

php[architect]

The Magazine for PHP Developers

```
#!/usr/bin/env php
<?php
    echo "Hello, CLI";
```

COMMAND LINE PICASSO

Creating Beautiful CLI Applications

Also Inside:

Education Station - Object oriented visibility

PHP Puzzles - Shell Sort

Security Corner - PHP, meet passkeys

DDD Alley - Bounded Fix

Barrier-Free Bytes - Done For You

Artisan Way - Alternative Architecture in Laravel

RADAR - WebAuthn: The Future to Securing Applications

Readable Code - Is Your Code Abstracted Enough?

finally{} - Gratefully Looking Back

JET
BRAINS



PhpStorm

Enjoy productive PHP

jetbrains.com/phpstorm



The Web Developer's

SECRET WEAPON!

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

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

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

DevOps monitoring, for developers. *gasp!*

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

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

Exception Monitoring

Delight your users by proactively monitoring for and fixing errors.

Uptime Monitoring

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

Check-In Monitoring

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



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

CONTENTS

NOVEMBER 2023
Volume 22 - Issue 11

```
#!/usr/bin/env php
<?php
    echo "Hello, CLI";
```



php[architect]

- | | | | |
|-----------|---|-----------|---|
| 2 | Command Line Picasso | 33 | WebAuthn: The Future to Securing Applications |
| 3 | Creating Beautiful CLI Applications in PHP with MiniCLI and Termwind | | RADAR |
| | Wendell Adriel | | Matt Lantz |
| 17 | Bounded Fix | 36 | Done For You |
| | DDD Alley | | Barrier-Free Bytes |
| | Edward Barnard | | Maxwell Ivey |
| 21 | Alternative Architecture in Laravel | 39 | Shellsort |
| | Artisan Way | | PHP Puzzles |
| | Steve McDougall | | Oscar Merida |
| 25 | Object Oriented Visibility | 42 | Is Your Code Abstracted Enough To Minimize Load? |
| | Education Station | | Readable Code |
| | Chris Tankersly | | Christopher Miller |
| 30 | PHP, Meet Passkeys | 52 | Gratefully Looking Back |
| | Security Corner | | finally}} |
| | Eric Mann | | Beth Tucker Long |

Edited by Thanksgiving Turkeys

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

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

Subscriptions

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

Advertising

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

Contact Information:

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

Print ISSN 1709-7169
Digital ISSN 2375-3544

Copyright © 2023—PHP Architect, LLC
All Rights Reserved

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

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

Command Line Picasso

Jacki Congdon

It's November already, in America, it's a time for expressing our gratitude to friends and family.

Gratitude: an acknowledgment of having received something good from another, as defined by Merriam-Webster. We have so much gratitude for all of our contributors and readers alike. We receive something good from each of you throughout the year. Our contributors provide the relevant content to keep us learning/growing in our professional development. And our readers offer important feedback. PHP Architect would only be here with both of these crucial components, and we thank you all.

This month's feature, by Wendell Adriel, is *'Creating Beautiful CLI Applications in PHP with MiniCLI and Termwind'*. You'll want to take some time to read this in full. Wendell gives an in-depth look at how PHP can be useful for things beyond web applications.

Steve McDougall is diving into software architecture in this month's Artisan Way with *'Alternative Architecture in Laravel'*. He will take you through MVC, ADR, DDD, and VSA. Steve is going to break down the various uses that best fit each micro-service. Find out some of the pros and cons to see if this approach is best for you and your team.

This month, in our column Barrier Free Bytes, Maxwell Ivey is giving us his take on websites/services that provide the option of 'done-for/with-you' and 'do it yourself' services in his article, *'Done For You'*. Max shares some of his background with receiving advice from disability advocates to be fiercely independent versus the advice he received from his own family, never to be afraid of asking for help.

Following up from last month's first article in the new column, Radar, Matt Lantz is giving us *'WebAuthn: The Future to Securing Applications'*. WebAuthn, short for web authentication, is the latest and greatest in online security that allows password-less authentication.

Legacy code can prove challenging for making/adding any changes. In our DDD Alley column, Edward Barnard will show us the way with *'Bounded Fix'*. Ed will walk you through ways that he has been able to make the changes needed to existing codebases without breaking things in the process.

Up next in Education Station, Chris Tankersly is bringing us *'Object Oriented Visibility'*. Chris will take you through programming, the development of PHP as an object oriented language, and visibility as it has evolved.

Next up in Security Corner, Eric Mann will introduce you to Passkeys in *'PHP Meet Passkeys'*. Eric is going to share the definition and evolution of Passkeys. In this article, Eric points out that passwords, multifactor authentication, and security in general don't need to be complicated.

Oscar Merida is back in our Puzzles column with *'Shell Sort'*. Oscar follows up on comb and bubble sort and shows how shell sort improves the execution of the Insertion Sort. He shows us the algorithm and solution. Additionally, you'll see a full comparison of all the sort methods used so far. And, as expected, Oscar leaves us with a new challenge for next month.

Christopher Miller is back in our Readable Code column with another great read, *'Is Your Code Abstracted Enough to Minimize Load?'*. This is a great read detailing the importance of abstraction in your code. It simplifies the entire process from user experience to development on the back end. Christopher goes on to discuss how you can minimize load for yourself, meaning your overall mental load of reading and understanding the code. You can check out examples of abstraction and even the risks of overly abstracted code.

And finally {}, Beth Tucker Long brings us *Gratefully Looking Back*.

Write For Us

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

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

Stay in Touch

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

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

Download the Code Archive:

https://phpa.me/November2023_code

Creating Beautiful CLI Applications in PHP with MiniCLI and Termwind

Wendell Adriel

Usually, when we think about PHP programming, our minds automatically drift toward Web Applications. PHP is widely known for web-oriented solutions. However, it's less recognized that PHP also has robust capabilities to build powerful CLI applications.

Command Line Interface (CLI) applications are essential tools in the realm of software development. They provide a user interface that interacts with a software or operating system using commands. Unlike Graphical User Interface (GUI) applications, where users interact with software using graphical icons and visual indicators, CLI applications rely solely on textual input and output.

Before PHP 4.3.0, creating CLI applications with PHP wasn't straightforward. It involved several workarounds and resulted in applications that weren't very efficient. However, the introduction of a new CLI SAPI (Server Application Programming Interface) from PHP 4.3.0 onwards revolutionized PHP usage in the CLI environment. This dedicated SAPI is very similar to the one that PHP uses for web programming, but it was engineered to perform exceptionally well in a CLI setting.

PHP CLI applications can be scripts for automation, admin tasks, tests, and even complex systems running in the background as daemons. They can also be trusted to manage file system tasks, perform network communications, or handle databases.

The beauty of CLI PHP applications is that they integrate easily with the existing systems and standard workflows. Whether you're building a small script to automate mundane tasks or complex systems, PHP for CLI applications provides powerful tooling while being lightweight and efficient.

Indeed, PHP's utility extends far beyond web application development. Although much less discussed, PHP's CLI application development capabilities make it a highly versatile programming language. While PHP is renowned for its stateless HTTP protocol for the web, it equally excels in managing the stateful, interactive shell environment of CLI applications. No matter what the use case, PHP's broad array of built-in functions and extensive library ecosystem ensure that PHP developers are readily armed to tackle any project that comes their way—including those working in a CLI context.

Creating and Running Our First CLI Script in PHP

PHP allows us to create simple CLI scripts, complex CLI applications, and everything in between. But before we start creating a full CLI application, let's learn how we can create

and run simple CLI scripts with PHP. For the examples that we are going to see in this article, we are going to use the latest version of PHP, version 8.2.

Create a folder named `hello-cli` in your machine, and inside this folder, create a file named `hello.php`:

```
mkdir hello-cli && cd hello-cli && touch hello.php
```

Now, open the `hello.php` file and add this content to it:

```
<?php
echo "Hello, CLI";
```

Save the file, go to the terminal, and run:

```
php hello.php
```

You are going to see the output `Hello, CLI` in your terminal. Done. We created and ran our first CLI script in PHP, but there are still some things we can do to make this even better. As you can see, when running our script, we need to use the PHP binary to execute it. What if we could execute it without the need to explicitly use the PHP binary to do so? We can do that, and I'm going to show how.

First, rename your `hello.php` file to `hello` only (without extension):

```
mv hello.php hello
```

Then open it, and update the content to this and save:

```
#!/usr/bin/env php
<?php
echo "Hello, CLI";
```

You can see that we just added a new line at the beginning of the file: `#!/usr/bin/env php`. This line is responsible for telling our OS that this file, which is now an executable one, should use the PHP binary when executed.

Now try running it in your terminal:

```
./hello
```

You're probably going to see an error message like this:

```
permission denied: ./hello
```

That's because our file is not set to be executable. We need to make a final adjustment setting the permissions on it to make sure we can use it as an executable:

```
chmod +x ./hello
```

Now that our file is an executable, we can execute it as we tried before:

```
./hello
```

This time, you will see the output `Hello, CLI` in your terminal. Much better, right? We created a CLI script in PHP as an executable file that we can run as any other executable without the need to know that this was built with PHP. But there are still some other details that we need to pay attention to while creating CLI scripts and applications with PHP. Let's improve our script even more.

Open the script and update it to be like this: (See Listing 1)

You can see that we added two things to the script: an `if` check at the beginning and wrapped the script logic in a `try/catch` block. But why we did we make these changes?

Let's understand the `if` check first. The `php_sapi_name` function returns a lowercase string describing the type of interface that PHP is using, so here we check if the context is not the CLI—we won't execute anything.

Now, the `try/catch` block is self-explaining. If the script runs into any error, it will print the exception and return 1, meaning that the process failed. When a CLI script runs and

Listing 1.

```
1. #!/usr/bin/env php
2. <?php
3.
4. if (php_sapi_name() !== 'cli') {
5.     exit;
6. }
7.
8. try {
9.     echo "Hello, CLI";
10.    return 0;
11. } catch (Throwable $exception) {
12.     echo "An error occurred\n";
13.     echo $exception->getMessage();
14.     return 1;
15. }
```

has a return different than 0, it means that the script failed. That's why the last line of the `try` block is returning 0, and the last line of the `catch` block is returning 1.

Now, go to the terminal and run once again our script:

```
./hello
```

You'll see again the `Hello, CLI` as output. Now, let's see how the script behaves when an exception occurs. Update the script to be like this: (See Listing 2)

Listing 2.

```
1. #!/usr/bin/env php
2. <?php
3.
4. if (php_sapi_name() !== 'cli') {
5.     exit;
6. }
7.
8. try {
9.     throw new Exception('DANGER - CLI ERROR!');
10.    echo "Hello, CLI";
11.    return 0;
12. } catch (Throwable $exception) {
13.     echo "An error occurred\n";
14.     echo $exception->getMessage();
15.     return 1;
16. }
```

Now, go to the terminal and run once again our script:

```
./hello
```

You'll see an output like this:

```
An error occurred
DANGER - CLI ERROR!
```

That's great, right? Now we have a way to see the errors from our script! But you're probably thinking that our script is missing something, and you're right. We can do more than just display data; we can also gather user input to make our scripts more dynamic. So, let's update our script to get the user name and display a custom message! (See Listing 3)

Listing 3.

```
1. #!/usr/bin/env php
2. <?php
3.
4. if (php_sapi_name() !== 'cli') {
5.     exit;
6. }
7.
8. try {
9.     $name = readline("What's your name?\n> ");
10.    echo "Hello, {$name}\n";
11.
12.    return 0;
13. } catch (Throwable $exception) {
14.     echo "An error occurred\n";
15.     echo $exception->getMessage();
16.
17.    return 1;
18. }
```


To get user input in our CLI scripts or applications, we can use the `readLine` function. It takes a string as an argument to be the prompt we show to the user, and the user input is going to be returned by it.

Now, save your script and run it again:

```
./hello
```

You'll see an output like this:

```
What's your name?
>
```

Type your name and hit enter and you'll see an output like `Hello, Wendell`. Beware that if the user hits enter without giving any value, the `readLine` function will return `false` as a value instead of a string. So like any other application, make sure to validate the user input, but for the sake of simplicity of this example, we won't add this validation.

And that's it; we just built our first **PHP-powered CLI script!** This is just a really simple example of how you can create CLI scripts with PHP. If you're not a huge fan of **bash**, you can start creating your own scripts with PHP and sharing them with your teammates now!

PHP Libraries for CLI Applications

Symfony Console

The **Symfony Console** library is a powerful tool that provides a robust framework for building command-line applications. This library, part of the larger **Symfony Framework**, offers a lot of flexibility and enhances PHP's core ability to create CLI applications. With it, developers can define commands and command parameters, handle user inputs and outputs, leverage its built-in testing capabilities, and do much more in creating feature-rich CLI utilities. This library is also used by other libraries as a base dependency, being one of the most famous libraries for CLI applications in PHP.

You can read more about it here: <https://phpa.me/symfony-comp-console>¹.

Laravel Zero

Laravel Zero is an open-source, streamlined version of Laravel that is specifically designed for building high-quality command-line applications. While **Laravel** is well-known for its extensive applications for the web, **Laravel Zero** targets the crafting of lightweight yet powerful CLI applications. Built on top of the foundational Laravel components, it adds a host of convenient features dedicated to enriching the CLI experience.

It provides features such as task scheduling, automatic testing, and Laravel's extensive **Eloquent ORM**. **Laravel Zero** also supports application compiling into a **PHAR archive**, making distribution easier.

It provides features such as task scheduling, automatic testing, and Laravel's extensive **Eloquent ORM**. **Laravel Zero** also supports application compiling into a **PHAR archive**, making distribution easier.

You can read more about it here: <https://laravel-zero.com>².

Laravel Prompts

Laravel Prompts is a highly useful library that can be extensively used for the development of command-line applications. It presents the functionality of a more interactive set of tools for collecting user input in CLI applications beyond the traditional command-line prompts. It has been designed to enhance the Laravel artisan console capabilities (but not limited to it, since you can use it on other PHP projects), and it intelligently extends the built-in prompt methods to provide richer user interaction.

Offering features that are **common in web applications** to CLI apps like selects, checkboxes, tables, progress bars, placeholders and data validation, the library aims to create beautiful and rich CLI apps with ease.

You can read more about it here: <https://laravel.com/docs/10.x/prompts>³.

MiniCLI

The **MiniCLI** library is a micro-framework that offers developers a **minimalist** and **highly efficient** toolset to create command-line applications. As it **doesn't require any dependencies**, **MiniCLI** makes for a perfect option when building **lightweight, standalone CLI PHP applications**. It provides a slim, portable solution that is ideal for projects that need a slim and highly customizable CLI tool.

Despite its minimalist design, it doesn't compromise productivity. It provides a structured way to create and organize commands alongside a variety of helpers to handle user input and print colored output to the terminal with support for themes.

You can read more about it here: <https://docs.minicli.dev>⁴.

Termwind

Termwind is a unique library tailored for crafting beautiful command-line applications. Heavily inspired by **Tailwind CSS**, a utility-first CSS framework, **Termwind** brings a similarly structured approach to the CLI world. With **Termwind**, printing styled text to the console becomes a breeze through its stylish, fluent API. The library provides a means to control and format console text output with the same level of nuance and freedom traditionally found in web development.

Termwind extends PHP's existing output capabilities in a way that is both familiar to **Tailwind** users and intuitive to developers new to the framework. It offers the ability to control text colors, background colors, formatting options, and alignment. You can add borders to elements, control spacing, and even handle inline elements—essentially, it brings styles to your terminal!

¹ <https://phpa.me/symfony-comp-console>

² <https://laravel-zero.com>

³ <https://laravel.com/docs/10.x/prompts>

⁴ <https://docs.minicli.dev>

You can read more about it here: <https://github.com/nunomaduro/termwind>⁵.

Creating CLImage, a Helper Tool for Images with Minicli and Termwind

Now that we already created our first CLI script with PHP and checked some libraries used to build CLI applications, let's create our first full CLI application! For this, we will use two of the libraries described in the previous section: **MiniCLI** and **Termwind**.

MiniCLI is my go-to library when building CLI applications in PHP because it's a lightweight yet powerful micro-framework with many features to build fast, small, and amazing CLI applications. Combining it with **Termwind** is a spicy combination because we can easily create beautiful and rich CLI interfaces with Termwind.

We are going to create a small but useful helper application that can resize and convert images called **CLImage**. For that, we will use an official template from **MiniCLI** that already supports **Termwind** and **Plates** (a minimalist and lightweight template engine for PHP) called **MiniTerm**: <https://github.com/minicli/miniterm>⁶.

Miniterm Overview

Let's start by creating our project:

```
composer create-project minicli/miniterm CLImage &&
cd CLImage
```

Before starting to code our application, let's take a look at the project structure and the demo commands to understand how **MiniCLI** and **Termwind** work together.

The project has **four main folders**:

- **app** - This is where your application code lives. Here, you'll find four subfolders:
 - **Command** - All your application commands
 - **Config** - These are configuration classes that can change the behavior of your application. This template already has two configuration files that set how **Termwind** is used.
 - **Services** - These are services that you can create to host business logic or to configure **3rd party libraries** into your application. This template already has two services: one for **Termwind** and another one for **Plates**.
 - **Views** - All view files that you can use the **Plates** engine with **Termwind** to create beautiful outputs for your application.
- **config** - This is where all configuration files for your application live. This template already comes with three configuration files, but you can **create additional configuration** files and access the values from it in your application with a **config helper** as well as create additional properties inside the existing configuration files (similar to what we have in **Laravel**).
- **app.php** - This is the main configuration file

⁵ <https://github.com/nunomaduro/termwind>

⁶ <https://github.com/minicli/miniterm>

for the application, where you can set the application name, the path(s) for the commands, and theme and disable/enable the debug mode.

- **logging.php** - This is where you can configure how the logs will work for your application. You can define the type (single log file or daily log file), the default log level and the time-stamp format for the logs.
- **services.php** - This is where you can configure all the services for your application. Those services are going to be injected into the application container so you can use them in your application through their names (keys).
- **logs** - This is where all your log files are going to be created.
- **tests** - This is where live all the application test files.

The entry point for our application is the `minicli` file at the root of the project. When you open this file, you'll see that it has a lot of things in common with the script we created earlier in this article. Go to the terminal and run the application:

```
./minicli
```

You're going to see an output like this:

```
MiniTerm - MiniCLI Application Template powered with
Termwind and Plates
```

If you pay attention, that's printing what we have defined in the `app_name` property of the `config/app.php` file. So, the first thing we will do is update this. Update this property to `CLImage---Helper Tool for Images`. Save the changes, and let's also rename our executable file:

```
mv ./minicli CLImage
```

Now run:

```
./CLImage
```

You're going to see an output like this:

```
CLImage - Helper Tool for Images
```

Understanding Minicli Commands

Now that we have set our application name and renamed the executable file, let's take a look at some of the demo commands that come with the template. Run this command:

```
./CLImage demo
```

You are going to see an output like this: (See Figure 1)

Figure 1.

```
>
./CLImage demo
```

```
INFO Run ./minicli help for usage help.
```

Let's understand how this command is executed. **MiniCLI** uses a file-based "routing" to define the commands. You can see that inside the `app/Command` folder, we have a folder named `Demo`. For **MiniCLI**, this is how you define a command name. You can also see that inside the `Demo` folder, we have multiple files. That's how we define **sub-commands** with **MiniCLI**. Among the files inside this folder, we have a file named `DefaultController.php`. When you run a command without any other sub-commands, this is the class responsible for the root command. So when we ran `./CLIImage demo`, the class being executed was the `DefaultController`. Let's check it: (See Listing 4)

This is a pretty straightforward command. It extends the `BaseController` from the template that extends the base controller from the **MiniCLI** library itself and adds some methods to use **Termwind** and **Plates**, like the `render` method. As you can see here, the `render` method accepts an HTML string with some **Tailwind** CSS classes; you can check which ones are supported in the **Termwind** docs to style the output we display in the terminal.

