removeEventListener()` to clear event listeners and avoid leaks.

Not understanding scope

I didn't fully grasp that variables declared with `let` and `const` are block-scoped, while `var` is function-scoped. This led to accidental global variables and closures I didn't intend. Take time to learn the rules of scope in JavaScript to avoid unwanted side effects in your code.

Misusing closures

Closures are a powerful feature of JavaScript, but they can also accidentally create memory leaks if misused. Any variables within the closure will remain in memory as long as the closure exists. Be careful what you're enclosing in closures to avoid leaks.

With regular practice, these concepts will become second nature.

Core Concepts: the Heap, Stack, and Garbage Collection

When you first start working with JavaScript, memory management can seem like a mystery.

How does the heap work? What's the difference between the stack and the heap? Why do we need garbage collection?

Don't worry, these core concepts are actually quite straightforward once you understand them.

The stack and the heap are the two places memory is allocated in JavaScript. The stack stores primitive types like **strings**, **numbers**, and **booleans**. It's very fast to allocate and access data on the stack. The heap, on the other hand, stores reference types like objects and arrays. *Accessing data on the heap is slower than the stack.*

Now, the heap is where things get tricky. When you create an object in JavaScript, memory is allocated for it on the heap. The *object is given a reference that points to its location in memory. If there aren't any references pointing to an object, it becomes "garbage"* and needs to be cleaned up. This is where garbage collection comes in.

The garbage collector periodically checks for objects on the heap without references pointing to them. It then frees up those objects' memory, making them available for new objects. This helps prevent your application from slowing down or running out of memory over time.

Understanding how the heap, stack, and garbage collection work together is key to writing optimized JavaScript code.

Some Resources to help:

- Javascript Fundamentals — Call Stack and Memory Heap[1]
- A Guide to Heaps, Stacks, References and Values[2]

1 https://phpa.me/medium

2 https://phpa.me/fjolt

Common Memory Leaks and How to Avoid Them

As you build more extensive and complex JavaScript applications, memory leaks become an increasingly annoying problem. But don't worry—with some diligence and the right techniques, you can plug most memory leaks.

Here are a few of the most common causes of memory leaks and how to avoid them.

1. Forgotten timers or event listeners

Have you ever set up a timer or event listener in your code and then forgotten about it? These are easy mistakes to make and can cause memory leaks. Always make sure to clear your timers and remove event listeners when they are no longer needed.

```
let timer = setTimeout(() => { / ... / }, 1000);
// Later...
clearTimeout(timer); // Clear the timer
```

2. Unreleased DOM references

If you attach event listeners or store references to DOM elements and don't remove them, the browser will keep the referenced elements in memory even after they are no longer displayed. Be sure to remove unused DOM references by calling `removeEventListener()` and setting references to `null`.

3. Closures

Closures are a key part of JavaScript, but they can also easily lead to memory leaks if you aren't careful. Any *variables within the closure scope are kept in memory* for as long as the closure exists. Make sure closures do not maintain references to unused objects, or the closure will prevent garbage collection of those objects.

4. Global variables

Any objects assigned to global variables will remain in memory for the lifetime of your webpage. Avoid assigning objects to global variables, and instead, pass references to objects as function arguments to limit their scope.

Keeping this common memory leak causes in mind as you build your applications will help ensure smooth performance and a good user experience.

Managing Memory with Primitives Vs. Objects

Memory management in JavaScript can seem tricky, but it's quite pragmatic when you boil it down to the fundamentals. As a developer, you really only need to worry about two types of values - primitives and objects.

Primitives Primitive values in JS are:

- Strings
- Numbers
- Booleans
- Null
- Undefined

Primitives are stored and managed directly on the stack. This means when a primitive value goes out of scope (the function it's declared in ends), it's deleted. Easy!

```
function doSomething() {
let name = "Rahul";
}
// name is undefined here
```

Once the `doSomething` function ends, the primitive string "Rahul" is removed from memory.

Objects

Objects (including arrays) are stored and managed on the heap. This means even when they go out of scope, they aren't immediately deleted. The memory on the heap needs to be manually allocated and deallocated.

```
function createObject() {
    let obj = { name: "Rahul" };
}

// obj is still in memory here!
```

Even though the `createObject` function ended, the object {name: "Rahul"} still exists in memory. This is known as a memory leak since that memory is inaccessible but not released. To fix this, you need to dereference the object by setting it to null:

```
function createObject() {
    let obj = { name: "Rahul" };
    obj = null; // Dereferencing the object
}
```

Now the memory for that object can be reclaimed by the garbage collector.

Keep it simple; focus on the fundamentals. It is very important to understand how primitives and objects are

stored and handled in memory to master Memory Management in JS.

The Tricky Parts: Closures, Callbacks, and References

So you've learned the basics of JavaScript memory management—how objects and variables are stored in the heap, referenced by pointers in the stack, and eventually (hopefully!) cleaned up by the garbage collector when no longer needed.

But there are a few tricky parts that can trip up even experienced devs.

Let's take a look at some of the weird ways memory can leak in your JS code.

Closures

Closures are functions inside functions that remember their lexical scope even when the outer function has finished executing. This can be useful but also poses a memory management issue.

If a closure retains a reference to variables in the outer scope, those variables cannot be garbage collected even after the outer function finishes running. (See Listing 1)

Listing 1.

```
1.  function makeAdder(a) {
2.      return function(b) {
3.          return a + b;
4.      };
5.  }
6.
7.  var add5 = makeAdder(5);
8.  var add10 = makeAdder(10);
9.
10. console.log(add5(2)); // Prints 7
11. console.log(add10(2)); // Prints 12
```

Here, the closures `add5` and `add10` maintain a reference to the variable in the outer `makeAdder` function scope. So these variables will not be garbage collected after `makeAdder` finishes executing, creating a memory leak.

To fix this, you can avoid closures or have them release references when no longer needed.

Callbacks

Similar to closures, callbacks can also inadvertently create memory leaks by maintaining references to variables that would otherwise be eligible for garbage collection.

```
function doSomething(callback) {
    var a = 1;
    callback();
}

doSomething(function() {
    console.log(a);
});
```

Here, the callback function maintains a reference to the `a` variable, so it cannot be garbage collected after `doSomething` finishes executing. To prevent this, you can avoid passing callbacks unnecessary references or have the callbacks release them when done.

Reference Cycles

Objects that reference each other in a circular fashion cannot be garbage collected because they will always have at least one reference pointing to them. This is known as a reference cycle and will steadily leak memory if not resolved.

```
var obj1 = {
    obj2: {
        obj1: obj1
    }
};
```

`obj1` and `obj2` reference each other in a cycle, so neither can be garbage collected. To fix this, you can have one object release its reference to the other when no longer needed.

```
var obj1 = {
    obj2: {
        obj1: obj1,
        releaseObj1: function() {
            obj1 = null;
        }
    }
};
```

By setting `obj1` to `null` within the `releaseObj1` function, you break the reference cycle, allowing the objects to become eligible for garbage collection when they are no longer needed.

My Go-to Tools for Diagnosing Memory Issues

Once you start building larger JavaScript applications, memory management becomes an important consideration. As your app grows, it can become difficult to track and fix where memory leaks are occurring. Here are some tools I generally use:

1. The Chrome DevTools Memory Panel

The Memory panel in Chrome's DevTools[3] is invaluable for tracking down memory leaks in your frontend React code. You can take heap snapshots at different points in your app to compare memory usage over time. The panel will show you which components are using the most memory so you can

3 https://developer.chrome.com/docs/devtools/

optimize or fix them. You can also check for detached DOM nodes, event listeners that weren't properly cleaned up, and much more.

2. Node.js Heapdump

For my Node.js backend, I use the heapdump module[4] to generate heap snapshots. This gives me a snapshot of the heap at a specific point in time so I can analyze memory usage in my Express server or other Node processes. I'll often take multiple snapshots[5] at strategic points, like when a server starts up, during periods of high traffic, and just before shutdown. Comparing the snapshots helps identify where memory leaks are happening over the lifetime of the process.

3. Visual Studio Code Memory Viewer

The Memory Viewer extension for VS Code[6] is great for analyzing heap snapshots on the fly. You can view memory usage over time, see which objects and functions use the most memory, check for detached DOM nodes or event listeners, and filter the snapshot by type. This makes it easy to zoom in on potential problem areas without having to export your snapshots to a separate analyzer tool.

Keeping a close eye on memory usage and optimizing where needed has allowed me to build faster, more stable apps with fewer crashes.

Optimizing Code for Memory Efficiency

Once you have a solid grasp of JavaScript's memory management model, optimizing your code for memory efficiency is the next step. A few simple techniques can go a long way toward improving your app's performance and reducing the chance of memory leaks.

Reuse variables and objects

Instead of creating new variables and objects each time through a loop or function, reuse them.

```
// Less Efficient: Creating new objects within
// each iteration
for (let i = 0; i < 10; i++) {
    let obj = {
        prop: i
    };
    // Perform operations using 'obj'
}
```

Doing so creates 10 new obj objects, using up memory. A better approach is:

```
// More Efficient: Reusing a single object
// across iterations
let obj = {};
for (let i = 0; i < 10; i++) {
    obj.prop = i;
    // Perform operations using 'obj'
}
```

Now only one obj is created and reused.

Clean up event listeners and timers

If you attach an event listener or set a timer in your code, make sure you remove it when it's no longer needed.

```
// Attaching an event listener
let el = document.querySelector('.some-element');
el.addEventListener('click', handleClick);

// When the event listener is no longer needed
el.removeEventListener('click', handleClick);
```

Failing to remove event listeners and timers can lead to memory leaks in your web app.

Minimize DOM queries

Repeatedly querying the DOM is inefficient and can bog down your web app. Store DOM elements, arrays, and objects in variables and update them as needed rather than continually re-querying the page.

```
// Efficient DOM querying

let items = document.querySelectorAll('.list-item');

// Utilize the 'items' array multiple times

items[0].textContent = 'Hello';
items[1].textContent = 'World';
```

Rather than:

```
// Inefficient DOM querying

let item = document.querySelector('.list-item');

item.textContent = 'Hello';
item.textContent = 'World';
```

Querying the DOM twice when you can do it once wastes resources.

By following these techniques, you'll be well on your way to optimizing your JavaScript code for memory efficiency.

Best Practices for Responsible Memory Usage

Once you have a solid grasp of how JavaScript handles memory allocation and garbage collection, it's time to put that knowledge into practice. Following a few best practices will help ensure your web apps are responsible memory users.

4 https://www.npmjs.com/package/heapdump

5 https://phpa.me/nodejs

6 https://phpa.me/marketplace

A. Avoid Memory Leaks

A memory leak occurs when your code unintentionally holds onto references of objects you no longer need, preventing the garbage collector from freeing up that memory. Some common causes of leaks are:

- **Closures**: When a function "closes over" variables in the surrounding scope, those variables are kept in memory as long as the function exists. Be careful what you capture in closures.
- **Timers**: Any variables captured in a setTimeout or setInterval callback will be kept in memory until the timer completes. Cancel timers when you no longer need them.
- **DOM References**: Holding onto DOM elements via JavaScript variables prevents their memory from being reclaimed. Remove event listeners and null out variables when done.
- **Circular References**: Objects that directly or indirectly hold references to each other cannot be garbage collected. Break circular references by nulling child or parent references.

B. Minimize DOM Manipulation

The DOM is memory intensive. Whenever possible, minimize reading from and writing to the DOM. Some optimizations include:

- Build up DOM strings and append them all at once rather than appending incrementally.
- Cache DOM queries in variables rather than re-querying the same elements repeatedly.
- Use `documentFragment.appendChild()` to build up DOM nodes in memory before adding them to the page.
- Use `classList.add()/remove()` instead of `className` manipulation, which requires a DOM read and write per change.

C. Be Mindful of Scope

The scope of a variable determines how long it will stay in memory. Variables declared with `let` and `const` are block-scoped, meaning they are destroyed when the block they are declared in exits. `var` is function-scoped, staying in memory for the life of the function.

Minimize the scope of variables when possible so they can be garbage collected sooner. Declare variables as close as possible to where they are used. And be careful *not to capture variables in closures* that you do not need to keep in memory long-term.

Following these best practices will make you a responsible JavaScript memory user and help avoid performance issues in your web apps.

FAQ: Answering Common Memory Management Questions

So you've learned the basics of JavaScript memory management but still have some lingering questions. Not to worry, we've got you covered. Here are answers to some of the most frequently asked questions about managing memory in JavaScript.

How do I know if my web app has a memory leak?

Some signs your web app may have memory leaks include:

- **The app slows down over time.** As more memory is leaked, the app has to work harder to function properly.
- **JavaScript heap size increases but doesn't decrease.** You can check this in your browser's DevTools.
- **Unused variables, event listeners, or DOM elements stick around.** These should be deleted when no longer needed.

To track down memory leaks, use the Chrome DevTools to monitor JS heap size and see which objects are accumulating over time. Then you can look for ways to delete them when no longer needed.

How often should I check for memory leaks?

It's a good idea to check for memory leaks in your web application regularly using the Chrome DevTools. Especially when:

- Adding new features or making major changes to existing code.
- Optimizing performance. Memory leaks can slow down your web app over time.
- Debugging or troubleshooting issues. Memory leaks are often the culprit behind strange behavior.

A quick check using the Memory panel in DevTools can help catch issues early and ensure your web app is running as efficiently as possible.

How often should I clear timeouts and intervals?

Any time you set a timeout or interval in JavaScript, it's good to clear it when it's no longer required. Some use-cases:

- When a user navigates away from the page.
- When a modal or other component using the timer is closed.
- Whenever the timer is no longer necessary for your web app's functionality.

Failing to clear timeouts and intervals can cause memory leaks. As a rule of thumb, always clear any timers in the corresponding cleanup function for the webpage or component.

Do I need to worry about memory management with third-party libraries?

Third-party libraries can often be a source of memory management issues in web applications. Some best practices include:

- Check the library's documentation for any memory management considerations. The authors should provide guidance on proper use.

- Test the library at scale to ensure there are no major memory leak issues before deploying in a production web app.
- Consider wrapping the library in your own component that properly handles cleanup to avoid leaks.
- Report any memory leaks or inefficient usage you discover to the library authors.
- Keep libraries up to date as updates often include performance and memory optimization improvements.

How can I manually trigger garbage collection?

You can't force garbage collection in JavaScript. The engine decides when to run the garbage collector based on certain heuristics. However, you can encourage garbage collection by nulling out references to objects you no longer need:

```
let obj = { /* ... */ };
// 'obj' is now eligible for garbage collection
obj = null;
```

Setting obj to null eliminates the reference, making the original object eligible for garbage collection. But again, you can't force the engine to run the GC at that exact point in time.

Conclusion

So there you have it, a practical approach to tackling memory management in JavaScript without pulling your hair out. By understanding the core concepts, implementing a few best practices, and using the tools built into the language, you'll be writing cleaner code and avoiding frustrating bugs in no time. You'll be optimizing components, cleaning up unused references, and still sleeping soundly at night.

Happy Coding!

I am an 18-year-old technical writer and developer. I have a deep passion for writing, coding, and everything in between. Over the past few years, I've written over 500 articles for developers, sharing my knowledge and expertise with others. Currently, I am working on two exciting projects. The first is Fueler.io[7], a platform for knowledge workers to create proof of works. The second is Learnn.cc[8], an online education platform that helps people learn new skills and advance their careers. @rahul_wip

7 http://fueler.io

8 http://learnn.cc

Software Archaeology - Part 1

Chris Tankersley

Imagine discovering an ancient city, its paths and alleys winding, its structures fascinating, and its history a mystery. You are here by your own accord, but your guides have disappeared. You have no map. You know where you need to go but have no idea how to get there. It is intriguing, and you take your first steps into the city. Now, replace the city with a vast, complex PHP codebase you have never seen. The excitement, curiosity, and intricate puzzles remain the same. For developers, this happens all of the time. Whenever any of us get a new codebase, no matter how well the codebase is "documented", there are always questions. No matter how many coworkers you have, the codebase hides secrets that you must suss out. Welcome to the world of Software Archeology—a practice that marries exploration, detective work, and technical prowess.

What is Software Archeology?

Software Archeology goes beyond mere code reading. It's about diving into the psyche of previous developers, understanding their decisions, design choices, and even quirks. With PHP's rich history, this becomes an adventure through different coding eras.

Legacy code in PHP is not a rare sight—it's a living testimony to the language's robustness and widespread adoption over the years. With PHP's journey from simple scripting in personal homepages to powering substantial parts of the web, there lies a treasure trove of applications, libraries, and systems developed using various versions and paradigms. Unearthing this complexity is where Software Archeology becomes a critical skill.

PHP's ease of use and flexibility led to its widespread adoption, especially in the early days of web development. Many of these applications continue to function, serving crucial roles in businesses, institutions, and platforms. Reviving or maintaining these codebases requires a deep understanding of practices that might seem archaic compared to modern PHP standards.

Legacy code is not a designation based on the age of the codebase. There is an argument that any code becomes legacy as soon as it is finished, but even more so as developers leave and move on. We always go on about the bus factor of code, but we rarely talk about the inevitability of people leaving and being unable to pass on knowledge.

Why is Legacy Code So Bad in Php?

A common challenge in managing legacy code is the loss of expertise. Developers who initially crafted the code may have moved on, leaving behind undocumented wisdom and hidden intricacies. This knowledge drain creates a vacuum where understanding why certain decisions were made or how specific functionalities work becomes daunting. The detective work of Software Archeology helps bridge this gap, uncovering the hidden rationale and making sense of the code's inner workings.

With the evolution of PHP, many functions have been deprecated, and new features and standards have emerged. Legacy code might be riddled with practices considered insecure or inefficient today. Understanding and adapting to these changes is not just about replacing outdated functions; it's about embracing the code's history and gradually guiding it toward modern practices.

Maintaining and upgrading legacy code is often more economically viable than building from scratch, especially when the existing system serves complex business logic. PHP's wide range of legacy applications supports many industries, and ensuring their seamless operation is crucial. Software Archeology, in this context, becomes a vital business skill, enabling developers to enhance, secure, and adapt legacy systems to contemporary needs.

The Code Makes It Harder

Legacy code is often likened to archaeological ruins, and for a good reason: the deeper you dig, the more complex structures you unearth. Not only does knowledge leaving a team impact your ability to figure out and understand code, but sometimes the code can just be less than fantastic.

I always like to joke that I am a great programmer, but me six months ago had no idea what he was doing. PHP developers contend with the language itself changing routinely, but also we, as developers, are getting better too. Design decisions we made at the time may no longer make sense from a business or even a knowledge perspective.

Spaghetti Code: Entangled Logic and Presentation

"Spaghetti Code" is a term used to describe code where the logic, presentation, and data handling are all tangled together, resembling a plate of spaghetti. Here's why it's problematic:

- **Mixing Presentation with Logic**: Code that intertwines HTML and PHP is hard to follow and

maintain. The coupling of different concerns makes modifications risky and debugging perplexing.

- **Unintentional Side Effects**: Functions may have hidden dependencies or unexpected consequences. Changing one part might inadvertently affect unrelated areas, leading to bugs that are hard to track down.

Consider this piece of code:

```
function fetchDataAndRender($id) {
    global $db;
    $query = "SELECT * FROM users WHERE id = $id";
    $result = $db->query($query);

    echo "<table>";
    while($row = $result->fetch_assoc()) {
        echo "<tr><td>" . $row['name'] . "</td></tr>";
        logActivity($id); // Side effect
    }
    echo "</table>";
}
```

There is a lot going on in these simple twelve lines of code. At a basic level, this function outputs HTML, which means we are mixing the presentation with logic. What happens if we want to re-use this function on another page but need a different layout? There may be a temptation to pass in a second parameter that allows us to add custom CSS classes.

We also pull in a global variable. This function is now directly tied to the status of the database, and if we change any of the database schemas, we may need to modify this function. Along those same lines, we also depend on the logActivity() function which probably relies on some sort of logging backend.

The older a codebase is, the more chance small changes like these work their way into innocuous places.

Dirty Code

Often, navigating legacy code means venturing into mazes of nested conditionals, loops, and tangled logic. These structural code problems not only make the code more challenging to read but also introduce a higher risk of bugs and maintenance difficulties. Let's dive into some of the common issues:

Deeply Nested Conditionals

As logic in a codebase becomes more complex, we have a tendency to break down conditionals to make it easier for us to read. In many cases, this ends up with deeply nested conditionals, where we put if-else blocks inside of other if-else blocks. The logic for these structures can be very hard to follow or even break down exactly what happens at each branch. (See Listing 1)

God Objects and Methods

"God Objects" are objects that know or do too much. Think of an object that is called quite a bit in a codebase, usually statically, for data like configuration values or database access. In many cases, these objects start with the best intentions, but

Listing 1.

```
1. if ($user) {
2.       if ($user->isActive()) {
3.            if ($order->hasItems()) {
4.                 // ...
5.            } else {
6.                 // ...
7.            }
8.       } else {
9.            // ...
10.      }
11. } else {
12.      // ...
13. }
```

as the object is called in more and more places, developers tend to add more logic to them. Need to make it easy to use a templating engine? Throw it into the God Object.

Much like with spaghetti code, the chance for side effects increases as more responsibility is added to a God Object. The application begins to rely on these side effects happening. Increasingly, understanding the overall logic flow becomes harder and harder.