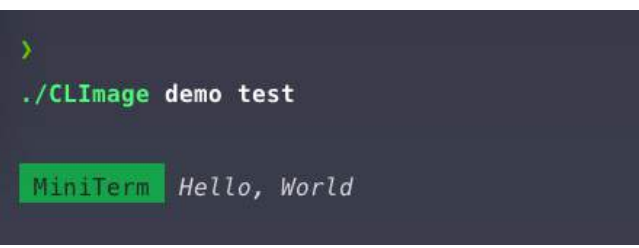
Using Command Parameters

Now, let's take a look at another example that uses command parameters. Run this:

```
./CLIImage demo test
```

You'll see an output like this: (See Figure 2)

Figure 2.



```
>
./CLIImage demo test

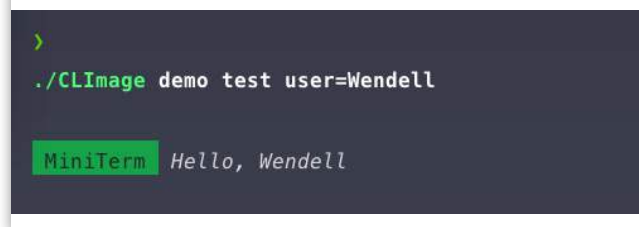
MiniTerm Hello, World
```

Now run the same command, but passing an user param:

```
./CLIImage demo test user=YOUR_NAME
```

You'll see an output like this: (See Figure 3)

Figure 3.



```
>
./CLIImage demo test user=Wendell

MiniTerm Hello, Wendell
```

You can see that the command is using the user param to customize the message, so let's see how that's done. As you can

Listing 4.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\Command\Demo;
6.
7. use App\Command\BaseController;
8.
9. class DefaultController extends BaseController
10. {
11.     public function handle(): void
12.     {
13.         $this->render(<<<HTML
14.             <div class="py-2">
15.                 <div class="px-1 bg-cyan-600">INFO</div>
16.                 <span class="ml-1">
17.                     Run <span class="font-bold italic">
18.                         ./minicli help
19.                     </span> for usage help.
20.                 </span>
21.             </div>
22.             HTML);
23.     }
24. }
```

imagine, since we are using a **sub-command** called `test` from the `demo` **command**, the class responsible for that **sub-command** is the `TestController` inside the `Demo` folder. (See Listing 5 on the next page)

Checking the code above you can see that the only thing different that we have here is the call to two methods: `hasParam` and `getParam`. These are methods that **MiniCLI** provides for checking and getting input data directly from the command calls.

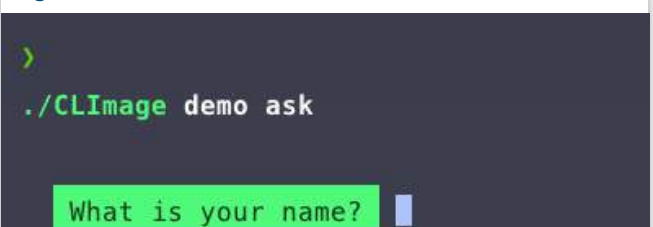
Getting User Input with Termwind

Now that we know how to handle user input from the command call, let's see another example to learn how to ask for user input when running commands. Run this command:

```
./CLIImage demo ask
```

You'll see an output like this: (See Figure 4)

Figure 4.



```
>
./CLIImage demo ask

What is your name?
```

Listing 5.

```

1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\Command\Demo;
6.
7. use App\Command\BaseController;
8.
9. class TestController extends BaseController
10. {
11.     public function handle(): void
12.     {
13.         $name = $this->hasParam('user') ?
14.             $this->getParam('user') : 'World';
15.
16.         $this->render(<<<HTML
17.             <div class="py-2">
18.                 <div class="px-1 bg-green-600">MiniTerm</div>
19.                 <em class="ml-1">
20.                     Hello, {$name}
21.                 </em>
22.             </div>
23.             HTML);
24.     }
25. }

```

Type your name and press enter. You'll see an output like this: (See Figure 5)

Figure 5.

```

>
./CLImage demo ask

What is your name? Wendell

MiniTerm Hello, Wendell

```

Now, you can see that instead of getting user input on the command call, we are getting the user input during the command execution. Let's see how we can do that. Open the AskController inside the Demo folder. (See Listing 6)

In the code above, you can see that we have a call to a new ask method. This method is really similar to the render method; the only difference is that it **waits for user input** and **stores it into a variable**.

Creating Views with Plates

We already know how to handle user input during the command execution. But we are missing one thing. We saw that this template uses the **Plates** template engine, but we

Listing 6.

```

1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\Command\Demo;
6.
7. use App\Command\BaseController;
8.
9. class AskController extends BaseController
10. {
11.     public function handle(): void
12.     {
13.         $name = $this->ask(<<<HTML
14.             <span class="mt-1 ml-2 mr-1
15.                 bg-green px-1 text-black">
16.                 What is your name?
17.             </span>
18.             HTML);
19.
20.         $this->render(<<<HTML
21.             <div class="py-2">
22.                 <div class="px-1 bg-green-600">MiniTerm</div>
23.                 <em class="ml-1">
24.                     Hello, {$name}
25.                 </em>
26.             </div>
27.             HTML);
28.     }
29. }

```

didn't see any examples of how to use it. So let's check one more command that comes with **MiniTerm**:

```
./CLImage demo table
```

You'll see an output like this: (See Figure 6)

Figure 6.

```

>
./CLImage demo table

+-----+-----+-----+
| Header 1 | Header 2 | Header 3 |
+-----+-----+-----+
| 1         | 7         | other string 1 |
| 2         | 3         | other string 2 |
| 3         | 10        | other string 3 |
| 4         | 4         | other string 4 |
| 5         | 3         | other string 5 |
| 6         | 2         | other string 6 |
| 7         | 10        | other string 7 |
| 8         | 5         | other string 8 |
| 9         | 7         | other string 9 |
| 10        | 1         | other string 10 |
+-----+-----+-----+

```

Let's check how this table is created in the TableController inside the Demo folder. (See Listing 7 on the next page)

Listing 7.

```

1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\Command\Demo;
6.
7. use App\Command\BaseController;
8.
9. class TableController extends BaseController
10. {
11.     public function handle(): void
12.     {
13.         $headers = ['Header1', 'Header2', 'Header3'];
14.         $rows = [];
15.
16.         for ($i = 1; $i <= 10; $i++) {
17.             $rows[] = [
18.                 (string) $i,
19.                 (string) rand(0, 10),
20.                 "other string {$i}",
21.             ];
22.         }
23.
24.         $this->view('table', [
25.             'headers' => $headers,
26.             'rows' => $rows
27.         ]);
28.     }
29. }

```

In the code above, you can see that we are just creating some random values for a table, and then we are calling a view method that accepts two parameters: the first one is the view filename, and the second is an array with the data that we want to pass to this view. As we saw earlier, all the views are stored in the `app/Views` folder. Open the `table.php` file inside it: (See Listing 8)

As you can see, this is a really simple HTML table, but you can create complex views using any classes supported by **Termwind** to style the output of your commands. If this syntax is unfamiliar, check out the **Plates** template engine docs here: <https://platesphp.com>⁷.

Creating Our Service

Now that we have seen how **MiniCLI** and **Termwind** work and how the **MiniTerm** template is structured, let's start creating our **CLIImage** application. The first thing that we are going to create is a service that's going to be responsible for image handling. For our application, we will rely upon the **Imagick PHP extension**, so if you don't have it installed, check the docs on how to install it: <https://phpa.me/symfony>⁸. If you have `pecl` installed, it's as easy as running:

```
pecl install imagick
```

Listing 8.

```

1. <table>
2.     <thead>
3.         <tr>
4.             <?php foreach ($headers as $header): ?>
5.                 <th><?=$this->e($header) ?></th>
6.             <?php endforeach; ?>
7.         </tr>
8.     </thead>
9.     <tbody>
10.        <?php foreach ($rows as $row): ?>
11.            <tr>
12.                <?php foreach ($row as $cell): ?>
13.                    <td><?=$this->e($cell) ?></td>
14.                <?php endforeach; ?>
15.            </tr>
16.        <?php endforeach; ?>
17.    </tbody>
18. </table>

```

Let's create our service. Go to the root of the project and run:

```
touch app/Services/ImageService.php
```

Update it to be like this:

```

<?php declare(strict_types=1);
namespace App\Services;

use Minicli\App;
use Minicli\ServiceInterface;

final class ImageService implements ServiceInterface
{
    public function load(App $app): void
    {
        // Nothing to do here
    }
}

```

In the code above, we create a skeleton of our service. All services must implement the `Minicli\ServiceInterface` and define a `load` method. This method is needed if you need to set anything for your service to correctly function in the application when it's loaded into the **container**. Our `ImageService` won't need anything, so we can leave it empty. Now, let's register our service in our application. Open the `config/services.php` file and add our service there like this:

⁷ <https://platesphp.com>

⁸ <https://phpa.me/symfony>

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

use App\Services\ImageService;
use App\Services\PlatesService;
use App\Services\TermwindService;

return [
    /*****
     * Application Services
     * -----
     *
     * The services to be loaded for your application.
     *****/

    'services' => [
        'termwind' => TermwindService::class,

        'plates' => PlatesService::class,

        'image' => ImageService::class,
    ],
];
```

With our service registered, we can call any public method of the service like this in our commands: `$this->app->image->METHOD`.

Our application will have two commands: one to **resize** images and another to **convert** images. So, let's update our service to handle these. Open the ImageService class that we created and update it to be like this: (See Listing 9)

Let's understand our service. It has three **public** methods:

- **info** - Used to get information from an image. We are going to use this to print information on the images in the terminal.
- **resize** - Used to create a copy of an image with the given **width** and **height**.
- **convert** - Used to create a copy of an image with the given **format**.

The methods are really simple and we have some common helper methods and classes that we are using to make the code cleaner and reusable.

We have a **read** method that's responsible for creating an **Imagick** object from the image path that we are going to provide. The **Imagick** object is provided by the **imagick** extension, and it will allow us to resize and convert images without the need for any additional PHP library.

All three public methods from this service also use an **ImageInfo** class, which is just a simple **DTO** that we will use to wrap information about an image. Let's create this class. First, create a **Support** folder inside the **app** folder and create the file inside it:

```
mkdir app/Support && touch app/Support/ImageInfo.php
```

Then update this file to be like this:

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

namespace App\Support;

final readonly class ImageInfo
{
    public function __construct(
        public string $filename,
        public int $width,
        public int $height,
    ) {
    }
}
```

As you can see, it's just a simple **DTO** to wrap information about an image. We use the new feature from **PHP 8.2** that allows us to mark a whole class as **readonly**, making all of its properties **readonly**.

The next thing we need is to create the **ImageFormat** class, which, in this case, is an **Enum** that we will use to describe the formats that we will support in **CLImage**. First, create an **Enums** folder inside the **app** folder and create the file inside it:

```
mkdir app/Enums && touch app/Enums/ImageFormat.php
```

Then update this file to be like this:

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

namespace App\Enums;

enum ImageFormat: string
{
    case JPEG = 'jpeg';
    case PNG = 'png';
    public static function allowedFormats(): array
    {
        return array_map(
            fn ($item) => $item->value, self::cases()
        );
    }
}
```

You can see that we are creating a **BackedEnum** with the formats that we will support, in this case, just **JPEG** and **PNG** for this example. But **Enums** are great for not only describing values but also putting some logic that is related to these values. In this case, we have a static function, **allowedFormats**, that's going to return an array with the values of all the cases listed in our **Enum**, which is going to be very handy in our application.

The Resize Command

Now that we have our **ImageService** complete, we can start working on the command that's going to be used for **resizing** images. As we saw earlier in this article, **MiniCLI** uses a **file-based convention** for naming the commands. So let's create a folder named **Resize** inside the **app/Command** folder, and then inside this folder, let's create a class named **DefaultController** since the command is not going to have sub-commands:

```
mkdir app/Command/Resize &&
touch app/Command/Resize/DefaultController.php
```

Now open the newly created `DefaultController` and update it to be like this: (See Listing 10 near end of the article)

The command is not complex; we are just getting some input from the user using the `ask` method that we saw earlier, doing some **validations** on the input given by the user, and using the `ImageService` through the `app` instance: `$this->app->image->resize` to resize the image and print some information about the process to the user.

Now you can see that we have two **Custom Exceptions**: `FileNotFoundException` and `InvalidDimensionException` that we are using. **Custom Exceptions** are the best and recommended way of handling errors on **MiniCLI** because it's going to use these exceptions to print messages to the user if something goes wrong.

Let's create those. First, create a folder called `Exceptions` inside the `app` folder, and then let's create these two exception classes inside it:

```
mkdir app/Exceptions &&
touch app/Exceptions/FileNotFoundException.php &&
touch app/Exceptions/InvalidDimensionException.php
```

These custom exceptions are going to be really simple; we are just going to extend the base `Exception` class from PHP and add a custom message to them.

Let's start with the `FileNotFoundException`. Update it to be like this:

```
<?php declare(strict_types=1);
namespace App\Exceptions;
use Exception;

final class FileNotFoundException extends Exception
{
    public function __construct()
    {
        parent::__construct('File does not exist!');
    }
}
```

Now update the `InvalidDimensionException` to be like this:

```
<?php declare(strict_types=1);
namespace App\Exceptions;
use Exception;

final class InvalidDimensionException extends Exception
{
    public function __construct(int $max)
    {
        parent::__construct(
            "The value must be between 10 and {$max}"
        );
    }
}
```

Now that we already have our **Custom Exceptions** created, we are going to create some **helper** methods that are being used in this command that were extracted to be reused. You can see that we have these methods being called: `buildQuestion`, `printImageInfo`, and `successMessage` that are not defined

in this class. That's because we want to have these methods available also for our `convert` command, so we are going to create them in the `BaseController` that all of our commands extend from. Open the `app/Command/BaseController.php` class and update it to be like this: (See Listing 11 on last page)

Here in this file, we are adding the three methods mentioned above:

- `buildQuestion` - Used to format how a question will be displayed in the terminal styled with **Termwind**.
- `successMessage` - Used to display a success message in the terminal styled with **Termwind**.
- `printImageInfo` - Used to display a table with information from an image in the terminal using **Termwind** and **Plates** with the table view that already comes with the **MiniTerm** template.

Now, the last detail is that you can see when calling the `resize` method from the `ImageService`, we are passing as the `outputPath` parameter this: `$this->app->config->output_path`. Here, we are accessing the application configuration that we define in our configuration files inside the `config` folder. Open the `config/app.php` file and add this new `output_path` configuration:

```
<?php declare(strict_types=1);
return [
    /*
     * Application Settings
     * -----
     * These are the core settings for your application.
     */
    'app_name' => 'CLImage - Helper Tool for Images',
    'app_path' => [
        __DIR__ . '/../app/Command',
        '@minicli/command-help'
    ],
    'theme' => '',
    'debug' => true,
    'output_path' => __DIR__ . '/../output',
];
```

Here we are setting the **output path** to be a folder named `output` at the root of the project. We don't have this folder yet, so let's create it:

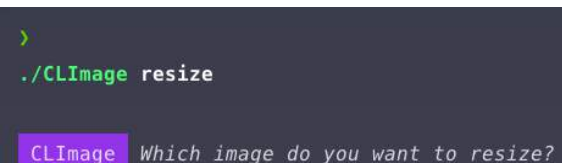
```
mkdir output
```

Now, with our `resize` command finished, let's test it! Go to the terminal and run:

```
./CLImage resize
```

You'll see an output like this: (See Figure 7)

Figure 7.



```
>
./CLImage resize
CLImage Which image do you want to resize?
```


Type the path for a non-existing file and press enter. You'll see an output like this: (See Figure 8)

Figure 8.

```
>
./CLIImage resize

CLIImage Which image do you want to resize?
test

An error occurred:
The given file does not exist!
```

You can see that this is the error message defined in our `FileNotFoundException`. That's why you should use **Custom Exceptions** to provide meaningful error messages for your users.

Now, run the command again, but provide an existing file, and you'll see an output like this: (See Figure 9)

Figure 9.

```
>
./CLIImage resize

CLIImage Which image do you want to resize?
/Users/wendell_adriel/Documents/Images/Me/2023-newest-lg.jpg

+-----+-----+-----+
| FILENAME | WIDTH | HEIGHT |
+-----+-----+-----+
| 2023-newest-lg.jpg | 4192 px | 4192 px |
+-----+-----+-----+

CLIImage What is the new WIDTH? [Choose a value between 10 and 4192]
```

You can see that now the information about the selected file is displayed in a table, and now we have another question about what's the new width that we want for the image. For our example, the **width** and **height** can't be **less than 10 or greater than the original dimension of the image**. To see our `InvalidDimensionException` in action, type 5 and press enter. You'll see an output like this: (See Figure 10)

Figure 10.

```
>
./CLIImage resize

CLIImage Which image do you want to resize?
/Users/wendell_adriel/Documents/Images/Me/2023-newest-lg.jpg

+-----+-----+-----+
| FILENAME | WIDTH | HEIGHT |
+-----+-----+-----+
| 2023-newest-lg.jpg | 4192 px | 4192 px |
+-----+-----+-----+

CLIImage What is the new WIDTH? [Choose a value between 10 and 4192]
5

An error occurred:
The given width is greater than 4192 px!
```

As you can see, the error message we defined in our **Custom Exception** class is displayed. Now, run the command again and provide the same image, such as 1024, as the new **width** and **height**. You'll see an output like this: (See Figure 11)

Figure 11.

```
>
./CLIImage resize

CLIImage Which image do you want to resize?
/Users/wendell_adriel/Documents/Images/Me/2023-newest-lg.jpg

+-----+-----+-----+
| FILENAME | WIDTH | HEIGHT |
+-----+-----+-----+
| 2023-newest-lg.jpg | 4192 px | 4192 px |
+-----+-----+-----+

CLIImage What is the new WIDTH? [Choose a value between 10 and 4192]
1024

CLIImage What is the new HEIGHT? [Choose a value between 10 and 4192]
1024

SUCCESS! The image was resized successfully! Check the details below:

+-----+-----+-----+
| FILENAME | WIDTH | HEIGHT |
+-----+-----+-----+
| /Users/wendell_adriel/Projects/Personal/CLIImage/output/2023-newest-lg.jpg | 1024 px | 1024 px |
+-----+-----+-----+
```

As you can see, it displays a success message and prints the information about the resized image. If you check the output folder now, you can see that the new file is there.

The Convert Command

Now, let's work on the command that's going to be used for **converting** images. Create a folder named `Convert` inside the `app/Command` folder, and then inside this folder, let's create a class named `DefaultController` since the command is not going to have sub-commands:

```
mkdir app/Command/Convert &&
touch app/Command/Convert/DefaultController.php
```

Now open the newly created `DefaultController` and update it to be like this: (See Listing 12 at end of article)

You can see that this command is simpler than the `resize` one, and we are using the same **helper methods** we created for the `resize` command here as well.

We do have some differences; we are using the `allowedFormats` method from the `ImageFormat` enum that we created to validate if the format given by the user is valid, and we are using a new **Custom Exception** for the validation: `InvalidFormatException`. Let's create this new exception at `app/Exceptions/InvalidFormatException.php` with these contents.

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

namespace App\Exceptions;

use App\Enums\ImageFormat;
use Exception;

final class InvalidFormatException extends Exception
{
    public function __construct()
    {
        parent::__construct(
            'Format must be one of the following: ' .
            implode(', ', ImageFormat::allowedFormats())
        );
    }
}
```

You can see that we are also using the `allowedFormats` from our `ImageFormat` enum here; that's why I said it was going to be a very handy method for us.

Now, with our `convert` command finished, let's test it! Go to the terminal and run:

```
./CLImage convert
```

You'll see an output like this: (See Figure 12)

Figure 12.