Cyclic Dependencies

Cyclical dependencies happen when one object relies on another but when that second object also relies on the first object. This can be quite subtle, but it joins different functions or classes together in odd ways. You may not even notice that it is happening until you understand the codebase even more. (See Listing 2)

Listing 2.

```
1. class Order {
2.     private $customer;
3.
4.     public function __construct(Customer $customer) {
5.         $this->customer = $customer;
6.         $this->customer->setOrder($this);
7.     }
8. }
9.
10. class Customer {
11.     private $order;
12.
13.     public function setOrder(Order $order) {
14.         $this->order = $order;
15.     }
16. }
17.
18. // Usage
19. $customer = new Customer();
20. $order = new Order($customer);
```

Changing either class can have unintended consequences, but it can become confusing when trying to understand the code. Since PHP passes objects around by reference, we can

get some unexpected consequences and side effects. You may be trying to look at a block of code trying to figure out why an order is all of a sudden appearing on a customer object only to find out that the customer object is being used in another class (Order) that is being "helpful" by wrapping up the setOrder() logic.

Magic Numbers and Strings

Hardcoded values are one of the most annoying things to run across in a codebase. It can be even more frustrating when an application has a configuration section or file, but not everything is in it. It is easy for things like URLs to be hardcoded for many libraries, but things get worse when values get hardcoded with the assumption that they will never change.

Sure, it makes sense today that all your currency work is done in the US Dollar, but as soon as you move to a country that does not use the dollar sign, you slowly find out everywhere you used the dollar sign.

Even worse is when a value moves from being hardcoded to being configurable, and not all of the places it was used were updated. Now you have to figure out where the hard-coded value came from and the best way to get the configurable version in its place.

Let's Start Investigating

The process of familiarizing yourself with the code structure is like piecing together a jigsaw puzzle. Each discovery brings clarity, forming a mental map that guides your subsequent investigations and alterations. It's a phase that requires patience, curiosity, and attention to detail, laying the foundation for a successful Software Archeology expedition.

Dig Through What is Left Behind

It may seem a bit obvious, but the first thing to do is to see what has been left behind. This can include any existing documentation in the form of user manuals, public documentation, and what little code comments may still exist. If there is source control, the source control logs can give some insight into what the developers were intending and what code they were working on at the time.

If accessible, engage with previous developers or users who might still be around. First-hand insights from those who've interacted with the code can be invaluable, bridging gaps that documentation or code analysis might miss. Many times a decision may only be remembered by one person on another team or a manager who may have requested the change because of a product decision.

Understand the Project Structure

One of the benefits of PHP is that there are a few common entry points for applications. For newer applications, this may mean an `index.php` file that is creating and running a framework. Even if this framework is a custom one, finding the initial bootstrap for the framework can give you the first place to start investigating.

If it is an older application, look for different `index.php` files that may exist in other folders. It was not uncommon for clean URL structures to be emulated by folder structures that take advantage of calling `index.php` automatically. Try and match up the URL structure of the application with the physical structure of the application.

While you are trying to match things up, try and understand what the structure of the application is. Does this application use the Model-View-Controller (MVC) paradigm? Is it purely page based? Sometimes the class names, file names, or the content of different classes or files can give an indication of what the intent of the code is. There may not be any real technical structure, but it never hurts to look.

As you dig around, look for those core files that make the application tick. If you are lucky enough to be working with an application that uses an established framework like WordPress, Drupal, or Laravel, look in the places their own documentation points to as being important. If the code is a custom application, look for files that are being included multiple times or even just sound important.

Often, legacy PHP projects house configuration settings in dedicated files or global arrays. Analyzing these can give insights into the system's behavior, including database credentials, environment settings, or feature flags. Try and find in the code where these configuration values are being used so that you can match up features with potential code.

As you are going through the code, make notes. If you are lucky enough to have existing technical documentation, consider augmenting it with the things you learn as you read into the code. If you have questions, any new members of the team will also have these questions. If there is no documentation, set up a wiki or a document to start putting your notes into and make it public to the team. Try and get other people on the team to contribute.

Embrace the Challenge

Facing a complex, legacy PHP codebase can be a daunting experience, especially if you're a junior to mid-level developer. The lines of code, some written long ago, might seem like a tangled web waiting to ensnare anyone brave enough to venture in. What appears to be an insurmountable challenge is, in fact, an opportunity waiting to be seized.

Understanding and improving legacy code is not just a task—it's a craft, a puzzle that rewards those who dare to solve it. It teaches resilience, sharpens analytical thinking, and fosters an appreciation for the history and evolution of software development. The satisfaction of untangling a tricky piece of code, fixing a long-standing bug, or optimizing an inefficient part of the system is immensely gratifying.

Remember, many developers have walked this path before you and succeeded. The tools and strategies to conquer legacy code are within reach, and the community of developers is here to support you. The expertise

gained from mastering legacy code is not only a personal triumph but a highly valuable skill in the industry.

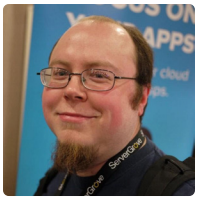
Coming Up Next: Tools and Strategies

While the code may appear overwhelming at first glance, there are proven methods to help you make sense of it all. Next month we will delve into specific steps and tools for digging through legacy PHP code. From using powerful debugging tools like xdebug, to writing unit tests that ensure code reliability, to employing static analysis for maintaining code quality, we will discuss ways to actually make changes to code and help make the codebase better as you go along.

These tools and techniques are not just for the coding elite; they are accessible and applicable to developers at all levels. By learning and applying them, you'll be transforming what once seemed like a mountain into a manageable hill.

Related Reading

- *Education Station: Serverless PHP with Bref* by Chris Tankersley, July 2023. https://phpa.me/education-jul-23
- *Education Station: Dude, Where's My Server?* by Chris Tankersley, June 2023. https://phpa.me/education-jun-23



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

Vulnerability Management 101

Eric Mann

Every piece of published code will eventually suffer a vulnerability. Recognizing this truth is the first step to establishing a vulnerability management program.

> *Last month, a critical bug hit the Mastodon community[1]. By hosting a malicious image on their own server, an attacker could potentially infect and hijack any server that federated its content. This provided malicious users with the ability to easily DOS[2] a victimized server. Such an attacker could even trigger remote code execution, giving them an interactive web shell on any affected server in the world.*

Mastodon isn't a single product run by one dev team. It's a community of interconnected ("federated") servers run by volunteer administrators—over 13,000 of them, to be precise. Rolling out a fix required all of the administrators to realize there was a problem and quickly patch their servers to avoid a compromise.

It also required these administrators to be *aware* of the vulnerability that existed in their software in the first place.

Vulnerabilities can exist in your own software. They can live within libraries or components upon which you build your tools. The underlying operating system or its various software modules could ship a change or interface that renders your application vulnerable. The act of merely staying aware of the ways in which your application could fail is paralyzing; it raises the question of how much risk you're willing to accept.

Risk Awareness

The core of any good threat modeling exercise is categorizing threats and evaluating the level of risk they pose to your business. It's vital that your team be aware of the risks inherent in your application and the libraries, tools, or other applications it depends on. Are these risks to your product itself? To your customers' accounts? To any financial or other protected/sensitive data?

Each risk carries a certain level of likelihood as well, and it's important that you properly mitigate and manage that risk. Taking on the right level of risk for your business—a level paired with adequate *reward* in terms of business opportunity—is critical to moving forward.

Risk Management

The only truly secure product is the one that never launches.

Said another way, literally every product on the market is launched with some kind of risk or insecurity. Just the fact that you've allowed other humans to interact with the product means someone, somewhere, can submit bad data to the system and trigger some kind of fault. A component will break. A hosted server will crash or otherwise fail.

As stated earlier, every product today on the market is a child of compromise—the ultimate balance between risk and reward.

Some risks are easy to prevent entirely—keeping an application off the public internet and only on a private, local network minimizes the possibility that an outside attacker can ever target it in the first place. Other necessary risks might need to be balanced by outward means—like cybersecurity insurance to protect in the event of a compromise.

In every case, it's a matter of identifying the type of risk, the impact on the business should it be realized, and the best way to manage (or mitigate) the overall impact on the business.

Vulnerability Management Strategies

Among the largest risk faced in the PHP community is the possibility of a vulnerability making its way into your application by way of a dependency. This could be a package installed by Composer, a legacy submodule built by a long-departed dev team, or a few hundred lines copied by an intern from Stack Overflow. Regardless of how the code made it into the project, the fact that *someone else* wrote and is responsible for its security keeps many an engineer awake at night.

It's critical that your team manage its dependencies well and ensure you're taking appropriate steps to identify *and quickly patch* and identified vulnerabilities. Unfortunately, there's no silver bullet here—no one true way to keep your stack safe and secure. Instead, you might want to leverage *multiple* strategies at the same time.

Static Code Analysis

Open source tools like Psalm[3] perform static analysis on your code. This is similar to unit testing but helps prevent further type-related errors at runtime so you can avoid potential bugs with corrupt user data impacting the flow of the application.

Dynamic Application Security Testing (dast)

DAST tools, like the analyzers provided by GitLab[4], scan your applica-

1 https://phpa.me/tootroot
2 https://phpa.me/dos
3 https://psalm.dev
4 https://phpa.me/app-security-dast

tion *while it is running* to identify any potential problems with its APIs and behavior. These checks are built into your application's continuous integration (CI) pipelines alongside any standalone tests and help the team prevent ever shipping known-to-be-insecure API patterns to production.

Automated Dependency Updates

Tools like GitHub's Dependabot[5] automatically submit pull requests for your project to update dependencies when a new version is deployed to patch a vulnerability. Your team still maintains control over the project, but you don't need to hunt within your software supply chain to find updates actively.

In Conclusion

You will never launch a product or application into the wild that is totally secure, without risk, and insulated from vulnerabilities. This means, it's not a matter of *if* you'll face an exploitable vulnerability, but *when*.

You owe it to yourself, your customers, the rest of your dev team, and anyone who's going to be paged at 2 in the morning to establish and exercise a solid vulnerability management program with your team.

5 https://phpa.me/software-supply-chain

Related Reading

- *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
- *Security Corner: Tabletop: Planning for Disaster* by Eric Mann, May 2023. https://phpa.me/security-may-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

Small Changes

Maxwell Ivey

Small changes have significant effects on your disabled audience. I really appreciate the opportunity to share my experiences with you. And I hope we can help developers and adaptive tech users better understand each other.

This month, I want to talk about updating websites, apps, and content. I want you to be aware of how small changes can drastically affect the user experience of people with disabilities.

I also want to help you avoid some of the frustration from your disabled users when circumstances require you to make updates.

When a visually impaired person visits a site with a screen reader or magnifier, they are like someone moving to a new city. They don't know where anything is. And they spend a lot of time learning their way around.

You probably remember how frustrating it was when street construction forced you to take a different route on short notice.

When learning my way around a new site, I have to figure out which of my navigation commands will let me find the information I need as quickly as possible. Once I have figured out the lay of the land, I then memorize what worked for that particular site.

And as long as a site doesn't change, my life gets easier every time I visit your site or use your app.