```
>
./CLImage convert

CLImage Which image do you want to convert?
```

Type the path for an existing file and press enter. You'll see an output like this: (See Figure 13)

Figure 13.

```
>
./CLImage convert

CLImage Which image do you want to convert?
/Users/wendell_adriel/Documents/Images/Me/2023-newest-lg.jpg

| FILENAME | WIDTH | HEIGHT |
| 2023-newest-lg.jpg | 4192 px | 4192 px |

CLImage What is the new FORMAT? [jpeg, png]
```

You can see that now the information about the selected file is displayed in a table, and now we have another question about what's the new format that we want for the image. For our example, the only allowed formats are **jpeg** and **png**. To see our `InvalidFormatException` in action, type `gif` and press enter. You'll see an output like this: (See Figure 14)

Figure 14.

```
>
./CLImage convert

CLImage Which image do you want to convert?
/Users/wendell_adriel/Documents/Images/Me/2023-newest-lg.jpg

| FILENAME | WIDTH | HEIGHT |
| 2023-newest-lg.jpg | 4192 px | 4192 px |

CLImage What is the new FORMAT? [jpeg, png]
gif

An error occurred:
Invalid format: gif (allowed: jpeg, png)
```

As you can see, the error message we defined in our **Custom Exception** class is displayed. Now run the command again and provide the same image, for example, `png` as the new format. You'll see an output like this: (See Figure 15)

Figure 15.

```
>
./CLImage convert

CLImage Which image do you want to convert?
/Users/wendell_adriel/Documents/Images/Me/2023-newest-lg.jpg

| FILENAME | WIDTH | HEIGHT |
| 2023-newest-lg.jpg | 4192 px | 4192 px |

CLImage What is the new FORMAT? [jpeg, png]
png

SUCCESS The image was converted successfully! Check the details below:

| FILENAME | WIDTH | HEIGHT |
| /Users/wendell_adriel/Projects/Personal/CLImage/output/2023-newest-lg.png | 4192 px | 4192 px |
```

As you can see, it displays a success message and prints the information about the converted image. If you check the output folder now, you can see that the new file is there.

Conclusion

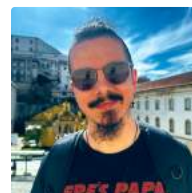
We did it! Our **CLImage** application is done! It's a really simple application, but while building it, we learned how to work with a lot of the features that **MiniCLI** and **Termwind** provide to us, and the **MiniTerm** template is a perfect starting point for that!

I hope that you liked building this application with me! You can find the code for this application in my **GitHub** here: <https://github.com/WendellAdriel/climage>⁹.

Now I have a challenge for you. Fork this repository and **update the styling of it for your liking**. Also, **add a new piece of information** when printing the image information in the terminal: **the filesize**! You can also add support for more formats and more commands to it! The sky is the limit!

Besides our application, we also learned how to create simple CLI scripts with PHP, following the best practices for doing so, and we checked some amazing libraries that can help us build full CLI applications that are beautiful and lightweight with PHP.

I hope that you learned something new from this article, and if you want to chat about it or anything else, you can find me at X: https://x.com/wendell_adriel¹⁰.



Wendell Adriel is a conference speaker, technical writer, open-source enthusiast, amateur photographer and cat lover. Currently works at TrackStreet as a Lead Software Engineer and in his free time, contributes and maintains some open-source projects and writes technical content for his blog. Reach him out on X at [@wendell_adriel](https://x.com/wendell_adriel)

⁹ <https://github.com/WendellAdriel/climage>

¹⁰ https://x.com/wendell_adriel

Listing 9.

```

1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\Services;
6.
7. use App\Enums\ImageFormat;
8. use App\Support\ImageInfo;
9. use Imagick;
10. use ImagickException;
11. use Minicli\App;
12. use Minicli\ServiceInterface;
13.
14. final class ImageService implements ServiceInterface
15. {
16.     /**
17.      * @throws ImagickException
18.      */
19.     public function info(string $imagePath): ImageInfo
20.     {
21.         $image = $this->read($imagePath);
22.
23.         return new ImageInfo(
24.             filename: basename($imagePath),
25.             width: $image->getImageWidth(),
26.             height: $image->getImageHeight(),
27.         );
28.     }
29.
30.     /**
31.      * @throws ImagickException
32.      */
33.     public function resize(
34.         string $imagePath,
35.         int $width,
36.         int $height,
37.         string $outputPath
38.     ): ImageInfo {
39.         $image = $this->read($imagePath);
40.
41.         $image->resizeImage(
42.             columns: $width,
43.             rows: $height,
44.             filter: Imagick::FILTER_UNDEFINED,
45.             blur: 1,
46.             bestfit: true
47.         );
48.
49.         $filename = basename($imagePath);
50.         $output = "{$outputPath}/{$filename}";
51.         $image->writeImage($output);
52.
53.         return new ImageInfo(
54.             filename: realpath($output),
55.             width: $width,
56.             height: $height,
57.         );
58.     }
59.

```

Listing 9 continued.

```

60.     /**
61.      * @throws ImagickException
62.      */
63.     public function convert(
64.         string $imagePath,
65.         ImageFormat $format,
66.         string $outputPath
67.     ): ImageInfo {
68.         $image = $this->read($imagePath);
69.
70.         $image->setImageFormat($format->value);
71.
72.         $filename = pathinfo(
73.             $imagePath,
74.             PATHINFO_FILENAME
75.         ) . "{$format->value}";
76.         $output = "{$outputPath}/{$filename}";
77.         $image->writeImage($output);
78.
79.         return new ImageInfo(
80.             filename: realpath($output),
81.             width: $image->getImageWidth(),
82.             height: $image->getImageHeight(),
83.         );
84.     }
85.
86.     /**
87.      * @throws ImagickException
88.      */
89.     private function read(string $imagePath): Imagick
90.     {
91.         $image = new Imagick();
92.         $image->readImage($imagePath);
93.
94.         return $image;
95.     }
96.
97.     public function load(App $app): void
98.     {
99.         // Nothing to do here
100.     }
101. }

```


Listing 10.

```

1. <?php declare(strict_types=1);
2.
3. final class DefaultController extends BaseController
4. {
5.     /**
6.      * @throws FileNotFoundException
7.      * @throws InvalidDimensionException
8.      */
9.     public function handle(): void
10.    {
11.        $question = 'Which image do you want to resize?';
12.        $askQuestion = $this->buildQuestion($question);
13.        $imagePath = $this->ask($askQuestion);
14.
15.        if ( ! realpath($imagePath)) {
16.            throw new FileNotFoundException();
17.        }
18.
19.        $imageInfo = $this->app->image->info($imagePath);
20.        $this->printImageInfo($imageInfo);
21.
22.        $newW = $this->ask(
23.            $this->buildQuestion(
24.                "What is the new WIDTH? " .
25.                "[10..$imageInfo->width]"
26.            )
27.        );
28.
29.        if ($newW < 10 || $newW > $imageInfo->width) {
30.            throw new InvalidDimensionException(
31.                $imageInfo->width
32.            );
33.        }
34.
35.        $newH = $this->ask(
36.            $this->buildQuestion(
37.                "What is the new HEIGHT?" .
38.                "[10..{$imageInfo->height}]"
39.            )
40.        );
41.
42.        if ($newH < 10 || $newH > $imageInfo->height) {
43.            throw new InvalidDimensionException(
44.                $imageInfo->height
45.            );
46.        }
47.
48.        $result = $this->app->image->resize(
49.            $imagePath,
50.            (int) $newW,
51.            (int) $newH,
52.            $this->app->config->output_path
53.        );
54.
55.        $this->successMessage('Resized successfully!' .
56.            'Check the details below:');
57.        $this->printImageInfo($result);
58.    }
59. }

```

Listing 12.

```

1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\Command\Convert;
6.
7. use App\Command\BaseController;
8. use App\Enums\ImageFormat;
9. use App\Exceptions\FileNotFoundException;
10. use App\Exceptions\InvalidFormatException;
11.
12. final class DefaultController extends BaseController
13. {
14.     /**
15.      * @throws FileNotFoundException
16.      * @throws InvalidFormatException
17.      */
18.     public function handle(): void
19.     {
20.         $imagePath = $this->ask(
21.             $this->buildQuestion(
22.                 'Which image do you want to convert?'
23.             )
24.         );
25.
26.         if ( ! realpath($imagePath)) {
27.             throw new FileNotFoundException();
28.         }
29.
30.         $imageInfo=$this->app->image->info($imagePath);
31.         $this->printImageInfo($imageInfo);
32.
33.         $allowedFormats=ImageFormat::allowedFormats();
34.         $newFormat = $this->ask(
35.             $this->buildQuestion(
36.                 'What is the new FORMAT? ['.
37.                 implode(' ', $allowedFormats).
38.                 ']'
39.             )
40.         );
41.
42.         if ( ! in_array($newFormat, $allowedFormats)) {
43.             throw new InvalidFormatException();
44.         }
45.
46.         $result = $this->app->image->convert(
47.             $imagePath,
48.             ImageFormat::from($newFormat),
49.             $this->app->config->output_path
50.         );
51.
52.         $this->successMessage('Success!' .
53.             'Check the details below:');
54.         $this->printImageInfo($result);
55.     }
56. }

```

Listing 11.

```

1. <?php
2.
3. declare(strict_types=1);
4.
5. namespace App\Command;
6.
7. use App\Support\ImageInfo;
8. use Minicli\Command\CommandController;
9. use Symfony\Component\Console\Output\OutputInterface;
10. use Termwind\Terminal;
11. use Termwind\ValueObjects\Style;
12.
13. abstract class BaseController extends CommandController
14. {
15.     protected function buildQuestion(
16.         string $question
17.     ): string {
18.         return <<<HTML
19.         <div class="py-2">
20.             <div class="px-1 bg-purple-600">CLI Image</div>
21.             <em class="ml-1">
22.                 {$question}
23.             </em>
24.         </div>
25.         HTML;
26.     }
27.
28.     protected function successMessage(
29.         string $message
30.     ): void {
31.         $this->render(<<<HTML
32.         <div class="py-2">
33.             <div class="px-1 bg-green-600">SUCCESS</div>
34.             <em class="ml-1">
35.                 {$message}
36.             </em>
37.         </div>
38.         HTML);
39.     }
40.
41.     protected function printImageInfo(
42.         ImageInfo $imageInfo
43.     ): void {
44.         $this->view('table', [
45.             'headers' => ['FILENAME', 'WIDTH', 'HEIGHT'],
46.             'rows' => [
47.                 [$imageInfo->filename,
48.                 "{$imageInfo->width} px",
49.                 "{$imageInfo->height} px"]
50.             ],
51.         ]);
52.     }
53.

```

Listing 11 continued.

```

54.     protected function render(
55.         string $content,
56.         int $options = OutputInterface::OUTPUT_NORMAL
57.     ): void {
58.         $this->getApp()->termwind
59.             ->render($content, $options);
60.     }
61.
62.     protected function style(
63.         string $name,
64.         Closure $callback = null
65.     ): Style {
66.         return $this->getApp()
67.             ->termwind
68.             ->style($name, $callback);
69.     }
70.
71.     protected function terminal(): Terminal
72.     {
73.         return $this->getApp()->termwind->terminal();
74.     }
75.
76.     protected function ask(
77.         string $question,
78.         iterable $autocomplete = null
79.     ): mixed {
80.         return $this->getApp()
81.             ->termwind
82.             ->ask($question, $autocomplete);
83.     }
84.
85.     protected function view(
86.         string $template,
87.         array $data = []
88.     ): void {
89.         $this->getApp();
90.         ->termwind
91.         ->render(
92.             $app->plates->view($template, $data)
93.         );
94.     }

```



Bounded Fix

Edward Barnard

For the past four years at the USA Clay Target League, we've been working toward updating our PHP software. Domain-Driven Design appeared to be a good direction for us. We're not there yet. But there is one particular pattern that I've been using over and over, which I'll be sharing below.

Fragile Software

Our software is in transition as we evolve to a different framework, different ORM, an API-First architecture, and so on. As with many PHP codebases that have been under development for ten years or more, it's often hard to fix or change something without risk. I could fix a database query, for example, and break something else because that something else relies on that same database query.

To me, this seems obvious – that introducing change carries the risk of also introducing an unexpected side effect (breaking something). We've had a process problem in that we've not sufficiently recognized how fragile our codebase really is. We're getting better at explicitly recognizing this risk with each new proposed change.

I now see more and more tasks phrased like this:

- Make this change with minimum risk of breaking any existing production features or
- Fix this broken Model (database query) without breaking anything else that might also be using that Model/query or
- Introduce this feature without creating a cross-dependency to other existing features.

At the same time, we want the assigned task to take us in the direction we're trying to evolve.

Given the strong desire to not break anything while quickly accomplishing the task at hand, and also given the fact that we have no automated regression tests to help catch breakages, I've been using a "tiny" version of a Bounded Context. Essentially, I'm encapsulating

the new feature/fix/tool inside a new folder, doing my best to avoid any outside dependencies.

Seam Model and Bubble Context

This idea has been around for a while. Michael Feathers, in *Working Effectively With Legacy Code*¹, presents "The Seam Model" (Chapter 4):

A seam is a place where you can alter behavior in your program without editing in that place... When you need to add a feature to a system and it can be formulated completely as new code, write the code in a new method. Call it from the places where the new functionality needs to be. You might not be able to get those call points under test easily, but at the very least, you can write tests for the new code.

Eric Evans, in "Getting Started with DDD When Surrounded by Legacy Systems"², describes the situation with "Strategy 1: Bubble Context":

We say that effectively applying the tactical techniques of DDD requires a clean, bounded context. This can be a daunting requirement when your work is dominated by legacy systems. These systems are often

tangled, and even when they are orderly they are usually not well suited to DDD... this first strategy does not require a big commitment to DDD. It allows even a small team to achieve a modest objective involving some intricate domain logic and, ideally, one with some strategic value. Then at some point the bubble bursts. The carefully designed code is gradually reabsorbed by the legacy. It is no longer a platform for innovative new development.

Evans defines his Bubble Context as:

A "bubble" is a small bounded context established using an Anti-corruption Layer for the purpose of a particular development effort and not intended necessarily to be expanded or used for a long time.

Every answer brings another question! What's an Anticorruption Layer³?

We need to provide a translation between the parts that adhere to different [conceptual] models, so that the models are not corrupted with undigested elements of foreign models. Therefore, create an isolating layer to provide clients with functionality in terms of their own domain model. The layer talks to the other system through its existing interface, requiring little or

¹ Feathers, Michael C. *Working Effectively with Legacy Code*. Robert C. Martin Series. Upper Saddle River, N.J.: Prentice Hall, 2013, pages 31 and 59.

² "DDD Surrounded by Legacy Software." *White paper copyright* 2013. Accessed November 5, 2023. <https://phpa.me/ddd-surrounded>

³ Evans, Eric. *Domain-Driven Design: Tackling Complexity in the Heart of Software*. Boston: Addison-Wesley, 2004, page 365.

no modification to the other system. Internally, the layer translates in both directions as necessary between the two models.

With our legacy codebase, when we disentangle pieces of code, we're essentially placing the extracted piece of code in its own separate world so that it can play by its own rules with its own assumptions. The separation is what Evans calls the Anticorruption Layer. For example, we've also been evolving our database tables. New features reference the new table, while old features reference a MySQL View constructed to look like the old table. That MySQL View is an Anticorruption Layer protecting new-table queries from being polluted by the needs/assumptions of the old-table queries.

Why do we care about these terms and definitions? With any design pattern, I find the theory to be as important as the suggested or sample code. I need to know *why* I am taking this approach so that I can think about how best to design the software I'm writing.

Below, we'll see a simple example of hiding a small feature, fix, or tool inside its own folder. With the new folder (and, therefore, new/separate PHP namespace), we minimize the risk of breaking anything else along the way. We're creating a *vanishingly small* bounded context. I call it a "bounded fix" or "feature in a pocket" to convey the idea that it's purposely small and self-contained.

One-off Project

We've been maintaining hand-edited KML (Keyhole Markup Language) files identifying the latitude/longitude of each of our participating teams. (The name "Keyhole" appears to derive from the original 1970s military reconnaissance satellites that were essentially peeking through keyholes.) I wrote a one-off project to import those files into a new MySQL table. Looking at one of the KML files showed me that this was a straightforward XML import. "Old school" PHP core features would be sufficient. However, I also know that "one-off" projects are sometimes not as "one-off" as planned. There's value in hanging on to the code, either for a second later import, or for answering questions as to precisely how the import was done, or to use as an example for some

other project in the future. I already had a folder `It_Tools/` for this sort of thing, so we'll use that as our starting point.

Page Controller

Listing 1 shows the simple page controller for this one-off tool. The feature itself is the `$controller` object. I call it the "controller" here because it's the thing that orchestrates the feature processing, like the Controller of the Model-View-Controller (MVC) design pattern.

In this case, the feature's name is "Kml", and I instantiate the feature with `KmlFactory::create()`. This is the starting point of my "bounded fix" pattern: for feature XXX, instantiate it with `XXXFactory::create()`. The business process is simple. When I click the "import" button (POST method), run the tool via `$controller->execute()`. Either way, display results on the webpage via `$controller->render()`.

Factory

Listing 2 shows the Factory class responsible for assembling the feature/fix/tool into a class returned by `create()`.

Listing 2.

```
1. <?php declare(strict_types=1);
2.
3. use LegacyBoundedContexts\Infrastructure\WrapDBAL\
4.     Repository\HandCodedWrite;
5. use Service\ViewService;
6. use Subsystems\IT_Tools\Populate\KML_Import\Kml;
7. use Subsystems\IT_Tools\Populate\KML_Import\RKml;
8.
9. class KmlFactory
10. {
11.     public static function create(): Kml
12.     {
13.         $view = (new ViewService())->setView(__DIR__ . '/View');
14.         return new Kml($view, self::repository());
15.     }
16.
17.     private static function repository(): RKml
18.     {
19.         return new RKml(new HandCodedWrite());
20.     }
21. }
```

Listing 1.

```
1. <?php
2.
3. declare(strict_types=1);
4.
5. use Subsystems\IT_Tools\Populate\KML_Import\KmlFactory;
6.
7. $controller = KmlFactory::create();
8.
9. if ($_SERVER['REQUEST_METHOD'] === 'POST') {
10.     $controller->execute();
11. }
12. echo $controller->render();
```

This separate factory class is arguably over-engineering the result. However, I find it works out well as a consistent approach. I've been making quite a number of these bounded fixes, features, or tools, as we disentangle our code. When I see `XXXFactory::create()`, I know what results to expect.

Also, including the Factory from the start allows for modest expansion or extension. I can add to the feature's capabilities "under the covers" without changing how I access the feature in the first place.

We're currently migrating to Twig but not Symfony. Since I want this tool to be self-contained, I want the twig templates

to be in the same folder (and thus separate from everything else). The call `setView(__DIR__ . '/View')` accomplishes this.

Listing 3.