But when you change a page layout, rename pages, change permalink structure, etc.; I am forced to start all over and figure out how to navigate the new version of your site or app.

And the change can be very small.

Examples of Frustration

For example, I use a website called speaker match, among others, to find podcasts seeking guests. I had total command of that website.

Then one day, they decided to change their secondary page links from hyphens to underscores to represent spaces in URLs.

I had to start over at the home page to figure out where they had moved the pages I used most often. I wasted over an hour before I figured it out. I was very frustrated and never found out what forced the change.

Another example is LinkedIn, changing the process for sharing your posts with groups. They added an extra step. Now you have to select audience, then select groups, then select the group, then press done, and then press post.

This one was much easier to figure out, but having to spend that extra time for no apparent reason is very frustrating.

So, how do you avoid creating roadblocks for your disabled users?

Evaluate Need

Please ask yourself if the changes you are proposing are absolutely necessary to the function, safety, or enjoyment of your project. Obviously, if the changes are simply cosmetic, it would be appreciated by folks like me if you decided to keep things the way they are. But if the change is required, you would naturally make the changes.

Ask for Input

If you have access to people using adaptive technology, please reach out to them and ask how they feel the change will affect their experience. If you don't have anyone on the team, I'd love to help you build relationships with people who could help you. Knowing the actual impact on users with disabilities can help you plan your updates better.

Communicate Changes

Once you have decided to make changes, you should reach out to your users with an announcement. You could send an email to everyone or post about the changes on social media.

Be sure to tell us what you are changing, why the update is needed, and how the site or app will behave going forward.

Disabled People are Shy

Communicating directly with just your users with disabilities is kind of hard. Because a lot of people with disabilities won't tell you they are disabled. They are worried about receiving a lower level of service if site owners know about their disability.

I personally find this short-sighted. I have been doing this for years, and I almost always get better treatment by letting people know in advance of my disability.

Of course, I approach people in a friendly, non-threatening, "let's find a way for me to use your site or app in the best way possible" manner.

We all know that the internet is a constantly changing environment. There are millions of users creating billions of pieces of content. I know change is inevitable. But it doesn't have to be frustrating to your users.

I hope I have given you a better understanding of the fragile nature of online navigation with adaptive technology. And I look forward to learning more about the development process from you. The better I understand your pressures and concerns, the better I can help you.

I look forward to your comments and hope to start some conversations. 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

Rector Refactoring

Joe Ferguson

This month we're covering the use of Rector, a tool for automatic code refactoring in PHP. We'll explain how to configure and use Rector to improve code quality, remove dead code, and enforce coding style. We'll also cover examples of how Rector can be used to refactor code in a real-world application, Snipe-IT.

Getting Rector installed in our application is the normal `composer require rector/rector --dev`, and we can bootstrap our configuration `rector.php` with `php vendor/bin/rector init`.

The `init` command creates our `rector.php` containing the following: (See Listing 1)

Listing 1.

```
1. <?php
2.
3. use Rector\Config\RectorConfig;
4. use Rector\Set\ValueObject\SetList;
5. use Rect...\TypedPropertyFromStrictConstructorRector;
6.
7. return static function (
8.     RectorConfig $rectorConfig
9. ): void {
10.     // register single rule
11.     $rectorConfig->rule(
12.         TypedPropertyFromStrictConstructorRector::class
13.     );
14.
15.     // here we can define, what sets of rules
16.     // will be applied
17.     // tip: use "SetList" class to autocomplete
18.     // sets with your IDE
19.     $rectorConfig->sets([
20.         SetList::CODE_QUALITY,
21.         SetList::DEAD_CODE,
22.     ]);
23. };
```

What your `rector.php` contains might be different based on the version of PHP you're using. To determine what version of rector was installed, we can do `composer show rector/rector`

```
$ composer show rector/rector
name     : rector/rector
descrip. : Instant Upgrade and Automated Refactoring...
keywords : automation, dev, migration, refactoring
versions : * 0.17.12
```

Knowing our Rector version is 0.17.12 is helpful to use version 0.17.12[1] for documentation to ensure we're setting our correct configuration. (See Figure 1)

Figure 1.

```
(~/Code/php-easy-math)(rector M:2 S:2 no-remote)
└─ composer require rector/rector --dev
./composer.json has been updated
Running composer update rector/rector
Loading composer repositories with package information
Updating dependencies
Lock file operations: 2 installs, 0 updates, 0 removals
  - Locking phpstan/phpstan (1.10.28)
  - Locking rector/rector (0.17.12)
Writing lock file
Installing dependencies from lock file (including require-dev)
Package operations: 2 installs, 0 updates, 0 removals
  - Downloading phpstan/phpstan (1.10.28)
  - Downloading rector/rector (0.17.12)
  - Installing phpstan/phpstan (1.10.28): Extracting archive
  - Installing rector/rector (0.17.12): Extracting archive
Generating autoload files
27 packages you are using are looking for funding.
Use the `composer fund` command to find out more!
No security vulnerability advisories found
Using version ^0.17.12 for rector/rector
```

For example, we can look at an example Math application written for PHP 8.0 and see Rector at work even with our basic configuration. (See Listing 2 on the next page)

Running `vendor/bin/rector process src --dry-run` will output the following: (See Figure 2 on the next page)

1 https://phpa.me/github

Listing 2.

```php
<?php
/**
 * \EasyMath\Addition Class
 */

namespace easyMath;

/**
 * Class containing addition methods
 */
class Addition
{
    /**
     * Sum 2 numbers
     * @param float $x
     * @param float $y
     * @return float
     */
    public function add($x, $y)
    {
        return $x + $y;
        return "Example Dead Code";
    }

    /**
     * Sum many numbers
     * @param float ...$numbers
     * @return float
     */
    public function sum(float ...$numbers): float
    {
        $sum = 0;
        foreach ($numbers as $number) {
            $sum += $number;
        }
        return $sum;
    }

}
```

If we run the command again without the `--dry-run` flag Rector would have changed our file. The two things Rector removed from our file we triggered by the rules mentioned in the output `RemoveUselessParamTagRector`, which will remove a documentation block with the same type as the parameter type. Since we're setting our `@param` to `float` while also specifying the type `float ...$numbers`, Rector's rule will remove the line for us. This also goes for the `@return float`, which is why it was also removed. Finally, Rector removed our `return "Example Dead Code";` as it would never have been reached during our program's execution. Dead code may sound like a trivial issue; however, in legacy code bases where logic is often buried deep within nested calls, it can be tricky to untangle what the application is doing.

Why didn't Rector complain about the `add()` method's parameters? Looking closer at the method: (See Listing 3)

Figure 2.

```
┌(~/Code/php-easy-math)(rector M:4 S:2 no-remote)
└─ vendor/bin/rector process src --dry-run
 1/1 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] 100%
1 file with changes
===================

1) src/Addition.php:18

    ---------- begin diff ----------
@@ @@
     public function add($x, $y)
     {
         return $x + $y;
-        return "Example Dead Code";
     }

     /**
      * Sum many numbers
-     * @param float ...$numbers
-     * @return float
      */
     public function sum(float ...$numbers): float
     {
    ----------- end diff -----------

Applied rules:
 * RemoveUselessParamTagRector
 * RemoveUselessReturnTagRector
 * RemoveUnreachableStatementRector


[OK] 1 file would have changed (dry-run) by Rector
```

Listing 3.

```php
/**
 * Sum 2 numbers
 * @param float $x
 * @param float $y
 * @return float
 */
public function add($x, $y)
{
    return $x + $y;
}
```

Since we don't specify types for `$x` and `$y` Rector realizes that we do need those `@params`. To stay with modern practices, we should add types and let Rector remove what we no longer need. (See Listing 4)

Listing 4.

```php
/**
 * Sum 2 numbers
 * @param float $x
 * @param float $y
 * @return float
 */
public function add(float $x, float $y): float
{
    return $x + $y;
}
```

Running Rector again, we can see our parameters have been removed: (See Figure 3)

Figure 3.

```
(~/Code/php-easy-math)(rector ??:1 M:2 S:2 no-remote)
  vendor/bin/rector process src
 1/1 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] 100%
1 file with changes
===================

1) src/Addition.php

    ---------- begin diff ----------
@@ @@
 {
 /**
  * Sum 2 numbers
- * @param float $x
- * @param float $y
- * @return float
  */
 public function add(float $x, float$y): float
 {
    ----------- end diff -----------

Applied rules:
 * RemoveUselessParamTagRector
 * RemoveUselessReturnTagRector


 [OK] 1 file has been changed by Rector
```

We can apply the same change to our `Subtraction.php` class' `subtract` method:

```
/**
 * Subtract 2 numbers
 * @return float
 */
public function subtract(float $x, float $y)
{
    return $x - $y;
}
```

You might have noticed we left a `* @return float` in our comment. We can add a return type, and then Rector will remove our return comment:

```
/**
* Subtract 2 numbers
*/
public function subtract(float $x, float $y): float
{
    return $x - $y;
}
```

Since we've made some changes to our application code, we should verify everything is still working by running our PHPUnit tests:

```
./vendor/bin/phpunit
PHPUnit 9.6.10 by Sebastian Bergmann and contributors.

..................... 20 / 20 (100%)

Time: 00:00.007, Memory: 6.00 MB

OK (20 tests, 20 assertions)
```

So we've added Rector to our application and configured it to check for dead code and code quality. You can learn more about what the dead code[2] and quality[3] rules from the documentation.

With the basic rules out of the way, we can start adding more complex rules, such as `SetList::``CODING_STYLE`, which will check and correct any issues it finds with our code based on its Coding Style[4] rules. One issue I noticed after running Rector with `CODING_STYLE` is it removed our spread operator, which is what allows us to accept a dynamic number of inputs. (See Figure 4)

Figure 4.

```
     /**
      * Sum many numbers
      * @param float ...$numbers
-     * @return float
      */
-    public function sum(float ...$numbers): float
+    public function sum(array $numbers = []): float
     {
         $sum = 0;
         foreach ($numbers as $number) {
             $sum += $number;
         }
+
         return $sum;
     }
```

Test failures confirm and indicate we need to investigate our code

```
1) AdditionTest::testEasyMathKnowsHowToSum with
 data set #0 (0, 0, 0) TypeError: Addition::sum():
Argument #1 ($numbers) must be type array, int given,
 called in tests/AdditionTest.php on line 37
```

The particular rule that was being triggered was from the class `Rector\CodingStyle\Rector\ClassMethod\UnSpreadOperatorRector`[5], which we can add to our `rector.php` to ignore this particular rule. (See Listing 5 on the next page)

2 https://phpa.me/github-deadcode

3 https://phpa.me/github-codequality

4 https://phpa.me/github-codingstyle

5 https://phpa.me/github-rules

Listing 5.

```
1. ...
2. use Rector\...\ClassMethod\UnSpreadOperatorRector;
3. ...
4. $rectorConfig->sets([
5.     SetList::CODE_QUALITY,
6.     SetList::DEAD_CODE,
7.     SetList::CODING_STYLE,
8. ]);
9.
10. // here we can define rules we want to ignore
11. $rectorConfig->skip([
12.     UnSpreadOperatorRector::class,
13. ]);
```

Remember that Rector is just like any other code quality or automatic refactoring tool in that it doesn't know your domain and your constraints, so it can only infer so much from your code. Use the rules for what they are, rules, not laws. The police are not coming after your code. At least not yet… You should take time to configure Rector for your project or even your team's needs, as there is never a silver bullet or one size fits all solution.