```

1. <?php
2.
3. declare(strict_types=1);
4.
5. use Subsystems\IT_Tools\Populate\KML_Import\RKml;
6. use Twig\Error\LoaderError;
7. use Twig\Error\RuntimeError;
8. use Twig\Error\SyntaxError;
9.
10. class Kml
11. {
12.     private Twig_Environment $view;
13.     private RKml $repository;
14.
15.     public function __construct(
16.         Twig_Environment $view,
17.         RKml $repository
18.     ) {
19.         $this->view = $view;
20.         $this->repository = $repository;
21.     }
22.
23.     /**
24.      * Where to find the KML files to import
25.      *
26.      * @return string
27.      */
28.     public function getDataDir(): string
29.     {
30.         return realpath(__DIR__ . '/Data/Maps');
31.     }
32.
33.     public function listFiles(): array
34.     {
35.         return glob($this->getDataDir() . '/*.kml');
36.     }
37.
38.     public function execute(): void
39.     {
40.         foreach ($this->listFiles() as $file) {
41.             $this->repository->importKml($file);
42.         }
43.     }
44.
45.     public function render(): string
46.     {
47.         $self = $_SERVER['PHP_SELF'];
48.         $files = $this->listFiles();
49.         $dataDir = $this->getDataDir();
50.         $twig = compact('self', 'files', 'dataDir');
51.         try {
52.             return $this->view->render('kml_html.twig', $twig);
53.         } catch (LoaderError|RuntimeError|SyntaxError $e) {
54.             return $e->getMessage();
55.         }
56.     }
57. }
```

Controller/service

Listing 3 shows the main service (which I call `$controller` above). Since this is a one-off project, I simply placed the KML files (to import) inside the same folder structure. The “old school” PHP `glob()` is perfectly adequate for listing the files available to process.

Mysql Insert

Listing 4 shows the MySQL insert statement and Listing 5 shows the table structure. “Insert ignore” makes the import idempotent, meaning I can re-run the file imports as many times as I need to.

Listing 4.

```

1. <?php
2.
3. declare(strict_types=1);
4.
5. interface SQLKml
6. {
7.     public const SQL_INSERT_PLACEMARKS = <<< SIP
8.     insert ignore into `placemarks` (`league_id`, `doc_name`,
9.     `point_name`, `coord_x`, `coord_y`, `coord_z`)
10.    VALUES (?, ?, ?, ?, ?, ?)
11.    SIP;
12.
13. }
```

Listing 5.

```

1. CREATE TABLE `placemarks` (
2.   `id` int unsigned NOT NULL AUTO_INCREMENT,
3.   `league_id` int unsigned NOT NULL,
4.   `doc_name` varchar(255) NOT NULL,
5.   `point_name` varchar(255) NOT NULL,
6.   `coord_x` varchar(255) NOT NULL,
7.   `coord_y` varchar(255) NOT NULL,
8.   `coord_z` varchar(255) NOT NULL,
9.   PRIMARY KEY (`id`),
10.  UNIQUE KEY `doc_name` (`doc_name`,`point_name`),
11.  KEY `league_id` (`league_id`),
12.  CONSTRAINT FOREIGN KEY (`league_id`)
13.    REFERENCES `leagues` (`id`)
14. ) ENGINE=InnoDB;
```

File Import

The repository (Listing 6 on the next page) imports one file at a time. My personal preference is to use hand-coded SQL for situations like this, and since I wrote the code, that’s how it happened! The “old school” `SimpleXMLElement` class works fine for ingesting and traversing the KML file.

Listing 6.

```

1.
2. declare(strict_types=1);
3.
4. namespace Subsystems\IT_Tools\Populate\KML_Import;
5.
6. use LegacyBoundedContexts\Infrastructure\WrapDBAL\
    DomainModel\Interfaces\IHandCodedWrite;
7. use SimpleXMLElement;
8.
9. use function explode;
10. use function file_get_contents;
11. use function preg_replace;
12. use function str_replace;
13. use function trim;
14.
15. class RKml implements SQLKml
16. {
17.     private IHandCodedWrite $write;
18.
19.     public function __construct(IHandCodedWrite $write)
20.     {
21.         $this->write = $write;
22.     }
23.
24.     public function importKml(string $path): void
25.     {
26.         $sql = self::SQL_INSERT_PLACEMARKS;
27.         $path = preg_replace('|^\.+/', '', $path);
28.         $leagueId = (int)str_replace('.kml', '', $path);
29.         $xml = new SimpleXMLElement(file_get_contents($path));
30.         $docName = trim((string)$xml->Document->name);
31.         $placemarks = $xml->Document->Folder->Placemark;
32.         foreach ($placemarks as $placemark) {
33.             $pointName = trim((string)$placemark->name);
34.             $coords = trim($placemark->Point->coordinates);
35.             [$coordX, $coordY, $coordZ] = explode(',', $coords);
36.             $parms = [$leagueId, $docName, $pointName,
37.                 $coordX, $coordY, $coordZ,];
38.             $this->write->updateRow($sql, $parms);
39.         }
40.     }
41. }

```

Summary

I find that, more and more often, I'm creating a small, self-contained fix, feature, or tool. I'm doing this to help ensure that, when introducing new code, I don't risk breaking anything else in the process.

This risk is especially pronounced with any feature that touches the database. That's because our Model classes are far more interconnected (and fragile) than they should be. When fixing a database query, it's far safer to write a completely new query and abandon using the current query.

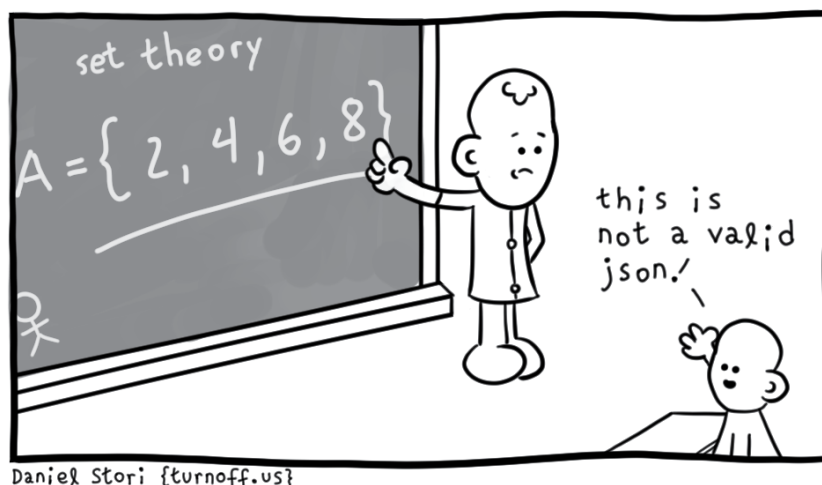
In essence, we're creating transitional code. It's a step in the direction of our target architecture without being the *final* step in that direction. This "temporary" code could remain in place for several years, to be sure, but if we introduced the desired change without breaking anything else, it's a success.

Related Reading

- *DDD Alley: Create Observability, Part 3: Rewrite Business Process* by Edward Barnard, August 2023.
<https://www.phparch.com/article/2023-08-ddd/>
- *DDD Alley: Create Observability, Part 4: Simple Queue System* by Edward Barnard, September 2023.
<https://www.phparch.com/article/2023-09-ddd/>
- *DDD Alley: Create Observability, Part 5: Offline Processes* by Edward Barnard, October 2023.
<https://www.phparch.com/article/2023-10-ddd/>



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.



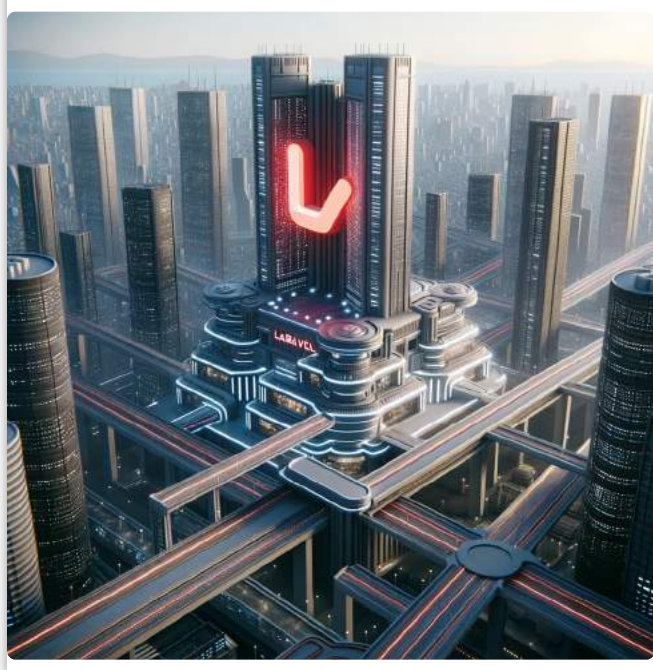


Alternative Architecture in Laravel

Steve McDougall

In Laravel, we are used to writing our code exactly as it comes out of the box. Model View Controller (MVC) pattern is something we are extremely familiar with—and the documentation leads us in this direction by default. However, does this work in every scenario? What happens when features pile up or our team grows to a point that makes it harder to manage and not trip over each other? Some people at this point would look to split the application out into multiple services or micro-services; however, that isn't always very beneficial. Let's take a walk down the path of Software Architecture and see what we can find.

Figure 1.



Many times, as people try and break free from the MVC architecture, they go straight into something like Action Domain Responder (ADR) or Domain Driven Design (DDD)—but this isn't the only approach you can take. Often, borrowing from multiple different architectures can add a real benefit to your application. For example, splitting your application up into Domains and following an additional architecture pattern to ensure that your code is easier to read—and it is also maintainable and testable.

For the sake of anyone who may not know MVC yet, I will do a quick tour of what it is and why it is. However, this article is not designed to teach you what MVC actually is—please look elsewhere for that! MVC is a fantastic pattern to use in any application as you start building. It allows you to separate your code into the behavioral logic and not mix the presentation into the business logic layer. In principle, this sounds

perfect, right? MVC is often a perfect fit for a simple website, blog, or even e-commerce store! You lean on external services for payment, shipping, and some of the more complicated parts that require a dedicated service. Likely, this doesn't sound too strange for a lot of you reading this. However, as your application grows and you start adding more and more features or even integrating with more and more third-party services, managing your application's complexities can suddenly become an uphill battle.

What MVC can give you is a solid foundation to build upon. Some parts of your application may always stay within this architectural approach. However, as your business logic becomes increasingly important, and you aren't just building CRUD layers into your application—Software Architecture quickly becomes your friend.

Let's discuss ADR for a moment, as it is less spoken about in the community. The Action Domain Responder is actually quite a beautiful design pattern to lean on for a growing application. The request comes into your application, which hits a controller. The controller will then call a domain-specific action to perform some sort of logic. Once the logic has been completed, the controller will then ask the responder to respond to the request. This probably sounds somewhat similar to the Action pattern that has been used in the Laravel community for quite some time now. The key difference here is that in ADR, the actions are typically split into domain-specific areas, and a dedicated class is created to handle responding. So think of it a little like how you might use the Action pattern in Laravel but a little more in-depth and advanced.

There are many benefits to this pattern; however, it has its downsides, too. The biggest downside that I have experienced is that code repetition can become a real problem—and it isn't really solving the main problem you fall into as your application scales. This is where I like to look beyond MVC and ADR to see where I can take the application to really give it some benefits. I have flirted with quite a few different approaches in my time, and each approach is only as good as its implementation. However, I am quite a fan of the Vertical Slice Architecture approach.



Some of you, in fact, a lot of you, have likely heard of the clean architecture approach. Well, the Vertical Slice Architecture (VSA) approach is almost the opposite of that. It is closer to a modular design approach where each “slice” is a separation of specific functionality. However, just using this pattern isn’t going to solve all of your problems. You really want to pair this with another pattern so that you can really squeeze the best out of your application and testing approach.

I like to pair something like VSA with DDD. What this allows me to do is have features and domains separated nicely. In fact, designing your domains around features or business functions feels quite natural. If we were to take a standard application and break it into Domain Driven Design for any reason—we would likely split this by feature or business function. This is pretty much what you would do in Vertical Slice Architecture, too. However, the key difference is that each “Slice” in VSA doesn’t really talk to the other features. Each feature is seen as independent, with no reliance or requirement on any other feature or slice.

Let’s take an example of a shipping application. We have main business areas, such as Stock Management, Quality Control, Logistics, Operations, and Finance, to name a few. If we were to design a Shipping Management Application, we would want to make sure that what we built replicated real life as much as possible so that we were, in effect, mirroring the physical world within the Digital world. Stock Management would allow us to handle all of the inventory, re-ordering, and suppliers. Quality Control is a part of the business that makes sure that any inventory leaving the stock area on its way to the logistics area is of the right quality so that it can protect the brand and the buyer. Logistics, obviously, handles shipping everything, as well as receiving any goods coming into the business. Finance deals with payments, accepting payments for the goods that are being sent out, as well as paying any bills that need to go out. Finally, Operations is the part of the business that ensures that the rest of the business is running as it should be.

If we take the above idea and think of it in technical terms, we can split the application into services, domains, and features. We have the Stock management side, which is where someone from the warehouse may look to see what is needed for an order. There would be workflows built so that if stock of a certain product got below a certain level, a new order could be sent out through finance so that logistics could receive the goods and put them back into the warehouse, ready to be ordered by more customers. It is quite a beautiful process when you look at it.

If we were building this in MVC, you can imagine how messy that logic could get very quickly. However, if we split this into slices using Vertical Slice Architecture, then we would have different slices per business area. Then, within each slice, we can split it further into domains or contexts. Leveraging more than one approach can really help you to fine-tune your application here.

There are always situations in software design that mean you must make an exception. What do you do with code that is intended to be shared? Or act as a pathway for one feature to interact with another feature. All features should be able to run independently of each other, but that isn’t how the real world works. Why don’t we walk through a high-level example of how the same code would be structured in each approach, starting with MVC.

Let’s take the example of Logistics accepting a delivery, which will start by accepting the delivery and then passing this over to Stock Management. Stock Management will then break the delivery down into two parts: items to store and items on pre-order. Once items are stored, their stock amount will increase. Any items that are on pre-order will be sent to the Fulfilment team within the Stock Management department. If we approached this in an MVC pattern, it would be quite large. There are a lot of different moving parts, and a lot of this will be distributed across multiple classes. For this example, I will show how it communicates to the external class only. (See Listing 1)

Listing 1.

```
1. final class DeliveryAcceptedController {
2.     public function __construct(
3.         private readonly DeliveryService $deliveryService,
4.     ) {}
5.
6.     public function __invoke(
7.         Request $request, string $delivery
8.     ): JsonResponse {
9.         // We start by validating our delivery
10.        if (! $this->deliveryService->expecting($delivery)) {
11.            throw new UnexpectedDeliveryException(
12.                message: "Unexpected delivery found: {$delivery}",
13.            );
14.        }
15.        // Next we want to ensure that the delivery
16.        // has all the expected parts
17.        if (! $this->deliveryService->validateContents(
18.            $delivery, $request->get('items')
19.        )) {
20.            throw new DeliveryItemsMisalignmentException(
21.                message: "Failed to validate contents of delivery,
22.                    manual check is required.",
23.            );
24.        }
25.        // Next, send this over to the stock management team
26.        dispatch(new DeliveryProcessed($delivery));
27.        // Finally we want to ensure that we notify the
28.        // Logistics team that this delivery has been processed.
29.        return new JsonResponse(
30.            data: [
31.                'message' => 'Delivery accepted.',
32.                'status' => DeliveryStatus::ACCEPTED,
33.            ],
34.            status: Status::ACCEPTED,
35.        );
36.    }
37. }
```



As you can see, this is handled by an API, which is what I would likely do in this scenario. We inject a Delivery Service that contains the main business logic we may require. Then, our Controller only really worries about ensuring that certain preconditions are met before moving on. We want to make sure that the delivery is an expected delivery as well as the contents of the delivery are acceptable. Finally, we dispatch a background job saying that the delivery has been processed, which can emit any events as required and notify all relevant people. Then, let the end user know that the delivery has been accepted. Now, if we were to refactor this into ADR, then we would do something relatively similar—but also ever so slightly different. Let's take a look at an example of this implementation. (See Listing 2)

Listing 2.

```

1. final class DeliveryAcceptedController
2. {
3.     public function __construct(
4.         private readonly DeliveryService $deliveryService,
5.         private readonly DeliveryFailedResponder $failed,
6.         private readonly DeliverySuccessfulResponder $responder,
7.     ) {}
8.
9.     public function __invoke(
10.         Request $request,
11.         string $delivery
12.     ): JsonResponse {
13.         // We want to validate the delivery is expected.
14.         if (! $this->deliveryService->expecting($delivery)) {
15.             return $this->failed->respondWithUnexpectedDelivery(
16.                 $request,
17.                 $delivery,
18.             );
19.         }
20.
21.         // We still want to validate the contents of the delivery
22.         if (! $this->deliveryService->validateContents(
23.             $delivery, $request->get('items')
24.         )) {
25.             return $this->failed->respondWithInvalidContents(
26.                 $request,
27.                 $delivery,
28.             );
29.         }
30.
31.         return $this->responder->deliverySuccessful($delivery);
32.     }
33. }
```

The key difference you see here is that instead of throwing an exception and letting this bubble up to our Exception Handler, we are using a failure responder to handle how we want to respond to a failure. It is not the concern of the controller how the responder may respond other than expecting it to return a JsonResponse that allows it to keep

the return type the same. The final step, when the successful delivery is accepted—all logic to do with dispatching a job, etc. (in other words, all behavioral logic) is then handled by the responder. There are situations where, perhaps, a job is not required to be dispatched that wasn't thought of early on—but the controller does not need to care about this. Its only job is to accept the request, check for any early failure signs, and return a response of some kind.

So, how would this be different if we were using the Vertical Slice Architecture? Honestly, it wouldn't be wildly different in terms of the controller code—that would stay relatively the same. However, the rest of the code being used would be different, as would the structure and namespacing of the code itself.

Vertical Slice Architecture wants you to split your application into as many independent features as possible, with each feature having full control over the controllers, routing, and anything else you might want to add to your application. Let's take a look at an example structure for this. (See Listing 3)

Listing 3.

```

1. └─ app
2.   └─ Deliveries
3.     └─ Commands
4.       └─ Controllers
5.         └─ AcceptNewDeliveryController.php
6.       └─ DeliveryServiceProvider
7.     └─ Events
8.       └─ DeliveryAccepted.php
9.       └─ DeliveryProcessed.php
10.      └─ DeliveryReceived.php
11.    └─ Jobs
12.      └─ DeliveryProcessed.php
13.  └─ Routes
14.    └─ api.php
15.    └─ web.php
16.  └─ Services
17.    └─ IncomingDeliveryService.php
18.  └─ Validators
19.    └─ NewDeliveryValidator.php
20. └─ Finance
21. └─ Logistics
22. └─ StockManagement
```

Here, we start to register all of the possible classes that focus on each “slice” or “domain”—however you may wish to refer to it. This allows each slice to be able to register itself and control how the application loads and sees its available classes and resources. Doing it this way enables you to switch our “slices” as required, as they should be independent of each other—with little to no direct requirement on each other. This approach is very testable and extremely resilient as it doesn't cross what we would call, in DDD terms, the “bounded-context”.

So, a question you may find yourself wondering is, how does communication happen in the scenarios where Deliveries



need to talk to Logistics, etc? This is where we look towards async messaging, such as Apache Kafka or RabbitMQ, not unlike the typical async Jobs we may use in any Laravel application. However, the key difference here is that the message is dispatched to a consumer, who will instantly receive and start handling the command itself. There is no adding to a queue; it is a lot closer to how you might use events and listeners in your Laravel application. Let's have a look at an example of the IncomingDeliveryService class. (See Listing 4)

What we do here is inject the distributed message client and delivery repository into the constructor of this service, which allows us to run specific database queries and send messages to different areas of our application with little effort. Taking this approach allows us to stay contained to our specific "slice" or "domain" while leaning on the dependency injection container to inject specific concrete instances of the dependencies we may have. This separates everything to a level that makes our application easy to use, test, manage, and refactor. From a testing perspective, we can test against specific functionality, outcomes, and side effects in each "slice" without it being tied too closely to the implementation and structure we are currently using.

The biggest downside to this approach is the manual setup, as it makes most of the make generators in Laravel's artisan console redundant. You will get much better results using this approach if you use a dedicated IDE that allows you to create and refactor classes easily. While this may not be the typical approach you see in Laravel development, it does lend itself very well to a scaling application.



Steve McDougall is a conference speaker, technical writer, and YouTube livestreamer. During the day he works on building API tools for Trebble, 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

Listing 4.

```
1. final class IncomingDeliveryService
2.     implements CommunicationService
3. {
4.     public function __construct(
5.         private readonly MessagingClient $message,
6.         private readonly DeliveryRepository $repository,
7.     ) {}
8.
9.     public function expected(string $delivery): bool
10.    {
11.        return $this->repository->expectingDeliveryId(
12.            id: $delivery,
13.        );
14.    }
15.
16.    public function validateContents(
17.        array $items,
18.        string $delivery
19.    ): bool {
20.        $valid = $this->repository->deliveryContainsItems(
21.            items: $items,
22.            id: $delivery,
23.        );
24.
25.        if (! $valid) {
26.            $this->message->distribute(
27.                new InvalidContents($items, $delivery)
28.            );
29.
30.            return false;
31.        }
32.
33.        return true;
34.    }
35. }
```

Beyond Laravel
An Entrepreneur's Guide to
Building Effective Software
by Michael Akopov

Harness the power of the Laravel ecosystem to bring your idea to life.

Written by Laravel and PHP professional Michael Akopov, this book provides a concise guide for taking your software from an idea to a business. Focus on what really matters to make your idea stand out without wasting time on already-solved problems.

Order Your Copy
<https://phpa.me/beyond-laravel>



Object Oriented Visibility

Chris Tankersly

Programming is weird. It is one of the newest disciplines that we have, but at the same time, it has been around since the 1950s. We seem to come up with concepts on an almost daily basis that, when dug into, we discovered years ago. It is almost like the jokes made about modern entertainment—all the good stories have been written, and all we can do is retell them. Sometimes, it feels like in the case of programming, all the good ideas have already been figured out; all we do is pretend we found them as brand new time and time again.

I have seen that, alongside this idea, the idea that the “old” way of doing things gets thrown to the side because, well, old languages did it, which means that today, in 2023, we no longer need those paradigms. Not only do we spend our time rediscovering things, we spend our time revolting against the old ways.

While I am all for coming up with new and better ways of programming, some things go together. In the case of PHP, it is now an object-oriented language. The object model may not be as robust as some other languages, but for the most part, the language is treated as object-oriented and adheres to most of those principles. One of those ideas is the visibility of methods and properties in classes.

In the world of object-oriented programming, visibility, often referred to as ‘access control’, plays a crucial role. It governs the accessibility of class components. By adopting visibility modifiers like `public`, `private`, and `protected`, programmers can structure their code more deliberately and establish boundaries, preventing any unintended or erroneous access.

Visibility Through the Ages

The foundational idea of visibility evolved alongside the maturation of programming paradigms, especially object-oriented programming.

This starts with Simula, developed in the 1960s. Often considered the first object-oriented language, Simula introduced the concepts of ‘classes’ and ‘objects’. While it set the groundwork for encapsulation, Simula did not delve into strict access controls or visibility as we understand it today. It was important as it separated data and behavior into class structures, setting the stage for more stringent access controls in later languages. (See Listing 1)

Then came Smalltalk in the 1970s, developed at Xerox PARC. This language took encapsulation even further. Smalltalk was among the first languages to stress the significance of keeping an object’s data private, making it accessible only through well-defined methods or ‘messages’. While lacking the explicit ‘private’ or ‘public’ labels, Smalltalk’s architecture made developers think about encapsulation and what the objects exposed to the wider program. (See Listing 2)

We eventually got languages like C++ and Java, which brought formal structures to visibility. C++ introduced explicit keywords like `public`, `private`, and `protected`, granting developers more control over class access. Java expanded on

Listing 1.

```
1. BEGIN
2.  CLASS Person(name); NAME name;
3.  BEGIN
4.    OUTTEXT("Name of the person is: ");
5.    OUTNAME(name);
6.    OUTIMAGE;
7.  END;
8.
9.  Person Per1("John");
10. Person Per2("Alice");
11. END;
```

Listing 2.

```
1. Object subclass: Person [
2.   Person class >> new: name [
3.     | instance |
4.     instance := self new.
5.     instance initialize: name.
6.     ^instance
7.   ]
8.
9.   | name |
10.
11.   Person >> initialize: aName [
12.     name := aName.
13.   ]
14.
15.   Person >> printName [
16.     Transcript show: 'Name of the person is: ', name; cr.
17.   ]
18. ]
19.
20. | per1 per2 |
21. per1 := Person new: 'John'.
22. per2 := Person new: 'Alice'.
23.
24. per1 printName.
25. per2 printName.
```




this, integrating its own set of visibility controls and introducing the ‘package-private’ visibility, the default visibility. Such languages positioned visibility as a cornerstone of their design philosophy, fostering safer and more maintainable coding practices. (See Listing 3)

Listing 3.

```
1. #include <iostream>
2. #include <string>
3.
4. class Person {
5.     private:
6.         std::string name;
7.     public:
8.         // Constructor
9.         Person(const std::string& aName) : name(aName) {}
10.
11.        // Method to print the name
12.        void printName() const {
13.            std::cout << "Name is: " << name << std::endl;
14.        }
15. };
16.
17. int main() {
18.     Person per1("John");
19.     Person per2("Alice");
20.     per1.printName();
21.     per2.printName();
22.
23.     return 0;
24. }
```

As the software industry transitioned into the 21st century, modern languages like Python, Ruby, and C# built upon the principles set by their predecessors. For instance, Python’s approach leaned towards convention over strict encapsulation, indicating ‘private’ attributes with a prefix underscore (though it does have a concept it calls “name mangling” for those times class-private items are needed).

It was not only about language syntax or conventions. The idea of visibility became integral to design patterns and architecture. Even in contemporary paradigms like microservices, service boundaries echo the principles of visibility but on a grander architectural scale.

What is Visibility?

For the most part, modern object-oriented languages deal with public, private, and protected visibility. Some languages expand on this, like the remove ‘package-private’ idea in Java or the name mangling in Python. Most languages that support objects usually include these three main access controls. These access controls generally work for both methods and properties of an object. I will reference the combined idea of methods and properties as just “members” of an object.

Public Visibility

When a member is declared as public, it means that it can be accessed from any class or method, irrespective of whether they are inside or outside its parent class. This level of accessibility embodies the idea of being “open to the world”. Such openness has its merits. It is invaluable when creating components meant to be universally available, like API endpoints, utility functions, or core functionalities of libraries and frameworks. With this accessibility comes a responsibility. As these members are exposed to the outside world, they become critical touchpoints of interaction, and any change in their behavior or structure could have far-reaching implications.

Private Visibility

At the other end of the spectrum lies the private modifier. Members declared as private are, in essence, locked within the confines of the class they belong to. No external class can access them, making them the most restrictive in terms of visibility. This kind of tight encapsulation is crucial for maintaining the integrity of a class. By keeping certain mechanisms, helper methods, or specific data storages private, developers ensure that external factors do not accidentally introduce errors. It also gives developers the freedom to make changes to these private members without the fear of disrupting external systems that might rely on them. In many ways, private visibility is about safeguarding and maintaining a controlled environment for the class’s core functionalities.

Protected Visibility

Sitting between public and private is the protected visibility. Members with this designation are accessible within their parent class and also by any subclasses that inherit from the parent. This particular visibility level shines when building extensible systems. For instance, in designing a base class that aims to be extended by various derived classes, protected members serve as controlled touchpoints. They offer a foundation that subclasses can build upon, adapt, or override to suit their specific needs. I tend to default to protected visibility in many of the libraries I build, as I expect people to need to extend my base classes.

How This Works in Php

When you dive into PHP’s object-oriented nuances, it becomes clear that its roots are not novel. PHP’s object model, especially its implementation of visibility, is heavily influenced by Java. Many developers were already familiar with the basic object syntax of languages like C++, and the additional robustness of Java’s system helped influence the early PHP object model.

In PHP, like in Java, visibility is denoted using three primary keywords: public, private, and protected.

- The public keyword, consistent with Java’s behavior, ensures that the member is universally accessible. Such members are interfaces to the external world,



facilitating interactions that are crucial for the class's purpose while shielding its inner mechanics.

- With `private`, PHP mirrors Java's encapsulation principle. Members are tightly bound within the class, ensuring that their access and potential modifications are controlled, thereby preserving the integrity of the class's behavior.
- The `protected` keyword, true to its Java counterpart, offers a nuanced accessibility. These members are available within the class and its subclasses, facilitating inheritance by providing foundational elements that derived classes can expand upon or change.

Drawing inspiration from our previous discussions, let's look at this with the `Person` example, now represented for PHP: (See Listing 4)

Listing 4.

```
1. <?php
2.
3. class Person {
4.     private $name; // Emulating Java's private behavior
5.
6.     // A public constructor, akin to Java's public methods
7.     public function __construct(private $name) {
8.     }
9.
10.    // A public method, reinforcing Java's approach
11.    public function printName() {
12.        echo "Name of the person is: " . $this->name . "\n";
13.    }
14.
15.    // A protected method that can be used in child classes
16.    protected function getFirstName() {
17.        return explode(' ', $this->name);
18.    }
19. }
20.
21. $per1 = new Person("John");
22. $per2 = new Person("Alice");
23.
24. $per1->printName();
25. $per2->printName();
```

In the above structure, PHP's adaptation of Java's visibility principles is evident. By inheriting Java's object model, PHP offers web developers a coherent and potent paradigm. This not only elevates the structural quality of web applications but also ensures that developers transitioning from Java and other object-oriented languages, or at least those familiar with its principles, find PHP's object-oriented programming features intuitive and effective.

Why Visibility Matters

Encapsulation, one of object-oriented programming's cornerstones, thrives on visibility. By bundling data and

methods within objects and setting appropriate visibility, objects can manage their state, ensuring data integrity and avoiding external disruptions.

Moreover, visibility plays a pivotal role in software maintenance and extensibility. By limiting access to class internals, future modifications become safer and more straightforward. Imagine the complications arising from a widely accessed method undergoing change—without visibility restrictions, this can break many dependent components.

Visibility also has a bearing on software security. Restricting access to sensitive components, for instance, is fundamental in applications handling sensitive data. A banking application might want to keep account balance details under wraps, only making deposit or withdrawal methods public.

After understanding the fundamental pillars of visibility in object-oriented programming, a natural progression is to delve into the intricacies of when and how to use these visibility modifiers. Making the right choice can dictate the success of a software project, its maintainability, and its adaptability to changes.

At the heart of every decision about visibility lies a programmer's design philosophy. When determining the visibility of a class member, it is important to think not only about the immediate needs but also about the potential future directions the software might take. For instance, exposing too many components as `public` might seem convenient in the short term, especially for easy access and testing. This can lead to challenges in the future when these components need modifications. External dependencies could break, leading to a ripple effect of issues across the system.

On the other hand, being overly restrictive and keeping too many components `private` can hinder extensibility and potential integrations. The goal is to strike a balance, ensuring that what needs to be kept internal remains so while allowing flexibility where it is needed. Starting with a more restrictive approach and then gradually opening up, based on tangible requirements, often proves to be a prudent strategy.

Another critical consideration, especially when working in collaborative environments, is the importance of clear documentation. A well-documented piece of code, complete with rationale for visibility choices, can be invaluable for team members, both present and future. For instance, when a developer chooses `protected` visibility, indicating its intended use and the reasons behind the decision can guide future developers when they are extending the class or trying to understand its architecture.

It is not just about explaining the choice of visibility but also about conveying the broader context. If a method is made `public` because it is expected to be a key integration point for other systems, this needs to be documented. If a method is `private` due to its handling of sensitive operations, flagging its critical nature in documentation becomes essential.

While most of this discussion centers on object-oriented programming, it is crucial to recognize that software development often involves blending different paradigms. A project



might integrate object-oriented programming with procedural or functional programming elements. In such scenarios, understanding how visibility in object-oriented programming interacts with similar concepts in other paradigms becomes crucial.

In functional programming, while the exact terminologies might differ, there's a concept of limiting the exposure of certain functions or modules. When object-oriented programming elements interact with functional components, developers need to ensure that visibility choices in one paradigm do not inadvertently compromise the intentions of the other.

In essence, the judicious use of visibility modifiers is not only about understanding their definitions. Still, it is an art that encompasses forward-thinking, clear communication, and the ability to adapt to diverse programming landscapes. Making informed decisions in this domain can set the stage for software that's robust, understandable, and ready for the challenges of the future.

Sometimes Things Exist for a Reason

Emerging from historical roots with Simula's rudimentary encapsulation to the more defined paradigms of C++ and Java, the concept of visibility has proven its worth. It is not only a technical construct but a guiding philosophy that should be

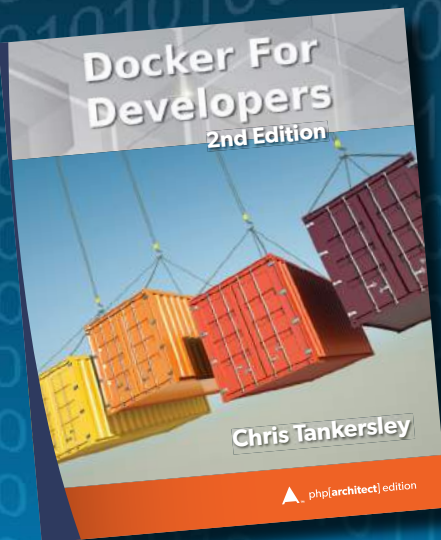
thought about through every line of code, ensuring software's structural integrity, extensibility, and robustness.

A careful balance of public, private, and protected modifiers, complemented by forward-thinking design and comprehensive documentation, is instrumental in creating software that stands the test of time. These decisions impact not only the immediate functionality but also the adaptability and resilience of systems in the face of evolving requirements.

In the vast and intricate landscape of programming, visibility stands as an important paradigm that helps guide developers toward crafting applications that are both powerful and reliable. As technology continues its relentless march forward, the principles of visibility remain timeless, reminding us of the delicate balance between accessibility and protection in our digital creations.



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.



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

This revised and expanded edition includes:

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

Order Your Copy

<http://phpa.me/docker-devs>



@phptek



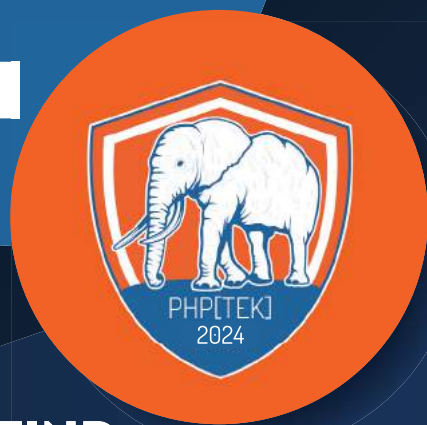
@tek@phparch.social



<https://tek.phparch.com>

PHP[TEK] 2024

**CONNECT WITH THE COMMUNITY,
LEVEL UP YOUR SKILLS, LEARN THE
LATEST TRENDS, AND MAYBE EVEN FIND
YOUR NEXT JOB. RESERVE YOUR SPOT TODAY.**



PHP[TEK] 2024

16TH ANNUAL PHP[TEK] CONFERENCE FOCUSED ON
PHP AND WEB DEVELOPMENT.

[HTTPS://TEK.PHPARCH.COM/](https://tek.phparch.com/)





PHP, Meet Passkeys

Eric Mann

Something you know, something you are, something you have. How does the new technology of passkeys fit into the proven authentication pyramid?

Typical authentication schemes are comprised of three different ways to verify you are who you claim to be:

- **Something you are** - this is typically your identifier. It could be a username, an email, or, in some cases, a unique ID tied to you as an individual.
- **Something you know** - this secret authentication factor helps prove you are who you say you are. It's typically a password or some other piece of secret information known only to you.
- **Something you have** - a device tied to your account that acts as a redundant form of authentication proving your intent to authenticate - even if someone managed to guess your password, they will lack this second physical factor and be unable to authenticate.

Unfortunately, truly secure authentication is hard. Unique identifiers are often public information. Users reuse passwords between various sites, enabling “credential stuffing” attacks¹ that render even secure systems vulnerable to abuse. Multifactor authentication (i.e., device-based second factors or even SMS-based one-time tokens) isn't universal and, even when supported, is often misunderstood by end users.

In 2012, various organizations came together to found the FIDO Alliance² to begin building an open specification for password-free authentication. Today, this organization counts major tech players like 1Password, Amazon, Apple, Google, Microsoft, and Yubico as members. They've worked together on various standards and protocols to make authentication easier for end users and more secure for developers. One of their latest innovations is the passkey.

Introducing Passkeys

Thanks to standard FIDO protocols and support, passkeys³ turn the devices you already know and trust **into** your passwords. Rather than entering a new password, you merely unlock your device using the same PIN or biometric control you already use to unlock other applications.

It's an incredibly smooth user experience and far superior to asking someone to remember a long, complex password for every site they use. Since you're already likely unlocking your phone and laptop with a PIN or biometric, this should

be a very familiar action that doesn't interrupt your use of any website or service.

Thanks to the underlying structure of the standards and protocols involved, it's also orders of magnitude more secure than a typical password workflow, even one protected with multifactor authentication!

How Passkeys Work

Longtime readers of Security Corner will be familiar with my love for efficient, privacy-preserving authentication. In 2018, I covered an approach⁴ to secure remote password authentication leveraging Libsodium. In that scheme, your password was converted to a public/private keypair that was then used to sign a challenge issued by the server in order to authenticate.

Passkeys work similarly, except without the use of a user-defined password.

Instead, a public/private keypair is generated directly on your device. The private key is stored in secure hardware, while the public key is passed to the server during device enrollment. Future authentication is performed by signing a challenge from the server with your private key (again, within secure hardware)—the signature is validated by the server using your public key. (See Figure 1 on the next page)

Everything about these protocols is geared around privacy. There's no password to exchange, just a public key and a signature, both tied to some random key created on the device. There's also no possibility of **leaking** the private key since it lives within the secure hardware embedded in the device.

The FIDO protocol also supports a different set of keys per site. Nothing is reused. Nothing can be phished. Nothing can fall victim to credential stuffing. You're relying on secure hardware and strong cryptographic protocols every step of the way!



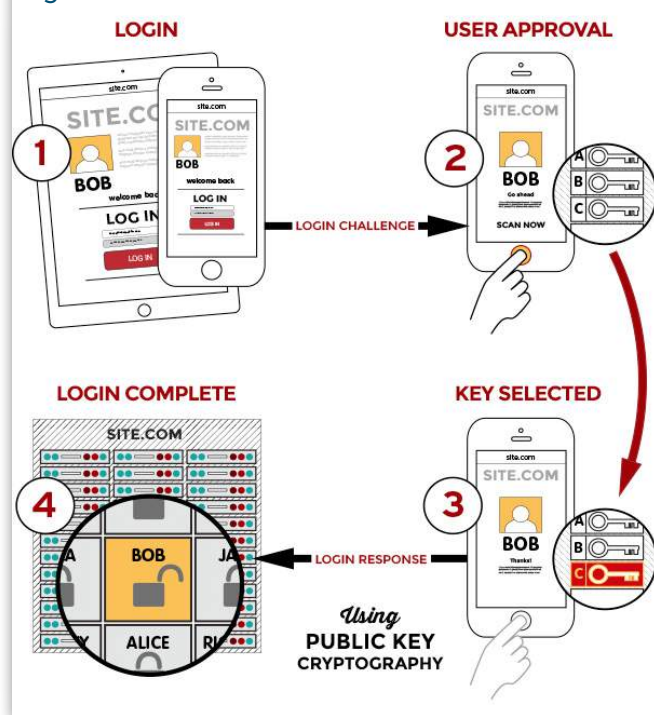
¹ <https://phpa.me/23andme-attack>

² <https://fidoalliance.org>

³ <https://fidoalliance.org/passkeys/>

⁴ <https://phpa.me/security-july-2018>

Figure 1.



Where to Use Them

Websites and applications are opening the door to passkeys daily. There's a high likelihood that your existing devices are compatible and you're ready to get started.

If you use an iOS device, an Android device, or a modern Mac or Windows computer, you likely have the requisite hardware and software already. ChromeOS supports passkeys in some situations, and even Ubuntu supports them when using Chrome or Edge browsers.

Google, PayPal, Instacard, Tailscale—the list of sites and services that support passkeys⁵ is frequently growing. There are also plenty of FIDO-compliant and webauthN⁶ libraries available that empower you to support passkeys in your own PHP application.

User security is hard. Passwords are hard. Multifactor authentication is hard. None of these statements need to remain true—we have newer, better alternatives that we can and should leverage to make the next generation of PHP applications easier and more efficient for our customers.



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

⁵ <https://www.passkeys.io/who-supports-passkeys>

⁶ <https://phpa.me/webauthn-framework>

Secure your applications against vulnerabilities exploited by attackers.

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

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





php[architect]

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

SUBSCRIBE

www.youtube.com/@Phparch



WebAuthn: The Future to Securing Applications

Matt Lantz

Online security is a predominant concern among most companies and developers. Traditional authentication methods, such as passwords, have proven to be increasingly vulnerable to hacking and phishing attacks.

Corporations' yearly five-minute videos that they force employees to watch do not address the root problem. To address this growing concern, a new standard called WebAuthn has emerged as a groundbreaking solution that promises to revolutionize online authentication. Furthermore, the WebAuthn Framework offers an elegant means of using this new technology with your standard PHP applications.

First, what is WebAuthn?

WebAuthn, short for Web Authentication, is a web standard developed by the World Wide Web Consortium (W3C) and the FIDO Alliance. Its primary goal is to provide a secure and easy-to-use framework for authenticating users on the web using public-key cryptography. This standard enables web applications

to utilize hardware-based authenticators, such as biometric sensors, USB tokens, or mobile devices, to verify a user's identity without requiring passwords, fundamentally password-less authentication.

It works by employing a two-factor authentication approach that combines "something you have" (the authenticator) and "something you know" (a user gesture like a PIN or fingerprint) to establish a user's identity. The process involves the following steps:

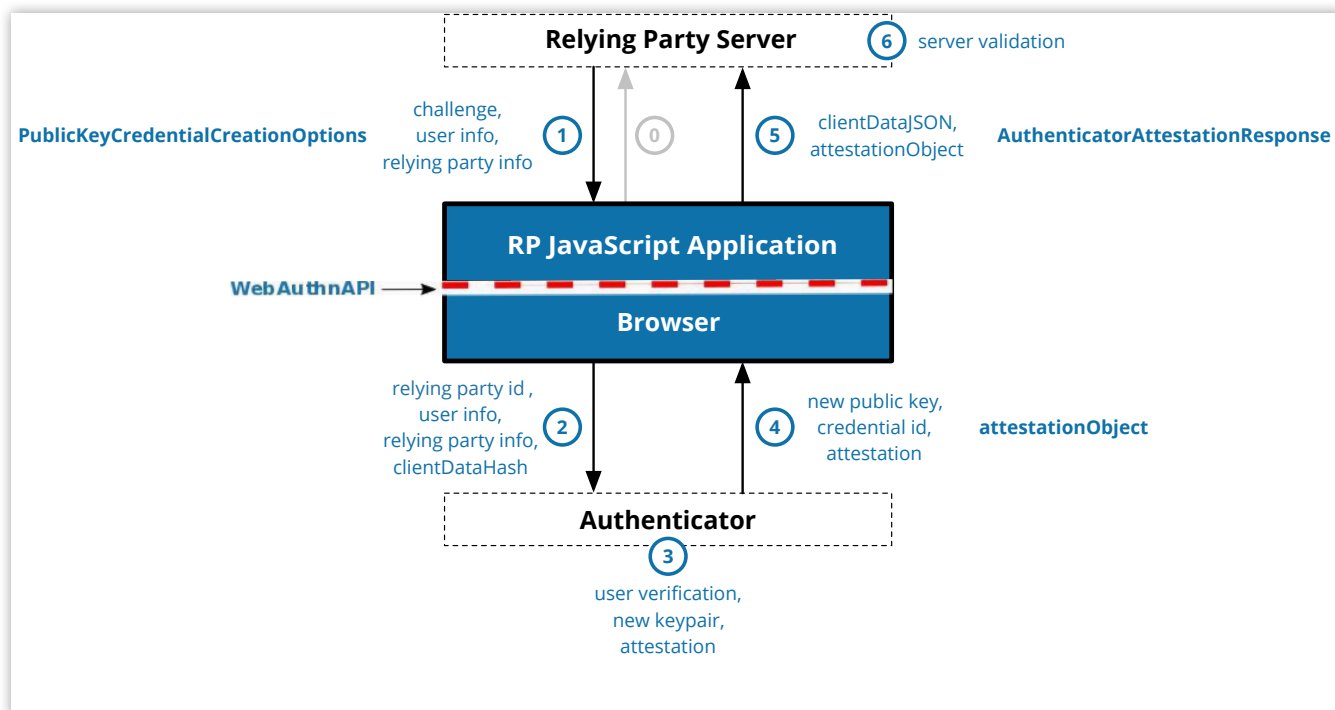
Registration: When users attempt to authenticate on a website supporting WebAuthn, they are prompted to register their authenticator device by binding it to their account. **Authentication:** During subsequent logins, the

website challenges the user's authenticator to prove their identity.

It sounds pretty simple upfront, but there are many layers beneath this where the framework's value comes in. Nearly all browsers support WebAuthn, and adaptor packages for Symfony and Laravel are available. One important thing to remember before you get too excited about this authentication system is that it is JavaScript API-driven, so beyond what packages cover, you may need to custom-write some JavaScript to fill in gaps and add details.

So, what are the overall benefits of implementing WebAuthn?

We have to look at WebAuthn as a way of making your application passwordless, meaning you would no longer be susceptible to a wide range of attacks





that OWASP outlines. It removes the threat of users using old, vulnerable passwords or requiring users to change their passwords consistently.

It enhances user experience as your application can utilize biometrics such as a finger tap, a photo, or a phone notification. There is no need to remember a password; furthermore, you don't even need a password manager tool to handle "auto-generating" a complex password. There is also a focus on privacy within WebAuthn since biometric data is processed locally and never leaves the user's device.

Phishing attacks also become stale since WebAuthn requires the device to be in the hands of the user at the time of its processing. Attempts at using fake websites to steal user credentials become useless.

As I've mentioned, WebAuthn is supported by all major browsers and has been adopted by the technology giants Apple, Google, and Microsoft. This leaves us with the question many of you started reading this article for:

How do I implement it?

If you're on a larger team and really need to build your integration from scratch, go for it. Most auth systems are best when the open-source community can help ensure they are stable and up to date with the best security practices and big fixes. Implementing the WebAuthn framework from scratch requires the following components in your PHP codebase.

- An Attestation Statement Support Manager
- At least one Attestation Statement Support object
- An Attestation Object Loader
- A Public Key Credential Loader
- An Extension Output Checker Handler
- An Algorithm Manager
- An Authenticator Attestation Response Validator
- An Authenticator Assertion Response Validator

Many of these components are between 5 to 10 lines of code - thanks to the WebAuthn framework - but each comes with various details that are impacted by whichever PHP framework you use. Beyond the above, you still need to implement the registration of users, storage of public key information, and, lastly, configure the actual authentication measures within your application, which may include writing middleware or firewalls. Furthermore, registration and subsequent login pages must contain some HTML and JavaScript to collect the appropriate WebAuthn credentials. The WebAuthn-framework recommends using this package to handle the JavaScript side of events:

<https://phpa.me/webauthn-browser>¹.

This JavaScript package predominantly handles browser permissions and integrates the various AJAX requests that perform the above validations. Overall, I do not recommend building an implementation of this from scratch. When reading through the documentation surrounding the framework, I found numerous "gotchas" outlined in their

documentation. I recommend using open-source packages for whichever PHP framework you are currently using. Below, I will outline how to implement a Laravel-based package I encountered in my research. I chose this one because it's on version 4.x and has over 400,000 downloads.

The webauthn-framework comes with a Symfony bundle baked in, but for Laravel developers, I currently recommend using `asbiin/laravel-webauthn`, which handles integrating the webauthn core elements into your application, including the bulk of the JavaScript coding required.

Installation as simple as running: `composer require asbiin/laravel-webauthn guzzlehttp/psr7`

Then you need to ensure that you publish the configuration as well by running `php artisan vendor:publish --provider="LaravelWebauthn\WebauthnServiceProvider"`

Lastly, as usual, run a migration with `php artisan migrate`.

Assuming you're working with a standard Laravel application, add the following to the `$routeMiddleware` array of your `app/Http/Kernel.php` file.