We've explored Rector with a simplistic application but what about real-world applications? Let's take a dive into the Snipe-IT[6], which currently supports PHP 8.1 but not 8.2. If you are unfamiliar with Snipe-IT, it is an open-source asset management library built with Laravel 8. It runs on PHP 8.1 and is a fantastic example of a real-world web application built with Laravel. I won't distract you with setting up Snipe-IT, but you'll need a MySQL database and a way to run a PHP 8.1 application.

Because we're using a different PHP version, we can use `rector init` to create our basic `rector.php` configuration. Remember to check your Rector version with `composer show rector/rector`; in my case, it installed `0.14.6`, and the configuration was slightly different. Just as we did before, but we're going to include Code Quality, Dead Code, and Coding Style sets. (See Listing 6)

Once we have the application setup, we can run `php artisan test` to get a baseline to see what's going on with the code before we apply any changes. (See Listing 7)

Listing 7.

```
1. $ php artisan test
2. ...
3. PASS  Tests\Feature\Reports\CustomReportTest
4.   ✓ custom asset report
5.   ✓ custom asset report adheres to company scoping
6.
7.   PASS  Tests\Feature\Users\UpdateUserTest
8.   ✓ users can be activated
9.   ✓ users can be deactivated
10.  ✓ users updating themselves do not deactivate
11.        their account
12.
13.  Tests:  2 warnings, 129 passed
14.  Time:   7.98s
```

Listing 6.

```
1. <?php
2.
3. use Rect...\InlineConstructorDefaultToPropertyRector;
4. use Rector\Config\RectorConfig;
5. use Rector\Set\ValueObject\SetList;
6. use Rector\Set\ValueObject\LevelSetList;
7.
8. return static function (
9.     RectorConfig $rectorConfig
10. ): void {
11.     $rectorConfig->paths([
12.         __DIR__ . '/tests'
13.     ]);
14.
15.     // register a single rule
16.     $rectorConfig->rule(
17.        InlineConstructorDefaultToPropertyRector::class
18.     );
19.
20.     // define sets of rules
21.     $rectorConfig->sets([
22.         SetList::CODE_QUALITY,
23.         SetList::DEAD_CODE,
24.         SetList::CODING_STYLE,
25.     ]);
26. };
```

Now that we can see the code is in a known good state we can start running Rector without the `--dry-run` flag to start writing some of these changes. Naturally, with such a large codebase it might be daunting just to run `rector app/` to see just how spectacularly things complain, so instead, I recommend you start with a section of the codebase. If you're uncertain about where to start, I suggest starting with your best-tested code so that you can quickly see any unintended side effects in your test results.

We're going to start processing the `tests/Feature` folder since these are easy to run and get feedback quickly without having to dive too far into the depths of the application logic. Rector wants to change nine files in this folder, so we will start out with one file at a time. (See Figure 5 on the next page)

The test in question is to test that the Asset index returns the expected assets based on the route options. The full test method is below, and you can see the entire file on GitHub[7] (See Listing 8 on the next page)

This test starts by creating 3 Assets via the `Asset::factory()` method. These are persisted to the database and can now be queried by our next statement acting as a super user via `$this->actingAsForApi` . The test creates a superuser and acts out the test as that user getting the JSON from the named route `api.assets.index` with extra parameters for sort, order, offset, and result limit. The test continues and asserts that the response code from the request is 200 via `->assertOk` and then validates the JSON structure containing the keys `total`

6 https://github.com/snipe/snipe-it

7 https://phpa.me/github-snipe

Figure 5.

```
(~/Code/snipe-it)(develop ??:1 AM:1 M:2 $:16)
(1)- ./vendor/bin/rector process tests/Feature/Api/Assets/AssetIndexTest.php --dry-run
 1/1 [▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓▓] 100%
1 file with changes
===================

1) tests/Feature/Api/Assets/AssetIndexTest.php:29

    ---------- begin diff ----------
@@ @@
                'total',
                'rows',
            ])
-           ->assertJson(fn(AssertableJson $json) => $json->has('rows', 3)->etc());
+           ->assertJson(static fn(AssertableJson $json) => $json->has('rows', 3)->etc());
    }

    public function testAssetIndexAdheresToCompanyScoping()
    ----------- end diff -----------

Applied rules:
 * StaticArrowFunctionRector


[OK] 1 file would have changed (dry-run) by Rector
```

Listing 8.

```
1. public function testAssetIndexReturnsExpectedAssets()
2. {
3.     Asset::factory()->count(3)->create();
4.
5.     $this->actingAsForApi(
6.             User::factory()->superuser()->create()
7.         )->getJson(
8.             route('api.assets.index', [
9.                 'sort' => 'name',
10.                'order' => 'asc',
11.                'offset' => '0',
12.                'limit' => '20',
13.            ]))
14.         ->assertOk()
15.         ->assertJsonStructure([
16.             'total',
17.             'rows',
18.         ])
19.         ->assertJson(
20.             fn(AssertableJson $json) =>
21.                         $json->has('rows', 3)->etc()
22.         );
23. }
```

and `rows`. The final `assertJson()` checks the assertion and there are three rows in the result.

Rector has an issue with the last line of our test `->assertJson(fn(AssertableJson $json) => $json->has('rows', 3)->etc());`, flagging that the `StaticArrowFunctionRector` needs to be applied translating the line into, `>assertJson(static fn(AssertableJson $json) => $json->has('rows', 3)->etc());`. This rule dictates that Arrow Functions should be set to static when possible. From the Rector Documentation:

```
StaticArrowFunctionRector

Changes ArrowFunction to be static when possible

    class: Rector\...\StaticArrowFunctionRector

-fn (): string => 'test';
+static fn (): string => 'test';
```

Remember, we don't have to live with Rector defaults, and if we disagree with making all of these static, we can ignore this rule as we have previously by adding the class to `$rectorConfig->skip()`.

```
use Rector\CodingStyle\Rector\
            ArrowFunction\StaticArrowFunctionRector;
...
$rectorConfig->skip([
    StaticArrowFunctionRector::class,
]);
```

As we expand the paths we're passing into Rector, we can begin to see other rules making changes, such as `RemoveUnusedVariableAssignRector`. (See Figure 6)

Figure 6.

```
    public function testUsersScopedToCompanyWhenMultipleFullCompanySupportEnabled()
@@ @@
            ['first_name' => 'Anakin', 'last_name' => 'Skywalker', 'username' => 'askywalker'],
        ])->create();

-       $sith = Company::factory()
+       Company::factory()
            ->has(User::factory()->state(['first_name' => 'Darth', 'last_name' => 'Vader', 'username' => 'dvader']))
            ->create();
```

In this instance, we can see Rector removes the unused variable `$sith` because it's not used anywhere else in the test. We can see the refactored test below: (See Figure 7)

Figure 7.

```
public function testUsersScopedToCompanyWhenMultipleFullCompanySupportEnabled()
{
    $this->settings->enableMultipleFullCompanySupport();

    $jedi = Company::factory()->has(User::factory()->count( count: 3)->sequence(
        ['first_name' => 'Luke', 'last_name' => 'Skywalker', 'username' => 'lskywalker'],
        ['first_name' => 'Obi-Wan', 'last_name' => 'Kenobi', 'username' => 'okenobi'],
        ['first_name' => 'Anakin', 'last_name' => 'Skywalker', 'username' => 'askywalker'],
    ))->create();

    Company::factory()
        ->has(User::factory()->state(['first_name' => 'Darth', 'last_name' => 'Vader', 'username' => 'dvader']))
        ->create();

    Passport::actingAs($jedi->users->first());
    $response = $this->getJson(route( name: 'api.users.selectlist'))->assertOk();

    $results = collect($response->json( key: 'results'));

    $this->assertEquals( expected: 3, $results->count());
    $this->assertTrue(
        $results->pluck( value: 'text')->contains(static fn($text) => str_contains((string) $text, 'Luke'))
    );
    $this->assertFalse(
        $results->pluck( value: 'text')->contains(static fn($text) => str_contains((string) $text, 'Darth'))
    );
}
```

Another Rector change can be seen in the following output showing an example of the `NullToStrictStringFuncCallArgRector` usage: (See Figure 8 on the next page)

Figure 8.

The purpose of this change is to enforce the value of `$text` being passed into `str_contains()` is typecast to a string. This might confuse some users who are just getting started with types. You can read more about type casting in the PHP docs for Type Juggling[8].

You may also see instances of `StaticClosureRector`, which will update closures to be static when possible. Depending on your preferences, this may lower readability; remember, readability is important, and you shouldn't make changes that make things harder for humans to read and understand. (See Figure 9)

Figure 9.

Another rule that I enjoy from Rector is `CatchExceptionNameMatchingTypeRector` because it helps me be less lazy with my usage of `} Catch (\Exception $e) {` will expand the variable name. We can see an application of this rule in the diff:

```
-         } catch (\Exception $e) {
-             \Log::debug($e);
+         } catch (\Exception $exception) {
+             \Log::debug($exception);
```

This is one of those things that I probably wouldn't ever fix myself, so it's nice to have Rector thinking about these things so I don't have to.

You may need to adjust Rector's code quality rules to your own or maybe bypass it entirely if you're already happy with something like PHP-CS-Fixer[9] already fixing your code style based on an established guide.

One last feature to show off what Rector can do is by using the set `LevelSetList::``UP_TO_PHP_82`, which will attempt to fix any issues with our code that would prevent us from running on PHP 8.2. Looking through some of the issues it's flagging on Snipe-IT, we can see influences from Psalm PHP in `AddDefaultValueForUndefinedVariableRector` which helps Psalm users from getting false positives in scenarios where the code can't be understood that a value isn't undefined.

```
public function store(
    AssetCheckoutRequest $request, $assetId
)
     {
+        $asset = null;
         try {
             // Check if the asset exists
             if (! $asset = Asset::find($assetId)) {
```

Think of this in the same manner back in "the old days" when we put our variable types in doc blocks because it helped the human code readers see what the function was taking and returning. In today's Rector world, we have tools helping other tools understand the code humans wrote. This certainly has no implications for artificial intelligence, so I'm sure we're totally safe. Rector also flags a number of false positives, likely because of how things were implemented. Remember that tools such as Rector and Psalm can only infer what you have in your code, not your *intention*. You have to ensure your code is appropriately encapsulating not only your domain and business logic but the intentions as well. This is where we tip our hats to Behavior Driven Development and similar processes, which put a significant focus on the application's behavior and testing the results against expectations.