```
use LaravelWebauthn\Http\Middleware\WebauthnMiddleware;
...
'webauthn' => WebauthnMiddleware::class,
```

Within your routes, you can then add it as a middleware:

```
Route::middleware(['auth', 'webauthn'])
    ->group(function () {
...
});
```

Lastly, to enable the passwordless authentication, you need to set your driver in your `config/auth.php`

```
'providers' => [
    'users' => [
        'driver' => 'webauthn',
        'model' => App\Models\User::class,
    ],
],
```

This package is intended to be used with standard responses and page loads, so if you're using an SPA approach, you need to disable the views, which you can do in the package configuration:

```
'views' => false,
```

If you're currently using Inertia, you can see an example application using this package, which the maintainer Alexis also made available here:

<https://phpa.me/laravel-webauthn-example>²

You can easily customize the bulk of the view files in this package to suit your application requirements. Further, there are customizable options similar to Laravel Fortify, which can help developers customize the Authentication Pipeline process.

¹ <https://phpa.me/webauthn-browser>

² <https://phpa.me/laravel-webauthn-example>



To enable further layers, you can set the method `Webauthn::authenticateThrough` in your boot method in the `FortifyServiceProvider` class as shown in Listing 1.

Listing 1.

```
1. Webauthn::authenticateThrough(
2.     function (Request $request) {
3.         return array_filter([
4.             config('webauthn.limiters.login')!= null ?
5.                 null :
6.                 EnsureLoginIsNotThrottled::class,
7.                 AttemptToAuthenticate::class,
8.                 PrepareAuthenticatedSession::class,
9.         ]);
10.     }
11. );
```

There are other options you can configure, as well as the rate-limiting, which uses email and IP by default. There is also a set of events you can tie into based on your application's needs.

On login with Webauthn check.

`\LaravelWebauthn\Events\WebauthnLogin`

On preparing authentication data challenge.

`\LaravelWebauthn\Events\WebauthnLoginData`

On a failed login check.

`\Illuminate\Auth\Events\Failed`

On registering a new key.

`\LaravelWebauthn\Events\WebauthnRegister`

On preparing register data challenge.

`\LaravelWebauthn\Events\WebauthnRegisterData`

On failing registering a new key.

`\LaravelWebauthn\Events\WebauthnRegisterFailed`

If you wish to add further details or customize the payload in the various responses from the WebAuthn sequences, you can bind it using the register method of your `AppServiceProvider`.

```
use LaravelWebauthn\Services\Webauthn;
```

```
class AppServiceProvider extends ServiceProvider
{
    public function register()
    {
        Webauthn::loginViewResponseUsing(
            LoginViewResponse::class
        );
    }
}
```

Then your `LoginViewResponse` can look something like the following:

```
class LoginViewResponse extends LoginViewResponseBase
{
    public function toResponse($request)
    {
        $publicKey = $this->publicKeyRequest($request);

        return Inertia::render('Webauthn/WebauthnLogin', [
            'publicKey' => $publicKey
        ]);
    }
}
```

Overall, the following responses can be overridden so long as they align with the corresponding contract:

Response	Contract
<code>loginViewResponseUsing</code>	<code>LoginViewResponseContract</code>
<code>loginSuccessResponseUsing</code>	<code>LoginSuccessResponseContract</code>
<code>registerViewResponseUsing</code>	<code>RegisterViewResponseContract</code>
<code>registerSuccessResponseUsing</code>	<code>RegisterSuccessResponseContract</code>
<code>destroyViewResponseUsing</code>	<code>DestroyResponseContract</code>
<code>updateViewResponseUsing</code>	<code>UpdateResponseContract</code>

Lastly, to ensure that you do all this work successfully, you must have a secure domain set up locally.

- proper domain (localhost and 127.0.0.1 will be rejected by `webauthn.js`)
- SSL/TLS certificate trusted by your browser
- HTTPS port 443 (ports other than 443 will be rejected)

WebAuthn is a game-changer in online authentication, offering a more secure, user-friendly, and interoperable approach to identity verification. WebAuthn has the power to mitigate the risks of credential theft and phishing attacks, which means it will only become more popular. As the world becomes increasingly dependent on digital interactions, adopting WebAuthn will be crucial to ensuring trust and confidence in online platforms.



Matt has been developing software for over 13 years. He started his career as a PHP developer for a small marketing firm but has since worked with a few Fortune 500 companies, led a couple teams of developers, and is currently a Cloud Architect for a significant Travel technology company. He's contributed to the open-source community on projects such as Cordova and Laravel. He also made numerous packages and has helped maintain a few. He spends time with his wife and kids when he's not tinkering with code or learning new technologies.

[@Mattylantz](#)



Done For You

Maxwell Ivey

This time I want to talk about an approach to improving accessibility that would probably be controversial to long-time advocates in the world of accessibility and inclusion. It is the idea that ‘done-for-you’ has a real place in accessibility.

To me, the goal should be to build a website or app in such a way that I can easily accomplish the goals the site was developed for in the first place. I want to be able to find my way around and take positive actions with as little thought and as much joy as possible.

Given the limitations of the available adaptive technology, the variety of disabilities, and the varying levels of proficiency with technology among users, sometimes ‘done-for-you’ could be the easiest way to make my life better.

This is especially true in the process of updating a system to include accessibility.

In fact, several of the companies and organizations I have worked with over the years required us to take this approach.

They had limited time and other resources.

With Nepris now called Pathful, an online education platform, I wouldn’t have even been able to create my profile when I first discovered it in 2018. In the beginning, they created my profile, set up subject matter notifications, accepted available presenter slots, and even helped me get logged into their proprietary meeting platform. Without their help, I would have missed the opportunity to share my knowledge and experience with thousands of young people across the country.

But even now, they still do a lot for me. Not because I need them to, but because it makes my life easier to allow them to.

Their site is now highly accessible.

A member of their team regularly monitors new classroom requests and emails me when they find one they think I would be a good fit for. That means I don’t have to sift through

the new listings on their site or work through lots of email notifications.

And because we built a relationship, my contacts there have become friends who support me in my activities.

With Pod Match, a site that matches up podcast hosts and interested guests, I would have missed many opportunities to find perfect guests for my podcast or to be interviewed by awesome hosts.

Those interviews allowed me to share my message, spotlight incredibly inspiring people, and build lasting relationships with fellow podcasters.

They had to create my original profile. They had to help me figure out how to navigate to their advanced search engine.

Recently, one of the businesses I am working with decided to take their team communications to Coda. When I visited their intake form for the group, I was only able to navigate half of the fields. Instead of getting upset, I reached out to them.

When they told me they would work on the accessibility issues, I approached the issue with my usual problem-solving mindset. I asked them if they would be willing to fill out my form for me if I provided the answers to the questions. I could read the field labels with my screen reader, but I couldn’t enter the required information in all of them.

They wrote back saying that they were willing. It turned out that I forgot a few of the fields, and their customer rep prompted me to provide the answers I had left out.

As a result, I now have a team profile there, and we are having conversations about accessibility.

There are a couple of examples where businesses use the ‘done-for-you’ approach on a larger formal scale.

For example, there is the accessibility-focused customer support team at Apple. They provide a different contact number where you reach a team of people who have been specifically trained to answer questions about Apple technology being used with adaptive technology options like their screen reader voice-over.

Another one is my hosting provider, HostGator. Their site is mostly accessible, but there are things that it is just easier to let a trained, experienced support person do for you.

I recently filed for a new domain name for my new podcast, The Accessibility Advantage, and as the home of my accessibility consulting website.

I called them up and asked them to help me activate the site, redirect the DNS name servers, turn on the security setting that makes a site HTTPS, and install WordPress for me. They have a dedicated 800 number for people with disabilities to call with problems. Their site is highly accessible, but it’s a huge site. And some of the functions you want to perform have dangerous possible effects if you make a mistake.

So, I am very happy to let them do it for me.

This whole attitude goes back to my early childhood education about my expected vision loss from RP, retinitis pigmentosa. My dad told me repeatedly never to be afraid to ask for help. As a result, I am not as fiercely independent as the instructors of people with disabilities encourage us to be. I am also more willing to work with people and accept their help than many of my disabled friends.

I believe in interdependence alongside or over independence. I didn’t even know the word inter-dependence, but I



knew that difficult, challenging things are not meant to be done alone.

Now, how do you take advantage of this?

First, by evaluating what the most important things are about someone using your website or app. For example, the most important part of an airline booking site is being able to choose your ticket and purchase it. Then, based on the reason behind the content in the first place, try to see whether you can design a navigation option that would allow a disabled person to operate your website or app successfully.

Then, consider whether or not it might make your life and the lives of your disabled users easier by offering a 'done-for-you' option. This could be for highly complicated operations or for functions that involve manipulating images or videos.

Once you identify these functions where 'done-for-you' is the best option for everyone, then create a notification to let the disabled person know how to proceed. This is usually done by having a text message, a link, or both saying if you have a disability, then please contact us here.

Naturally, you have to create the system to support that option. You have to make sure those contacts go to an individual or a team capable of handling the requests and prepared to receive them. And you need a way to track those interactions to make sure the proper department is handling them promptly, efficiently, and personably. It would help if you realized that not everyone is going to accept 'done-for-you' as an accessibility answer. Many are too caught up in doing things by the book, such as the WCAG or the ADA, and they can't accept answers based on a pragmatic problem-solving approach.



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

The advertisement features a background image of a laptop with code on the screen. Overlaid on this is a large orange shape containing several hexagonal callouts. The central text reads 'php[architect] [consulting]'. The callouts list services: 'Get customized solutions for your business needs', 'Leverage the expertise of experienced PHP developers', 'Create a dedicated team or augment your existing team', 'Improve the performance and scalability of your web applications', and 'Building cutting-edge solutions using today's development patterns and best practices'. The php[architect] logo (an elephant) is in the top right, and the email 'consulting@phparch.com' is at the bottom right.

**php[architect]
[consulting]**

Get customized solutions for your business needs

Leverage the expertise of experienced PHP developers

Create a dedicated team or augment your existing team

Improve the performance and scalability of your web applications

Building cutting-edge solutions using today's development patterns and best practices

consulting@phparch.com

REAL SOLUTIONS *for* REAL NETWORKS

ADMIN is your source for technical solutions to real-world problems.

Improve your admin skills with practical articles on:

- Security
- Cloud computing
- DevOps
- HPC
- Storage and more!

GET IT FAST
with a digital subscription!



@adminmagazine



@adminmag



ADMIN magazine



@adminmagazine

6 issues per year!

ORDER NOW

shop.linuxnewmedia.com





Shellsort

Oscar Merida

Much like Comb Sort improved on the Bubble Sort, Shellsort also improves the execution of the Insertion Sort. Again, the improvements come from comparing items that are far apart in the array in early iterations until we are comparing adjacent items again.

Recap

For next month, write an implementation of the Shellsort, a variant of the insertion sort. Generate an array of N random numbers. Pick as large a range as you want to work with, but don't make the range too small, and use your comb sort function to order it.

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

Algorithm

Let's recall that the insertion sort works. On any pass, k , we look at the items from the beginning of the array through the k th element. For the element at spot k , we find where in that part of the array it belongs and insert it into the right spot. Thus, after a pass, the elements in spots 0 through k are properly sorted. Each subsequent pass is trying to figure out where the next element, $k+1$, belongs in that "trailing" sorted array.

This inspection requires many comparisons between adjacent elements. Like the bubble sort, it means moving items that should be at the very front or very end of the array to their correct spot requires multiple calls to our `swap_elements()` function. How can we get items near their final spot quicker?

The comb sort improved performance by comparing more distant elements in our arrays. Shellsort takes a similar approach by comparing sets of elements that are some "gap" distance apart.

Again, turn to the Wikipedia article for shellsort¹ for an explanation of the pseudocode to implement it. If, like me, you're a visual learner, the animations on Wikipedia are invaluable for grokking these sorts. (See Listing 1)

We'll come back to what the sequence of gaps might be. The original algorithm proposed by Shell used a gap that decayed exponentially. The first time, the gap was $n/2$, where n is the number of array elements. Each subsequent gap is one-half of the previous one until the gap is 1. At that gap, the shell sort is identical to the insertion sort.

Listing 1.

```
1. # Start with the largest gap and work down to a gap of
2. # 1 similar to insertion sort but instead of 1, gap is
3. # being used in each step
4. foreach (gap in gaps)
5. {
6.     # Do a gapped insertion sort for every elements in
7.     # gaps. Each loop leaves a[0..gap-1] in gapped order
8.     for (i = gap; i < n; i += 1)
9.     {
10.        # save a[i] in temp and make a hole at position i
11.        temp = a[i]
12.        # shift earlier gap-sorted elements up until the
13.        # correct location for a[i] is found
14.        for (j=i; (j>=gap) && (a[j - gap]>temp); j -= gap)
15.        {
16.            a[j] = a[j - gap]
17.        }
18.        # put temp (original a[i]) in its correct location
19.        a[j] = temp
20.    }
21. }
```

Solution

Listing 2 (on the next page) shows an implementation of the Shellsort using that gap sequence.

To understand what's happening, let's look at the first pass. We have the following array with eight elements. Our first gap size is 4.

```
9 15 24 19 2 1 3 21
```

That means we create set of all elements 4 spaces apart:

```
9 15 24 19 2 1 3 21
9 ----- 2
15 ----- 1
24 ----- 3
19 ----- 21
```

For each pair, in this case, we use an insertion sort to order them. In this case, we're swapping only two elements. We make the following swaps. After that pass, our array is sorted as shown at the end. Our smaller numbers are much closer to the start of the array, and the larger numbers are near the end.

¹ <https://en.wikipedia.org/wiki/Shellsort>



```

2 ----- 9
 1 ----- 15
   3 ----- 24
     19 ----- 21
2 1 3 19 9 15 24 21

```

On the next pass, our gap is 2 – one-half of 4, and the sets we sort are:

```

2 1 3 19 9 15 24 21
2 - 3 -- 9 -- 24
 1 - 19 - 15 -- 21

```

Each set gets sorted and combined:

```

2 - 3 -- 9 -- 24
 1 - 15 - 19 -- 21
2 1 3 15 9 19 24 21

```

Finally, the next gap is 1 and our final gap. This sorts the array using a regular insertion sort but with more elements close to where they should be.

```

2 1 3 15 9 19 24 21  Start
1 2 3 15 9 19 23 21  Sort 0...1 requires 1 insertion
1 2 3 15 9 19 23 21  Sort 0...2, already sorted
...                  Sort 0...3, 0...4 already sorted too
1 2 3 9 15 19 23 21  Sort 0...5, requires 1 insertion
1 2 3 9 15 19 21 23  Sort 0...6, 0...7 already sorted
1 2 3 9 15 19 21 23  Sort 0...8 requires 1 insertion

```

Benchmarks for Exponential Gaps

Shellsort 1	1000	5000	10000
Fastest	0.02035	0.1704	0.3968
Slowest	0.03222	0.2394	0.9712
Mean	0.02462	0.2003	0.6082

Listing 2.

```

1. function shell_sort_ciura(array &$list): void
2. {
3.     $data = [1, 4, 10, 23, 57, 132, 301, 701];
4.     $gaps = array_reverse($data);
5.
6.     foreach ($gaps as $gap) {
7.         if ($gap > count($list)) continue;
8.
9.         $i = $gap;
10.        while ($i < count($list)) {
11.            for ($j = $i;
12.                ($j >= $gap && $list[$j-$gap] > $list[$j]);
13.                $j -= $gap
14.            ) {
15.                swap_elements($list, $j, $j - $gap);
16.            }
17.            ++$i;
18.        }
19.    }
20. }

```

Ciura's Gap Sequence

The gap sequence we chose to use can greatly affect the performance of the Shellsort. Too many gaps produce overhead, while too few gaps make subset insertion sort slower. The Wikipedia article lists a baker's dozen of proposed gap sequences and their theoretical performance. Most, like the exponential sequence, are based on the size of the array. Ciura's sequence is experimentally derived and minimizes the average number of comparisons. The sequence is surprisingly short, though the Wikipedia article gives a formula for calculating larger sequences.

1, 4, 10, 23, 57, 132, 301, 701

Listing 3 (on the next page) shows a Shellsort that uses Ciura's sequence for gaps.

Ciura Benchmarks

Shellsort C	1000	5000	10000
Fastest	0.01789	0.1333	0.3559
Slowest	0.03301	0.2012	0.5008
Mean	0.02097	0.1594	0.4106

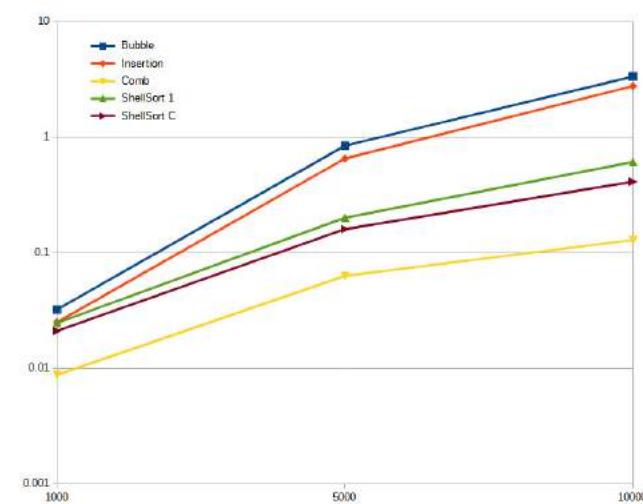
Comparison

The table below shows how the sorting algorithms we've looked at compare. Ciura's sequence of gaps is faster than the exponential gaps from our first implementation. Its mean performance does not suffer as the array size gets larger compared to the first. Surprisingly, the Comb sort is still king of the hill. Figure 1 on the next page shows that the Shellsort performance is in the middle of the pack.

Sort	1000	5000	10000
Bubble	0.0321	0.8414	3.3420
Insertion	0.0249	0.6525	2.7604
ShellSort 1	0.0246	0.2003	0.6082
ShellSort C	0.0210	0.1594	0.4106
Comb	0.00873	0.06275	0.1286



Figure 1.



Quicksort

For next month, generate an array of N random numbers and use Quicksort to order them from smallest to largest. 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.

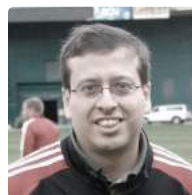
Some Guidelines And Tips

- 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.

Listing 3.

```

1. <?php
2.
3. function swap_elems(array &$list, int $i, int $j): void
4. {
5.     // safety check
6.     if (array_key_exists($i, $list) &&
7.         array_key_exists($j, $list)
8.     ) {
9.         $tmp = $list[$i];
10.        $list[$i] = $list[$j];
11.        $list[$j] = $tmp;
12.        return;
13.    }
14.
15.    throw new \InvalidArgumentException(
16.        "Invalid index specified"
17.    );
18. }
19.
20. /**
21.  * We're assuming sequential integer keys
22.  * @param array<int, scalar> $list
23.  */
24. function shell_sort(array &$list): void
25. {
26.     $gap = floor(count($list) / 2);
27.
28.     while ($gap > 0) {
29.         $i = $gap;
30.         while ($i < count($list)) {
31.             for (
32.                 $j = $i;
33.                 ($j >= $gap && $list[$j - $gap] > $list[$j]);
34.                 $j -= $gap
35.             ) {
36.                 swap_elems($list, $j, $j - $gap);
37.             }
38.             ++$i;
39.         }
40.         $gap = floor($gap / 2);
41.     }
42. }
```



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>

Is Your Code Abstracted Enough To Minimize Load?

Christopher Miller

Last month, we looked at the handling of plans—well, building from that, we’re going to start delving into abstraction and minimizing load. What do I mean, though, by abstraction?

Abstraction is a fundamental concept in programming that helps us manage complexity. At its core, abstraction means hiding the unnecessary details and showing only the essential parts of something. Let’s break it down:

Imagine you’re driving a car. When you press the accelerator, the car moves forward, right? You don’t need to know how the engine, transmission, or fuel injection system work under the hood to make the car go. That’s abstraction in action. You interact with a simple interface (the pedal), and the complex mechanisms are hidden from you.

In code, abstraction means creating a simplified, high-level representation of a more complex system. You expose only the necessary functionalities while hiding the implementation details. For example, in object-oriented programming (OOP), you use classes to define objects. The class defines the structure and behavior of an object, and you interact with it through methods and properties.

Let’s say you’re building a banking application. You create a `BankAccount` class. Users of your code don’t need to know how transactions are stored in the database or how interest calculations are done. They interact with the `BankAccount` object through methods like `deposit`, `withdraw`, and `getBalance`. The underlying complexities are abstracted away, making it easier for both developers using your code and users of the banking application.

Abstraction not only simplifies the interaction but also allows for flexibility and easier maintenance. If you ever need to change how interest is calculated, you can do it inside the `BankAccount` class without affecting the rest of your application. This is a key principle of object-oriented programming and a great way to manage complexity in software engineering.

What Load are We Minimizing?

Here, we are specifically talking about the cognitive load of reading and understanding the code.

In the context of a developer reading code, cognitive load refers to the mental effort and capacity required to understand, analyze, and work with a piece of code. It’s a critical aspect of software development, and reducing cognitive load is essential for writing clean, maintainable code. Let’s break down cognitive load in this context:

- **Code Complexity:** The complexity of the code, such as nested loops, conditional statements, and deep function calls, can increase cognitive load. Simple, well-structured code is easier to understand.
- **Inconsistent Style:** If code lacks a consistent style and follows no established conventions, it can be mentally taxing to decipher. Consistent naming, formatting, and organization help reduce cognitive load.
- **Comments and Documentation:** Well-documented code reduces cognitive load significantly. Comments explain the purpose of functions, classes, or sections of code, making it easier for a developer to understand the intent.
- **Modularity:** Breaking code into smaller, reusable modules and adhering to the principles of OOP can make it more understandable. A well-designed class or function should have a single responsibility, reducing cognitive load.
- **Variable and Function Names:** Descriptive and meaningful variable and function names can lower cognitive load. Developers should be able to infer the purpose of a variable or function just by looking at its name.
- **Avoiding Magic Numbers and Strings:** Using constants or enums instead of magic numbers or strings enhances code clarity. It reduces the need to remember what specific values mean.
- **Code Optimization:** Highly optimized code can be harder to understand due to the use of intricate algorithms or techniques. Balancing optimization with readability is crucial.
- **Version Control:** An organized version control system like Git can help reduce cognitive load by allowing developers to trace the history of code changes and understand the context behind them.
- **Testing and Debugging:** Proper unit tests and debugging tools can lower cognitive load by helping developers quickly identify and fix issues.
- **Dependency Management:** Clear management of dependencies and libraries can simplify code comprehension. Overly complex dependency structures can increase cognitive load.

- **Domain Knowledge:** Understanding the problem domain is vital. When developers have a deep understanding of the problem they are solving, it reduces cognitive load as they can relate the code to the real-world context.

An Example of Badly Abstracted Code

So, now we have an idea of what Abstraction and Cognitive Load are, let's look at a badly abstracted code example. We're going to use a simplified Shopping Cart System, Listing 1 gives us our starting point.

Listing 1.

```
1. <?php
2. class Product {
3.     public function __construct(
4.         public string $name,
5.         public string $price,
6.         public string $category
7.     ) { }
8. }
9.
10. class ShoppingCart {
11.     public $products = [];
12.
13.     public function addProduct($name, $price, $cat) {
14.         $product = new Product($name, $price, $cat);
15.         $this->products[] = $product;
16.     }
17.
18.     public function calculateTotal() {
19.         $total = 0;
20.         foreach ($this->products as $product) {
21.             $category = $product->category;
22.             if ($category == 'Electronics') {
23.                 // 10% tax for electronics
24.                 $total += $product->price * 1.1;
25.             } elseif ($category == 'Clothing') {
26.                 // 20% tax for clothing
27.                 $total += $product->price * 1.2;
28.             } else {
29.                 $total += $product->price;
30.             }
31.         }
32.         return $total;
33.     }
34. }
35.
36. $cart = new ShoppingCart();
37. $cart->addProduct('Laptop', 1000, 'Electronics');
38. $cart->addProduct('Shirt', 20, 'Clothing');
39. $cart->addProduct('Phone', 500, 'Electronics');
40. $cart->addProduct('Jeans', 50, 'Clothing');
41.
42. $total = $cart->calculateTotal();
43. echo "Total cost: $" . $total;
```

Let's break down why the provided code is badly abstracted and explore the problems in detail.

- **Lack of Separation of Concerns:-** The ShoppingCart class handles both storing products and calculating the total cost. This is a violation of the Single Responsibility Principle (SRP) in object-oriented programming.

- Ideally, there should be a separation between the data storage (shopping cart) and the logic to calculate the total cost. This lack of separation makes the code less modular and harder to maintain.

- **Conditional Logic for Tax Calculation:-** The calculateTotal method contains conditional statements to determine the tax rate based on the product's category.

- This conditional logic for tax calculation makes the code more complex and harder to read. If new categories or tax rules are introduced, this code would need modification, leading to potential errors.

- **Limited Reusability:-** The tax calculation logic is tightly coupled with the ShoppingCart class. This means you can't easily reuse this logic in other parts of your application.

- Abstraction would allow you to encapsulate tax calculation in a separate, reusable component, making your code more modular and efficient.

- **Poor Scalability:-** As the codebase grows, managing product categories and tax rates within the calculateTotal method becomes increasingly challenging.

- This lack of abstraction and organization can lead to spaghetti code, making maintenance and scaling a nightmare.

So, as you can see, this becomes really complicated to understand very quickly. Imagine using this on a store the size of Amazon. It would be impossible to manage. Let's take the steps in turn and improve our code to be clearer.

Separation of Concerns

Separation of Concerns (SoC) is a fundamental software design principle in object-oriented programming, emphasizing the division of a software system into distinct, self-contained modules or classes, each responsible for a specific aspect of the application. This principle helps in managing complexity, improving maintainability, and promoting code reusability. Let's explore how SoC applies to the original code and then how we can restructure it to adhere to this principle.

Original Code:

In the original code example, there's a violation of the Separation of Concerns principle. The ShoppingCart class is responsible for both maintaining the list of products and calculating the total cost. Here's what's problematic:

- **Mixed Responsibilities:-** The ShoppingCart class has mixed responsibilities. It's managing the shopping cart (product list) and performing complex calculations, including tax calculation.

- This mixing of responsibilities makes the codebase



harder to understand and maintain.

- **Limited Reusability:-** The tax calculation logic is tightly coupled to the ShoppingCart class, making it challenging to reuse in other parts of the application.
- **Maintenance Challenges:-** Any changes or updates to tax rules or product categories would require modifications in the ShoppingCart class, potentially introducing errors or unexpected behavior.

Listing 2 shows how we can restructure the code to adhere to the SoC principle.

Explanation of Using Category Class:

- **Introduction of the Category Class:-** We've introduced a Category class that encapsulates the category name and its corresponding tax rate.
- **Category Management:-** The ShoppingCart class now maintains a list of categories, allowing you to add and manage categories with their respective tax rates.
- **Tax Calculation with Categories:-** The TaxCalculator class uses the list of categories to find the tax rate

Listing 2.

```
1. <?php
2. class Product {
3.     public $name;
4.     public $price;
5.     public $category;
6.
7.     public function __construct(
8.         $name, $price, $category
9.     ) {
10.         $this->name = $name;
11.         $this->price = $price;
12.         $this->category = $category;
13.     }
14. }
15.
16. class Category {
17.     public $name;
18.     public $taxRate;
19.
20.     public function __construct($name, $taxRate) {
21.         $this->name = $name;
22.         $this->taxRate = $taxRate;
23.     }
24. }
25.
26. class TaxCalculator {
27.     public static function calculateTax(
28.         $product, $categories
29.     ) {
30.         foreach ($categories as $category) {
31.             if ($category->name == $product->category) {
32.                 return $product->price * $category->taxRate;
33.             }
34.         }
35.         return 0; // No tax if category not found
36.     }
37. }
38.
```

Listing 2 continues.

```
39. class ShoppingCart {
40.     public $products = [];
41.     public $categories = [];
42.
43.     public function addProduct($name, $price, $cat) {
44.         $product = new Product($name, $price, $cat);
45.         $this->products[] = $product;
46.     }
47.
48.     public function addCategory($name, $taxRate) {
49.         $category = new Category($name, $taxRate);
50.         $this->categories[] = $category;
51.     }
52.
53.     public function calculateTotal() {
54.         $total = 0;
55.         foreach ($this->products as $product) {
56.             $tax = TaxCalculator::calculateTax(
57.                 $product,
58.                 $this->categories
59.             );
60.             $total += $product->price + $tax;
61.         }
62.         return $total;
63.     }
64. }
65.
66. $cart = new ShoppingCart();
67. $cart->addProduct('Laptop', 1000, 'Electronics');
68. $cart->addProduct('Shirt', 20, 'Clothing');
69. $cart->addProduct('Phone', 500, 'Electronics');
70. $cart->addProduct('Jeans', 50, 'Clothing');
71.
72. // 10% tax for Electronics
73. $cart->addCategory('Electronics', 0.1);
74. // 20% tax for Clothing
75. $cart->addCategory('Clothing', 0.2);
76.
77. $total = $cart->calculateTotal();
78. echo "Total cost: $" . $total;
```

associated with a product's category.

- This eliminates the need for conditional logic and provides a cleaner, more maintainable approach.
- **Improved Extensibility:-** Adding new product categories and tax rates is straightforward by using the `addCategory` method in the `ShoppingCart` class.

By adopting a `Category` class to manage tax rates for different product categories, you've abstracted the tax calculation logic in a clean and extensible manner, adhering to the principles of Separation of Concerns and making your code more readable and maintainable.

So we're already headed in the right direction, but I'm sure we can do more:

A Fully Abstracted Example

Listing 3 on the next page shows the full abstraction with the following key abstraction concepts:

- **Interface for Tax Calculation:-** We define a `TaxCalculator` interface, allowing for various tax calculation strategies.
- **Category-Specific Tax Calculation:-** Each Category now has its own tax calculator, which can be easily customized by implementing the `TaxCalculator` interface.
- **Encapsulation:-** The tax calculation logic is encapsulated within each category, making it easy to swap tax calculation methods as needed.
- **Usage of Anonymous Classes:-** Anonymous classes are used for inline tax calculator implementations, promoting a highly abstracted and flexible approach.

By taking code abstraction to the next level, this example promotes greater flexibility, extensibility, and maintainability while still adhering to clean coding principles. It allows for easily adding new categories and customizing tax calculation strategies without affecting the core logic.

Another Example

We'll look at a simple content management system (CMS) that handles articles. We'll start with a badly abstracted code example shown in Listing 4, explain why it's problematic, and then provide a fully abstracted and improved version.

Issues with the Badly Abstracted Code:

- **Lack of Abstraction:-** The `ArticleManager` class directly deals with the creation, retrieval, and deletion of articles, which is a violation of the Single Responsibility Principle.
- **Limited Extensibility:-** The code is not designed to easily accommodate future changes or additional functionality. For example, if you want to add tags or categories to articles, the code becomes unwieldy.
- **Tight Coupling:-** The code is tightly coupled, making it challenging to substitute or extend components. Any change requires modifications throughout the class.

Listing 4.