Rector isn't going to magically upgrade your code to PHP 9.5 and have all your tests exploding in a chorus of songs for you. Rector *will* help you spot problems in your code that *will* save you time. If you're already using a static analysis tool Rector might not offer you too much; however, it is still worth checking out the PHP compatibility set lists to check what versions your applications can support. Happy Rectoring.

Related Reading

- *The Workshop: The Workshop: PostgreSQL* by Joe Ferguson, July 2023. https://phpa.me/workshop-jul-23
- *The Workshop: Databases as a Service* by Joe Ferguson, June 2023. https://phpa.me/workshop-jun-23
- *The Workshop: The Workshop: Minicli* by Joe Ferguson, May 2023. https://phpa.me/workshop-may-23

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

8 https://phpa.me/php-typejuggling

9 https://github.com/PHP-CS-Fixer/PHP-CS-Fixer

Bubble Sorting

Oscar Merida

We've escaped the maze(s)! On to new subjects, lets explore sorting and the different ways we have to put things in order. We'll start with the first one I remember writing in Turbo Pascal.

Recap

> *For the next issue, 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. Write a function that sorts your array from lowest to highest by comparing two adjacent elements at a time—a bubble sort.*
>
> *Bonus points—generate many such arrays, collect and present data on how long it takes to sort. Show the shortest, longest, and mean times.*

The Algorithm

Sorting algorithms have been of interest since the earliest computing days. It's no wonder why. Computers can store lists of things in memory; it's natural to ask if they can re-order by one or more attributes of your data. Howard B. Demuth explored "Electronic Data Sorting"[1] in his doctoral thesis back in 1956. He observed:

> *Since sorting is currently the bottleneck in many data processing systems, the results should be of interest to those who use or manufacture such systems.*
>
> *Sort algorithms are evaluated using Big-O notation, based on the size of the list. As the number of elements in a list increases, so does the algorithm's workload in terms of comparisons and swaps. The number of comparisons may also depend on the initial sort order. You should be familiar with this notation. For bubble sort, the average performance is N^2. That's not since it means that for a list of some size, say 100 elements, it might take up to 10,000 operations to sort the list. For real-world performance, that's not practical. We'll see if there are better sorting algorithms.*

Another consideration is the amount of memory consumed. If your algorithm has to make copies of elements or use temporary variables, it'll use more memory. Nowadays, our computers have unheard-of amounts of RAM, but we also tend to work with very large data sets. Running out of memory may mean using slower swap memory or watching your program halt suddenly. Since bubble sort modified the array in place, it doesn't need more memory.

So why's it called a bubble sort? The algorithm steps through an array and compares an element with the next element. If the first element is greater, it swaps the two. It makes one pass through the array, swapping elements as it goes, and then starts again. The list is sorted and stops if it makes a pass without making swaps. We'll use integers for keys to keep it simple. An example of the algorithm in practice looks like:

```
6 12 8 3 11
6 8 12 3 11
6 8 3 12 11
6 8 3 11 12
6 3 8 11 12
3 6 8 11 12
```

Swapping Elements

Before we get to the sorting algorithm, it looks like we need a function to swap two elements in an array. The signature could look like the one below. Since arrays are passed-by-reference until modified, this might use more memory than we intend if our function makes a copy of the input array.

```
function swap_elements(
    array $list,
    int $i,
    int $j
): array

// called like:
// $list = swap_elements($list, 1, 2);
```

It'd be safer to pass it by reference so we don't make a copy and use twice as much (or more) memory.

```
function swap_elements(
    array &$list,
    int $i,
    int $j
): void;
// called like:
// swap_elements($list, 1, 2);
```

One implementation, with some index checking to verify `$i` and `$j` are valid is in Listing 1.

1 https://www.proquest.com/docview/301940891

Listing 1.

```
function swap_elements(
    array &$list,
    int $i,
    int $j
): void {
    // safety check
    if (
        array_key_exists($i, $list) &&
        array_key_exists($j, $list)
    ) {
        $tmp = $list[$i];
        $list[$i] = $list[$j];
        $list[$j] = $tmp;
        return;
    }
    $message = "Invalid index specified";
    throw new \InvalidArgumentException($message);
}
```

Scanning and Swapping

Now that we can swap any two elements in the array—we should save that function, which might come in handy later—let's loop through the array, making swaps until we don't make any. Listing 2 has a first implementation,

I added some echo statements to see the swaps it made. For an array with eight items, it made eight swaps because my "randomly selected" starting list was mostly sorted.

```
1, 5, 8, 12, 15, 19, 4, 11
1, 5, 8, 12, 15, 4, 19, 11
1, 5, 8, 12, 15, 4, 11, 19
--- end of scan
1, 5, 8, 12, 4, 15, 11, 19
1, 5, 8, 12, 4, 11, 15, 19
--- end of scan
1, 5, 8, 4, 12, 11, 15, 19
1, 5, 8, 4, 11, 12, 15, 19
--- end of scan
1, 5, 4, 8, 11, 12, 15, 19
--- end of scan
1, 4, 5, 8, 11, 12, 15, 19
--- end of scan
```

One thing we can notice at the end of the first scan loop, the largest element is in the last spot. After the next scan loop, the last two elements are in their final spot. One tweak we can make to reduce the number of adjacent element checks is not scanning all the way to the end of the array in each scan. This doesn't reduce the number of swaps we need to make, but it does mean we make fewer checks in later scans.

Testing Performance

The second part of this month's exercise is to test the algorithm with many large arrays. We need a way to make a test list quickly, and php's functional array functions can do it without looping (see Listing 3).

Listing 2.

```
function bubble_sort(array &$list): void
{
    $max = count($list);
    do {
        // stop two elements before max so we have two
        // elements to swap
        $swapped = false;
        for ($i = 0; $i < $max -1; $i++) {
            if ($list[$i] > $list[$i + 1]) {
                $swapped = true;
                swap_elements($list, $i, $i + 1);
            }
        }
    } while ($swapped);
}
```

Listing 3.

```
function make_test_list(
    int $size,
    int $min=0,
    int $max=20000
): array {
    $list = array_fill(0, $size, 0);
    array_walk($list,
        function(&$item) use ($min, $max): void {
            $item = random_int($min, $max);
        }
    );
    return $list;
}
```

Listing 4.

```
$runtimes = [];
$runs = 100;
$size = 100;
for ($i = 0; $i < $runs; $i++) {
    $list = make_test_list($size);
    $start = hrtime(true);
    bubble_sort($list);
    $end = hrtime(true);
    $runtimes[] = ($end - $start) / 1000000000;
    unset($list);
}
echo "\n\nFastest: " . min($runtimes) . "\n";
echo "Slowest: " . max($runtimes) . "\n";
echo "Mean:    ".(array_sum($runtimes) / $runs)."\n";
```

The loop in Listing 4 will collect some rough performance data.

I collected the stats here; you can see that as the size of the list gets larger, performance drops significantly.

List Size	100	1000	5000	10000
Fastest	0.000284	0.0301	0.8095	3.2070
Slowest	0.000665	0.0370	0.8780	3.4395
Mean	0.000349	0.0321	0.8414	3.3420

Rabbits and Turtles

One drawback of the bubble sort is that we compare adjacent elements. It takes a lot of swaps for elements that have to move a large distance. The Comb Sort addresses this limitation; for next month, write an implementation. 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.

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.*

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

2 https://psysh.org

Create Observability, Part 3: Rewrite Business Process

Edward Barnard

Last month we looked at flowcharting as a way of capturing and communicating "tribal knowledge". This month, as we rewrite a business process, we'll use a flowchart to document the new business process. We needed to fix the business process before we could fix the code.

Last month I used a flow diagram as a way of documenting a business process. That process was a *temporal* process, meaning that certain phases needed to follow in sequence. We saw that season competition follows season registration at the USA Clay Target League. Tournament competition follows season competition.

The problem we addressed was that the software for each phase assumes our database and other configuration items are in the correct state for the current phase. But, to develop a new feature for a future phase, we needed a way to set up the development environment for that new feature. We created the flow diagram to show where the setup steps fit in.

Current Business Process

This month we have a new business requirement. We need the ability to add a new team. We've been adding new teams for over a decade, so we already have that capability. This is another case of "tribal knowledge". Nothing concerning this capability is written down. Our PHP codebase and database tables implement the capability, but that's the extent of what's written.

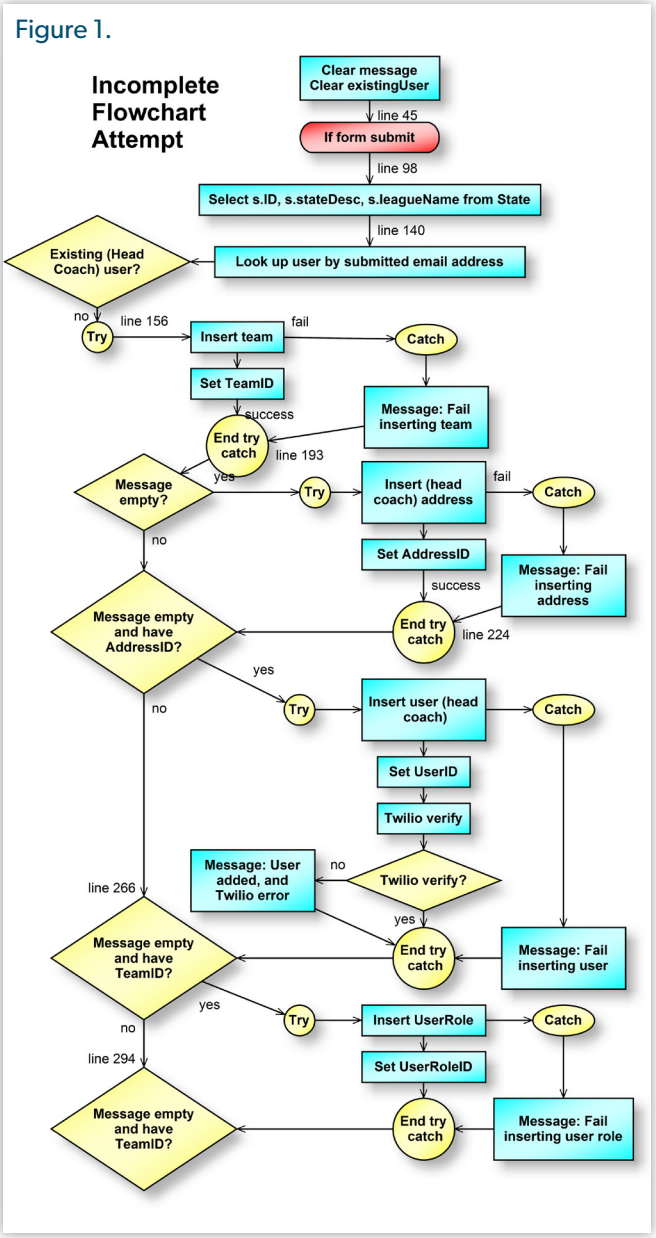
This month we need to address a problem with this feature. If something goes wrong while adding a team, we are leaving the database in an inconsistent state. Because of this problem, we have been "backing out" the partially-completed database records by hand. However, at the moment, we have a relatively large stack of new teams to add and need this capability to work correctly, not leaving partially-created teams lying around.

I tried to draw out the Add Team feature as it currently exists. This quickly turned into a hopeless exercise. See Figure 1. The PHP logic was too convoluted. I started adding line numbers to the flow chart to try to keep track of where I was.

This turned out to be a useful exercise. That's why I'm showing you the broken flowchart in Figure 1. We have the sequence of steps out of order. For example, we are verifying form data *after* inserting the record into the database.

The current approach is far too optimistic. We need to insert several database records, but we insert each record as a separate step. If that step fails, we reload the "add team" page with the error message. That approach leaves the database in an inconsistent state.

Figure 1.



Desired Business Process

The broken flowchart assisted our conversation about what the business process *should* be. When adding a team, we add both the team and the Head Coach. The Head Coach might or might not already be in our system. It's quite common for the new team's Head Coach to have already been on staff for a different team.

Our business flow is as follows:

1. Validate form data: Head Coach phone and email
2. If prospective Head Coach is not in database, insert new user record (i.e., find or create user)
3. Create team records (team record, initial permissions, connect Head Coach to team)
4. Send out Welcome email to Head Coach

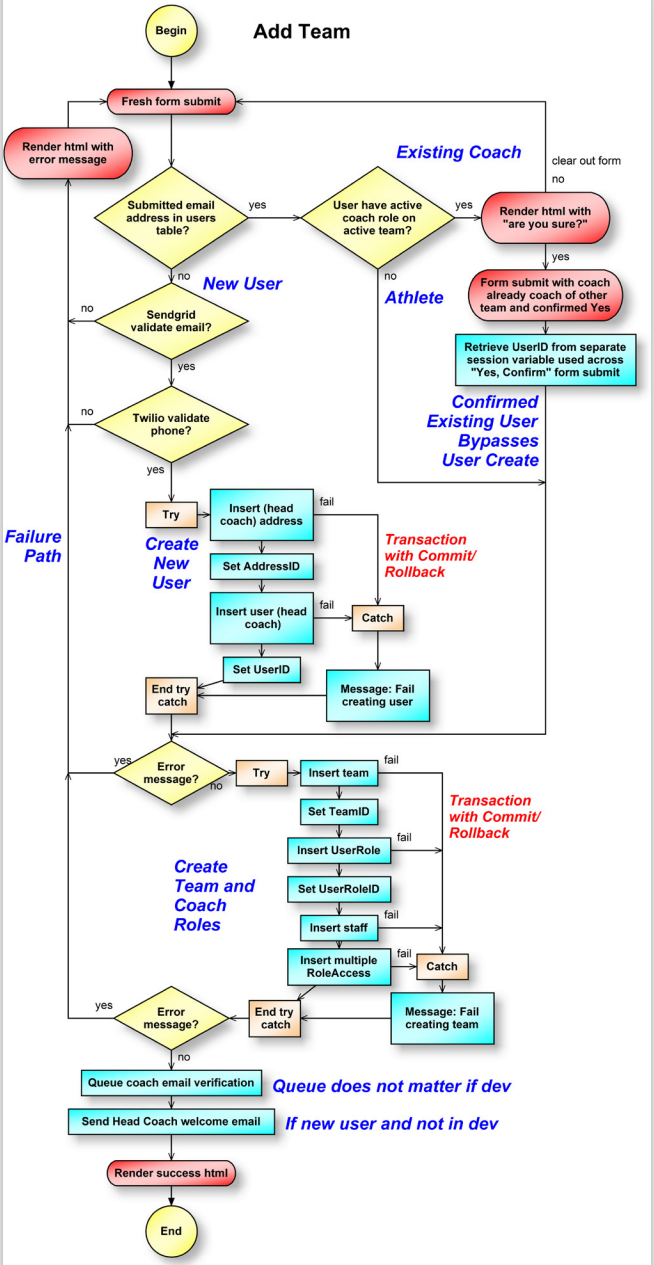
At each of the above steps, if that step fails, we can stop and report the error to the administrator who is adding the new team. We do not need to see if the team already exists; we trust the administrator to know this is a new team. If step 3 fails (create team records), we can safely leave the user record (the new head coach) in our database. The next time through, we should find that user and use the existing record.

The current code (before this rewrite) used try/catch but did not use database transactions. With the rewrite, both step 2 and step 3 each run as a database transaction inside a try/catch. Therefore, the entire step succeeds, or the entire step fails and rolls back. (See Figure 2)

We discovered one other use case. What if the prospective Head Coach is in our system but with a different address or phone number? We decided that adding a new team should *not* alter an existing user record. If the prospective Head Coach is in our system, we use the Head Coach record as-is.

This decision changed our business flow. We first check to see if the prospective Head Coach is in our system. If so, we use the existing record and ignore the submitted form data. Thus, the validation step (step 1 above) only happens when we are creating a new user (Head Coach) record. Figure 2 shows the correct flow, i.e., only do form-data validation when creating a new user record.

Figure 2.



Tribal Knowledge

Should our PHP implementation look like a flowchart? No, probably not! What I did do with the PHP code, however, was to add comments using the exact text from the flowchart's decision boxes. See Listing 1 (on the next page). The coding style is ugly, but given the flowchart graphic, it's easy to understand what the code does and how it does it.

In changing the order-of-processing for our Add Team capability, we changed the "tribal knowledge". Only those present in that conversation know what the new process is. This brings out an important principle of legacy software design:

> *Who touched it last?*

In fact, that's an awful question to ask, with the implication of blaming someone else for the horrible state of our legacy codebase. However, it remains an important question to get answered in the sense of "Who has the tribal knowledge?".

Fortunately, this time around, we have a flowchart showing the business process:

- It shows the intended order of steps
- It shows the two use cases (prospective Head Coach is or is not already in the database)
- It names the database records to create and, therefore, the requisite configuration needed for writing features related to a new team
- It shows the failure paths

The flowchart does *not* dictate implementation. We have no variable names or class diagrams. It's our description of how the business process works. If the code changes, the new code should still implement this diagram. If, on the other hand, the business process changes, this diagram needs to change, and therefore the code implementing the diagram would also change.

Summary

When faced with a feature that was not handling errors adequately, I attempted to draw out the business process. That is, I attempted to make a diagram showing the steps as currently performed. That diagram got so complicated so quickly that it became clear that this convoluted logic was likely part of the problem.

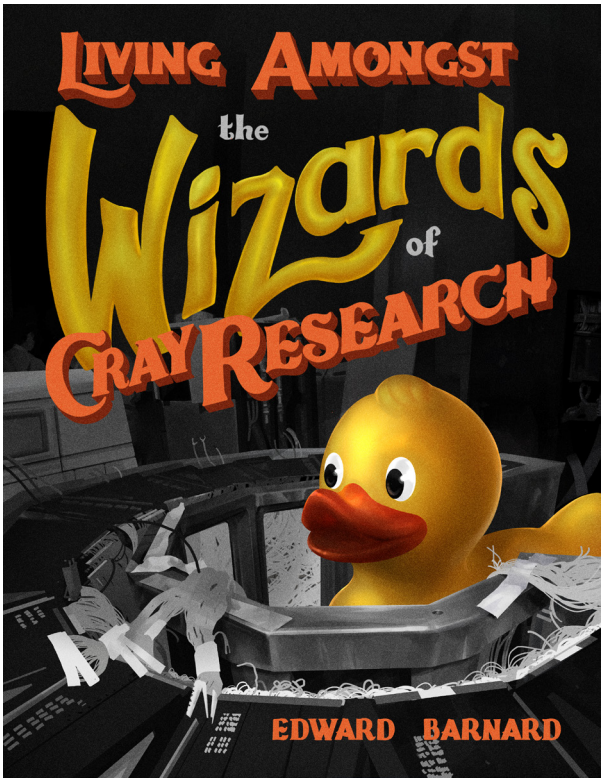
This is *not* always the right approach. In this case, given the problem to solve (the database in an inconsistent state after errors occur), it *was* important to identify the steps and their order of processing.

We changed our discussion to "How should we do this correctly?". That discussion carried the danger of creating new tribal knowledge. We solved that by creating a new flowchart. We reviewed the flowchart to confirm it reflects our discussion. I then rewrote the PHP code to implement that new flowchart.

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

Listing 1.

```
1. <?php
2.
3. /** Submitted email address in users table? */
4. if ($existingUser) {
5.   /** User have active coach role on active team? */
6.   $schools =
7.       $controller->getExistingCoachRoles($UserEmail);
8.   $hasOtherCoachRoles = !empty($schools);
9.   /** Fall through to render with "are you sure?" */
10. } else {
11.   /** Submitted email address NOT in users table */
12.   /** Sendgrid validate email? */
13.   if ($controller->validateEmail($UserEmail)) {
14.     /** Twilio validate phone? */
15.     if ($controller->validatePhone($Phone)) {
16.       /** Create new user */
17.       $addUserResponse = $controller
18.                   ->createNewUser($addTeamRequest);
19.       if ($addUserResponse->isSuccess()) {
20.         $UserID = $addUserResponse->getUserId();
21.       } else {
22.         $message = $addUserResponse->getErrorMessage();
23.       }
24.     } else {
25.       /** Twilio did NOT validate phone */
26.       $message = 'Invalid phone number';
27.     }
28.   } else {
29.     /** Sendgrid did NOT validate email */
30.     $message = 'Invalid email address';
31.   }
32. }
```



RFC: Where To Start

Frank Wallen

If you have been following the progress of PHP since version 5.6, even 7.4, you know there have been a lot of changes. How does that work? How does that happen? How do I get involved? To get you started on your way, let's understand what an RFC is: *Request for Comments*. That doesn't really explain a lot, but it's the beginning of the discussion on a new implementation, change, or feature added or made in the underlying PHP engine by the internals team. This is the place where *all* of those discussions begin.

There are a few stages of a Request for Comments and its progress toward implementation. First, one should create a write-up of how implementation will work and provide supporting statements to convince the readers that it is a useful change that will improve the lives of the PHP community. It's important to understand that your change can impact many coders and communities and will be carefully considered. You don't have to be a C programmer to present an RFC; however, if no one volunteers to code it, the chances of implementing it will be very small. Becoming part of the PHP RFC community and getting to know who's who and what is being done will be crucial. If you need a C programmer for your RFC, then that community is the place to start—where you can find one that supports your idea. If you *are* a C programmer, you will still need to be a part of the community and show that you offer positive and productive changes to the PHP language. To learn more about maintaining and extending PHP, check out the internal references page at wiki.php.net[1].