```

1. class Article {
2.     public $title;
3.     public $content;
4.
5.     public function __construct($title, $content) {
6.         $this->title = $title;
7.         $this->content = $content;
8.     }
9. }
10.
11. class ArticleManager {
12.     private $articles = [];
13.
14.     public function addArticle($title, $content) {
15.         $article = new Article($title, $content);
16.         $this->articles[] = $article;
17.     }
18.
19.     public function findArticleByTitle($title) {
20.         foreach ($this->articles as $article) {
21.             if ($article->title == $title) {
22.                 return $article;
23.             }
24.         }
25.         return null;
26.     }
27.
28.     public function deleteArticle($title) {
29.         foreach ($this->articles as $key => $article) {
30.             if ($article->title == $title) {
31.                 unset($this->articles[$key]);
32.             }
33.         }
34.     }
35. }
36.
37. $articleManager = new ArticleManager();
38. $articleManager->addArticle('First Article',
39.     'This is the content of the first article.');
```

```

40. $articleManager->addArticle('Second Article',
41.     'Content of the second article.');
```

```

42.
43. $foundArticle = $articleManager->findArticleByTitle(
44.     'First Article');
```

```

45. if ($foundArticle) {
46.     echo "Found article: {$foundArticle->title}";
47. }
48.
49. $articleManager->deleteArticle('First Article');
```




Listing 3.

```
1. <?php
2. interface TaxCalculator {
3.     public function calculateTax(Product $product):float;
4. }
5.
6. class BasicTaxCalculator implements TaxCalculator {
7.     public function calculateTax(
8.         Product $product
9.     ): float {
10.         return 0; // Default tax calculation, no tax
11.     }
12. }
13.
14. class Product {
15.     public $name;
16.     public $price;
17.     public $category;
18.
19.     public function __construct($name, $price, $cat) {
20.         $this->name = $name;
21.         $this->price = $price;
22.         $this->category = $cat;
23.     }
24. }
25.
26. class Category {
27.     public function __construct(
28.         public string $name,
29.         private readonly TaxCalculator $taxCalculator
30.     ) { }
31.
32.     public function calculateTax(
33.         Product $prod
34.     ): float {
35.         return $this->taxCalculator->calculateTax($prod);
36.     }
37. }
38.
39. class ShoppingCart {
40.     private $products = [];
41.
42.     public function addProduct(Product $product) {
43.         $this->products[] = $product;
44.     }
45.
46.     public function calculateTotal(): float {
47.         $total = 0;
48.         foreach ($this->products as $product) {
49.             $total += $product->price +
50.                 $product->category->calculateTax($product);
51.         }
52.         return $total;
53.     }
54. }
```

Listing 3 continued.

```
55.
56. // Usage
57. $cart = new ShoppingCart();
58.
59. $electronicsCategory = new Category(
60.     'Electronics',
61.     new class implements TaxCalculator {
62.         public function calculateTax(
63.             Product $product
64.         ): float {
65.             return $product->price * 0.1; // 10% tax
66.         }
67.     });
68.
69. $clothingCategory = new Category(
70.     'Clothing',
71.     new class implements TaxCalculator {
72.         public function calculateTax(
73.             Product $product
74.         ): float {
75.             return $product->price * 0.2; // 20% tax
76.         }
77.     });
78.
79. $product1 = new Product(
80.     'Laptop', 1000, $electronicsCategory
81. );
82. $product2 = new Product(
83.     'Shirt', 20, $clothingCategory
84. );
85. $product3 = new Product(
86.     'Phone', 500, $electronicsCategory
87. );
88. $product4 = new Product(
89.     'Jeans', 50, $clothingCategory
90. );
91.
92. $cart->addProduct($product1);
93. $cart->addProduct($product2);
94. $cart->addProduct($product3);
95. $cart->addProduct($product4);
96.
97. $total = $cart->calculateTotal();
98. echo "Total cost: $" . $total;
```

Listing 5 shows a fully abstracted and improved version:

Explanation of the Improved Code:

- **Separation of Concerns:-** We've separated the Article class to represent an article's structure and data.
- The ArticleRepository class is responsible for managing articles, adhering to the Single Responsibility Principle.
- **Encapsulation and Dependency Injection:-** The Article class encapsulates its data and provides methods for retrieval, promoting data privacy.
- Dependency injection is used to pass Article objects into the ArticleRepository.
- **Flexibility and Extensibility:-** This code is more adaptable to future changes. Adding features like tags, categories, or improved searching can be done without major modifications.
- **Reduced Coupling:-** Components are loosely coupled, making it easier to replace or extend individual parts of the code.

In this improved code, we've achieved a high level of abstraction, adhering to the principles of clean coding, making it more maintainable, extensible, and adaptable to future requirements. The code focuses on a single responsibility for each class, ensuring better organization and ease of development.

Let's take the abstraction to an even higher level. In Listing 6 on the next page, we'll further abstract the code by introducing an abstract Content class that can represent various types of content, not just articles. We'll also employ a repository pattern for better data management and make use of interfaces for extensibility.

Explanation of the High-Level Abstraction:

- **Content Interface:-** We've introduced a Content interface that defines methods for getting the title and content. This abstracts content to be more than just articles.
- **Extensible Content Types:-** We've added a Video class that implements the Content interface, showing how the code can easily accommodate different content types.
- **Generic Content Repository:-** We've created a ContentRepository interface and an implementation ContentRepositoryImpl, which can handle various types of content (articles, videos, etc.) through the Content interface.
- **Reduced Coupling and Enhanced Flexibility:-** The code is highly abstracted, making it easy to add new content types or features without major modifications.

This high-level abstraction allows for managing various types of content with minimal changes to the code, adhering to clean coding principles. It promotes flexibility, extensibility, and reduced coupling while maintaining a clean and organized code structure.

Listing 5.

```

1. class Article {
2.     private $title;
3.     private $content;
4.
5.     public function __construct($title, $content) {
6.         $this->title = $title;
7.         $this->content = $content;
8.     }
9.
10.    public function getTitle() {
11.        return $this->title;
12.    }
13.
14.    public function getContent() {
15.        return $this->content;
16.    }
17. }
18.
19. class ArticleRepository {
20.     private $articles = [];
21.
22.     public function addArticle(Article $article) {
23.         $this->articles[] = $article;
24.     }
25.
26.     public function findArticleByTitle($title) {
27.         foreach ($this->articles as $article) {
28.             if ($article->getTitle() == $title) {
29.                 return $article;
30.             }
31.         }
32.         return null;
33.     }
34.
35.     public function deleteArticle(Article $article) {
36.         $articles = $this->articles;
37.         foreach ($articles as $key => $storedArticle) {
38.             if ($storedArticle == $article) {
39.                 unset($this->articles[$key]);
40.             }
41.         }
42.     }
43. }
44.
45. $articleRepository = new ArticleRepository();
46. $article1 = new Article('First Article',
47.     'This is the content of the first article.');
```



Listing 6.

```
1. interface Content {
2.     public function getTitle(): string;
3.     public function getContent(): string;
4. }
5.
6. class Article implements Content {
7.     private $title;
8.     private $content;
9.
10.    public function __construct($title, $content) {
11.        $this->title = $title;
12.        $this->content = $content;
13.    }
14.
15.    public function getTitle(): string {
16.        return $this->title;
17.    }
18.
19.    public function getContent(): string {
20.        return $this->content;
21.    }
22. }
23.
24. class Video implements Content {
25.     private $title;
26.     private $url;
27.
28.    public function __construct($title, $url) {
29.        $this->title = $title;
30.        $this->url = $url;
31.    }
32.
33.    public function getTitle(): string {
34.        return $this->title;
35.    }
36.
37.    public function getContent(): string {
38.        return "<iframe src='{$this->url}'></iframe>";
39.    }
40. }
41.
42. interface ContentRepository {
43.     public function addContent(Content $content);
44.     public function findContentByTitle(
45.         string $title
46.     ): ?Content;
47.     public function deleteContent(Content $content);
48. }
49.
```

Listing 6 continued.

```
50. class ContentRepositoryImpl
51.     implements ContentRepository {
52.     private $contents = [];
53.
54.     public function addContent(Content $content) {
55.         $this->contents[] = $content;
56.     }
57.
58.     public function findContentByTitle(
59.         string $title
60.     ): ?Content {
61.         foreach ($this->contents as $content) {
62.             if ($content->getTitle() == $title) {
63.                 return $content;
64.             }
65.         }
66.         return null;
67.     }
68.
69.     public function deleteContent(Content $content) {
70.         $contents = $this->contents;
71.         foreach ($contents as $key => $storedContent) {
72.             if ($storedContent == $content) {
73.                 unset($this->contents[$key]);
74.             }
75.         }
76.     }
77. }
78.
79. $repository = new ContentRepositoryImpl();
80.
81. $article = new Article('First Article',
82.     'This is the content of the first article. ');
83. $video = new Video('Intro Video',
84.     'https://www.youtube.com/embed/12345');
85.
86. $repository->addContent($article);
87. $repository->addContent($video);
88.
89. $foundContent = $repository->findContentByTitle(
90.     'Intro Video'
91. );
92. if ($foundContent) {
93.     echo "Found: {$foundContent->getTitle()}\n";
94.     echo "Display:\n{$foundContent->getContent()}";
95. }
96.
97. $repository->deleteContent($article);
```

The Risks of Over Abstraction

Let's first provide a badly abstracted code example showing in Listing 7. After that, we'll delve into an over-abstracted code example, highlighting the potential risks associated with excessive abstraction.

Listing 7.

```
1. <?php
2. class Employee {
3.     public $name;
4.     public $salary;
5.
6.     public function __construct($name, $salary) {
7.         $this->name = $name;
8.         $this->salary = $salary;
9.     }
10. }
11.
12. class Payroll {
13.     public $employees = [];
14.
15.     public function addEmployee($name, $salary) {
16.         $employee = new Employee($name, $salary);
17.         $this->employees[] = $employee;
18.     }
19.
20.     public function calculateTotalSalary() {
21.         $total = 0;
22.         foreach ($this->employees as $employee) {
23.             $total += $employee->salary;
24.         }
25.         return $total;
26.     }
27. }
28.
29. $payroll = new Payroll();
30. $payroll->addEmployee('John', 50000);
31. $payroll->addEmployee('Alice', 60000);
32.
33. $totalSalary = $payroll->calculateTotalSalary();
34. echo "Total salary: $totalSalary";
```

Issues with Badly Abstracted Code:

- **Lack of Separation of Concerns:-** The Payroll class is responsible for both managing employee data and performing salary calculations, which violates the Single Responsibility Principle.
- **Low Reusability and Flexibility:-** The code is not easily reusable for different salary calculation methods or for handling other aspects of employee management.
- **Tight Coupling:-** The code is tightly coupled, making it challenging to adapt to changes or different employee data structures.

Now, let's explore an example of over-abstracted code and highlight the potential risks shown in Listing 8 on the next page.

Risks of Over-Abstracted Code:

- **Complexity:-** The code introduces unnecessary complexity with multiple classes and interfaces. In this example, managing employees and their salaries is more complicated than needed.
- **Reduced Readability:-** Over-abstraction can make the code harder to read and understand, especially for simple tasks like calculating salaries.
- **Maintenance Challenges:-** Excessive abstraction can result in code that is difficult to maintain and modify. For example, adding a new type of employee may require changes across multiple classes.
- **Decreased Development Speed:-** Over-abstracted code often results in slower development because developers need to navigate complex structures for relatively simple tasks.

In this over-abstracted code, we can observe that while it adheres to abstraction principles, it introduces unnecessary complexity and potential maintenance challenges. It's crucial to strike a balance between abstraction and simplicity to achieve clean and maintainable code.

Striking a Balance

Balancing abstraction and simplicity is crucial to writing clean and maintainable code. The goal is to abstract where it adds value, making the code more flexible and understandable without overcomplicating it. Here are some key principles to achieve this balance:

- **Single Responsibility Principle (SRP):** Each class or module should have one and only one reason to change. When abstracting, ensure that each component is responsible for a single aspect of the application.
- **Keep It Simple:** Start with the simplest solution that works and only add complexity when necessary. Avoid over-engineering by considering the current requirements and potential future changes.
- **Clear and Intuitive Abstractions:** Make abstractions that are intuitive and easy to understand. A well-abstracted codebase should make the code more readable, not less.
- **Use Interfaces Sparingly:** While interfaces can be powerful tools for abstraction, don't create interfaces for every class. Use them when you need to define a contract that multiple classes should adhere to.
- **Abstraction Levels:** Consider different levels of abstraction. High-level abstractions are suitable for complex components, while low-level abstractions work well for small, reusable functions or objects.
- **YAGNI (You Ain't Gonna Need It):** Don't abstract or add complexity based on hypothetical future requirements. Abstraction should be driven by present needs, not speculative ones.



Listing 8.

```
1. interface Employee {
2.     public function getName(): string;
3.     public function getSalary(): float;
4. }
5.
6. class RegularEmployee implements Employee {
7.     private $name;
8.     private $salary;
9.
10.    public function __construct($name, $salary) {
11.        $this->name = $name;
12.        $this->salary = $salary;
13.    }
14.
15.    public function getName(): string {
16.        return $this->name;
17.    }
18.
19.    public function getSalary(): float {
20.        return $this->salary;
21.    }
22. }
23.
24. class ContractorEmployee implements Employee {
25.     private $name;
26.     private $hourlyRate;
27.     private $hoursWorked;
28.
29.     public function __construct(
30.         $name, $hourlyRate, $hoursWorked
31.     ) {
32.         $this->name = $name;
33.         $this->hourlyRate = $hourlyRate;
34.         $this->hoursWorked = $hoursWorked;
35.     }
36.
37.     public function getName(): string {
38.         return $this->name;
39.     }
40.
41.     public function getSalary(): float {
42.         return $this->hourlyRate * $this->hoursWorked;
43.     }
44. }
```

Listing 8 continued.

```
45.
46. class EmployeeRepository {
47.     private $employees = [];
48.
49.     public function addEmployee(Employee $employee) {
50.         $this->employees[] = $employee;
51.     }
52.
53.     public function calculateTotalSalary() {
54.         $total = 0;
55.         foreach ($this->employees as $employee) {
56.             $total += $employee->getSalary();
57.         }
58.         return $total;
59.     }
60. }
61.
62. $employeeRepository = new EmployeeRepository();
63. $employeeRepository->addEmployee(
64.     new RegularEmployee('John', 50000)
65. );
66. $employeeRepository->addEmployee(
67.     new ContractorEmployee('Alice', 30, 160)
68. );
69.
70. $salary = $employeeRepository->calculateTotalSalary();
71. echo "Total salary: $salary";
```


- **Code Reviews and Refactoring:** Regularly review and refactor your code. Over time, your understanding of what should be abstracted may change. Refactoring helps to maintain a good balance.
- **Real-world Testing:** Apply abstractions when they are tested in real-world scenarios. Code abstractions are valuable when they make testing and maintenance easier.
- **Documentation:** Document your abstractions and the reasoning behind them. This helps other developers understand why a specific level of abstraction was chosen.
- **Pragmatism:** Be pragmatic. There is no one-size-fits-all answer. The right level of abstraction depends on the context, the size and complexity of the project, and the team's familiarity with the codebase.

Balancing abstraction and simplicity is an ongoing process. It's about finding the right level of abstraction that solves the current problem efficiently while allowing for future changes with minimal disruption. Regularly revisit your code to ensure that your abstractions remain relevant and aligned with your project's needs.

Next month, we're delving into our next question—"Is Your Code Encapsulated Enough To Be Clear?". You'll have to wait until next month to find out how, but Encapsulation and Abstraction are not the same thing!



In 1983, Christopher was introduced to computers by his dad, at the tender age of 3. now, over 40 years later, he has been working in the industry for over 20 years making an impact across multiple sectors of the industry. Starting with launching the first web development company in Staffordshire, Christopher dealt with the web - when the web was little more than just pretty text. He established the websites for many different businesses in their first inception, before moving onto web applications a little while later. Illness prevented Christopher from working in the industry full time for some considerable time - but recovery meant he could tackle once again the joys of code - but he soon found that his skills had become out of date, so thanks to the School of Code in the UK he was able to return to the workplace with revitalised skills ready to tackle the next wave. Specialising since the School of Code in Readable Code, He has worked with a large number of languages, specialising in supporting businesses to grow standards for their code base, and now he is ready to share his processes with the world.
[@ccmiller2018](https://twitter.com/ccmiller2018)



From Capone to Cray

WHERE COMPUTERS REALLY CAME FROM

by Edward Barnard

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

<https://leanpub.com/capone>



Gratefully Looking Back

Beth Tucker Long

I was recently talking to someone who had no idea what I did. I explained a bit about the programming I did, and even though they are in a completely different field, they had heard just enough about it to know that JavaScript was a thing. Did I work with that? Yes, a bit, but mostly I work with PHP. Oh, PHP is a really old language, isn't it?

That is a tough question. After all, “old” is such a relative term. According to my kids, anything from the “late 1900’s” is “super old”. Then again, compared to COBOL, Fortran, or even C, PHP is pretty young. Either way, it is definitely one of the more established web programming languages of our time.

PHP has come a long way since it was first created as a hobby project to solve a specific problem. This is a testament to the dedicated community around it. Despite being almost entirely run by volunteers, the PHP infrastructure, much like the language itself, has only gotten more organized and more efficient over time. Releases are clearly scheduled, well-documented, and given a set roadmap so that everyone knows how long any given release will be maintained. This all allows developers to rely on the stability of PHP while knowing when upgrades will need to take place to stay running on a fully-supported version.

The PHP Foundation has taken this a step further, adding in more stability and security to the developers working on PHP’s core. Again, started by and supported by PHP community members (there are almost 200 organizations and over 1000 individuals supporting the PHP Foundation), the PHP Foundation organized a way to provide paid development time to bring consistency to internal development and make sure that we are not relying on too few people to know and manage internals. This was such a critical and influential step because even though the PHP community is an amazing group of volunteers, it is unreasonable to expect volunteers to keep such a heavily-relied-upon system running in their free-time. The creation of a foundation acknowledges the

immense value of contributions to PHP’s core and seeks to ensure that those contributions keep coming.

With the release of 8.3 rapidly approaching, take a moment to not only look at the new features PHP is gaining but also how far it has come. It’s hard to believe this started as one person trying to code less and now supports a worldwide industry of people coding more and more amazing things. To everyone who has contributed to PHP, whether it be tests, code, or even financially through the PHP Foundation, you are all so appreciated. Thank you!

Related URLs:

- Donate to The PHP Foundation: <https://thephp.foundation/donate/>¹
- Interview with Rasmus Lerdorf on the beginnings of PHP: <https://phpa.me/webarchive-conversationsnetwork>²
- PHP 8.3: <https://wiki.php.net/todo/php83>³



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

¹ <https://thephp.foundation/donate/>

² <https://phpa.me/webarchive-conversationsnetwork>

³ <https://wiki.php.net/todo/php83>

The PHP
Foundation

The PHP Foundation is a collective established with the non-profit mission to support, advance, and develop the PHP language. We are a community of PHP veterans, community leaders, and technology companies that rely on PHP as a critical digital infrastructure. We collaborate to ensure PHP language long-term success and maintenance.

DONATE TO OUR COLLECTIVE