The first step is to join the internal mailing list. On the mailing lists page, you will find many different mailing lists. There is the **Internals List**[2] itself, but there are others you should join, too, like the **PHP Standardization and Interoperability List**[3]. You would then email your proposal to the mailing list and describe it there, including any developer willing to work on it. Listen to the feedback you receive with a positive and open view; these people can guide you to reality and will be voting on your RFC. That feedback will help you hone your proposal into something fellow PHP developers will look forward to.

Creating the RFC will require registering on the PHP wiki[4]. You can do this before or after you email the internals list, but the wiki is where you will create the actual RFC. When you register, you will need to provide your username and email, of course, but the form will also have a required question about where you will next be sending an email (hint: it's where you need first to send the initial concept of your proposal).

Once you register, the landing page[5] will have numerous links to internal resources—not just the development of an RFC but the current progress of implementations already in development. However, some pages may fall behind on current events, but you can stay on top of those by way of the mailing lists.

If you are interested in contributing to PHP development but don't have a specific RFC in mind, you can browse the list of open RFCs and provide feedback. This can include pointing out potential issues, suggesting improvements, or expressing support for the proposal. Contributing to the RFC process in this way is a valuable method for getting involved and helping shape

1 https://wiki.php.net/internals/references

2 mailto:internals@lists.php.net

3 mailto:standards@lists.php.net

4 https://wiki.php.net/start?do=register

5 https://wiki.php.net/start

the future of PHP. Once your proposal has been refined and is ready for consideration, it will be presented to the PHP internals team. The team will review your proposal and vote on whether to accept or reject it. If your proposal is accepted, it will be implemented in a future version of PHP.

If your proposal is rejected, it's important to remember that the rejection does not mean your idea is bad or that your work was wasted. Instead, it means there were concerns that needed to be addressed, or the proposal was not feasible at this time. Therefore, you can always revise your proposal and submit it again in the future. This rejection could be due to a variety of reasons. Perhaps technical issues need to be resolved before the proposal can be accepted. Whatever the reason, it's crucial to take the feedback you receive seriously and use it to improve your proposal. As you revise your proposal, take the time to address the concerns that were raised in the rejection. Be open to feedback, and consider reaching out to other members of the community for help or advice.

Keep in mind that the process of creating an RFC takes time, and setbacks are normal. However, by remaining committed and persistent, you can make a positive impact on the PHP community. Ultimately, the goal of an RFC is to create a better future for PHP, and your contribution could be a significant part of achieving that goal.

In conclusion, if you are interested in contributing to PHP development, understanding the RFC process is a crucial first step. By joining the internals mailing list, creating a proposal, and seeking feedback from the community, you can help shape the future of PHP and improve the lives of its users. Keep in mind that the process can take time, and there may be setbacks, but your proposal can be refined and eventually implemented. Don't hesitate, get involved, and make your voice heard in the RFC process.

Related Reading

- *PSR Pickup: PSRs And PERs: What's Next?* by Frank Wallen, July 2023.
  https://phpa.me/psr-jul-23
- *PSR Pickup: PER: Coding Style* by Frank Wallen, June 2023.
  https://phpa.me/psr-jun-23

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

Convention over Configuration: Anti-Patterns in Laravel

Matt Lantz

There are often various threads of people ranting on Twitter and other social platforms about anti-patterns and the chaos they produce in applications. In a classic developer argument manner, their arguments are often anecdotal and rife with emotionally driven opinions over conclusive pros and cons arguments. Laravel, like any other code and framework, is filled with highly opinionated code; you buy into some of the opinions or avoid them. Laravel itself is not immune to anti-patterns, nor does it distinctively encourage developers to avoid common ones. However, there are common anti-patterns that Laravel developers can avoid while still utilizing the framework conventionally.

Leaning into some Anti-Patterns

Laravel often gets attacked for its Facade pattern, but various components of the framework can be considered anti-pattern. Does that mean you should avoid them at all costs? No, you should consider them and determine if they fit your needs and architectural vision. Laravel promotes convention over configuration, which was initially introduced by David Heinemeier Hansson of Ruby on Rails. The idea is that there is a sensible default that most code components require to perform their work. When leaning into these defaults, developers have fewer decisions they need to make and can build more effectively as a result. It also promotes a commonality among Laravel projects, rather than every project having its "own" way of handling "X."

Some baked-into Laravel anti-patterns are the global methods such as app, dispatch, view, etc. These global methods though handy, are often magic methods that can reassess code that has already been mapped or reference already loaded data. They may be too magical, but they often enable developers to write code quickly. Furthermore, the app method does a generally elegant job of handling dependency injection and proper bindings, which are set in the service providers enabling developers to avoid having to perform dependency injection in specific places. Though this, too, can be considered a bit of a Golden Hammer, it does help centralize how and where developers should set dependencies.

Laravel also promotes using an Active Record pattern with the Eloquent ORM. In many cases, this is considered an anti-pattern. So too, are PHP traits, yet, Laravel and many community packages lean into traits for adding or separating responsibilities from their Models. Doing so helps reduce model bloating overall and maintains a clear location of responsibilities. Though Eloquent can be remarkably effective in most use cases, it can get a bad rap because of instances where developers are not considering N+1 conditions or over-adding special query methods to their models. Ultimately Laravel leans into Active Record heavily, and avoiding it because it can be considered an anti-pattern will come at a high cost to the development team and knowledge transfers for new team members.

Lastly, at least in this article, many people consider Facades to be a nasty anti-pattern as they violate many of the SOLID principles. The critical point in many cases is that the sub-system of the Facade is tightly coupled to its wrapper. Though there is validity in the critiques of this, in nearly all cases, Laravel Facades are in place to provide clean access to a sub-system of the Laravel application components and, in turn, rely on the Service Providers to handle setting the appropriate bindings.

Most developers using Laravel as a choice lean into these components and ignore the concerns about anti-patterns as these patterns are conventions within the Laravel framework. The more significant issues occur when developers begin letting other anti-patterns slip into the components of their code that they are responsible for.

Avoiding the Anti-Patterns IRL

Bloated controllers and models are frequently discussed in developer communities, and many Laravel apps are susceptible to this anti-pattern. Some developers combat this by implementing Action classes, though this too can become bloated by having everything shoved into actions. Sticking to a convention with CRUD controllers and single-responsibility controllers is best when basic CRUD is insufficient. Utilize Actions only when there is more integration than a basic database transaction or when multiple triggers can initiate the action. Following this helps us avoid the God pattern as well as Functional Decomposition.

Since I'm writing some beautiful classes and Jobs and Events with Listeners in Laravel, one might think I'm avoiding all spaghetti code, but this is simply not the case. Spaghetti code can go beyond ad-hoc files with logic all over the place and can easily bleed across multiple files of a single application. Avoid using any logic inside blade templates, and when checking for permissions, stick to convention and use the Gate systems. Developers can easily let permission and other logic get repeated in applications.

Laravel applications can also fall into the Lava flow trap quite quickly; this is where development teams will abstract certain components of their application, like a core logic system making them framework agnostic. The probability of any application's framework change is slim to nil. Teams are far better off working with conventions so all developers can understand the core systems.

Additionally, some teams get spooked by vendor lock-in. More often than not, these fears are derived from a combination of other anti-patterns more so than an actual vendor lock-in. Cases where Dead-End code is put into place which only works with a specific provider are pretty normal, as most applications are not changing their architecture or infrastructure frequently. However, avoid it when possible, as all technology has a dated lifespan, and there should rarely be an expectation for any tooling to exist for all time. Building complex queries to a query builder that is only compatible with ElasticSearch is committing long-term to ElasticSearch and should be deeply considered. There is no perfect system for this, as community package providers often need help to create ideal interoperability between systems, and they rarely get that type of support.

Overall, most Laravel applications will stick to most conventions and, in turn, will adopt some anti-patterns. The best advice for those teams and developers is to ensure that when you deviate from the convention, you thoroughly document why and how. Many developers will rant and rave about how using any anti-patterns will derail your productivity and create havoc in your team. However, in most cases, it's not the anti-pattern that is the root issue; it's the anti-convention pattern that creates problems. Be aware of the well-documented anti-patterns in software development, but don't stress about killing every variant. Instead, focus on building clean, simple code with few responsibilities, and document it well. By doing this, you will often avoid the most common anti-patterns.

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

Have You Tried Turning It off and...Being Bored?

Beth Tucker Long

When work gets hectic, we push harder at a frantic pace. We work overtime to get everything done. We plow through patches and tickets, slowly chipping away at the mountain of work before us. We know what needs to be done, but what if there's a better way?

When someone is trying to remember something and just can't think of it, they often say, "I know I'll remember it when I'm trying to fall asleep tonight." Common advice when you are stuck on a problem is to take a nap, and maybe you'll solve it in your sleep. Memes highlight "shower thoughts", which are epiphanies about how disparate things are interconnected, which people would never realize under normal circumstances. What is it about sleep and showering that leads to astounding realizations and innovative solutions?

Boredom.

Yes, you heard me correctly. Boredom has been shown to stimulate creativity and problem-solving, and it is severely lacking in our modern-day work environments. Each moment of our work day is packed full. If someone doesn't have any work to do right now, then management starts talking about cuts to staffing to save money. This means we are forced to show progress at all times via emails answered, tickets closed, and code committed.

If we are able to sneak some downtime, we grab our phones to check texts, read articles, or play games. This leads to a system that churns out a lot of "progress" as far as work delivered, but it is not going to deliver the kind of progress that alters markets or brings about significant change.

Our brains need periods of emptiness to filter through and process what we've done and what we've learned. If we are constantly feeding our brain new information and asking it to spit back out what we already know, our brain is too busy to truly be creative. When we have nothing to do but think, our brains begin to make new connections. We are better able to devise solutions for the problems we need to solve.

Studies have shown that boredom not only helps us work through existing problems, but also makes us more creative when we encounter future tasks or problems. People who were put in an environment that induced boredom were shown to significantly outperform those who were kept engaged immediately before being asked to perform a creative task.

So the next time you are headed into a project planning meeting or need to brainstorm a solution for a problem, try sitting bored for ten minutes before you start. It may just be the right piece of unproductivity you need to get the job done.

Related URLS:

- Journal of Experimental Social Psychology: "Approaching novel thoughts: Understanding why elation and boredom promote associative thought more than distress and relaxation" - https://phpa.me/sciencedirect[1]
- Creativity Research Journal: "Does Being Bored Make Us More Creative?" - https://phpa.me/tandfonline[2]
- Mayo Clinic Health System: "Boost your brain with boredom" - https://phpa.me/mayoclinic[3]

Beth Tucker Long is a developer and owner at Treeline Design[4], a web development company, and runs Exploricon, a gaming convention, along with her husband, Chris. She leads the Madison Web Design & Development and Full Stack Madison user groups. You can find her on her blog (http://www.alittleofboth.com) or on Twitter @e3BethT

1 https://phpa.me/sciencedirect
2 https://phpa.me/tandfonline
3 https://phpa.me/mayoclinic
4 http://www.treelinedesign.com