



PHP 8 Bits and Git

Free
Sample
Article

PHP 8 Distilled

Applying Best Coding Practices to PHP, Part Two

Power Up with Git

ALSO INSIDE

Education Station:
Using Factories and Hydration

PHP Puzzles:
Grid Mapping

The Workshop:
Git Hooks with CaptainHook

Sustainable PHP:
Deep Problem Analysis

Security Corner:
Circuit Breakers

Community Corner:
An Interview with Andreas Heigl

finally{}
Resolutions

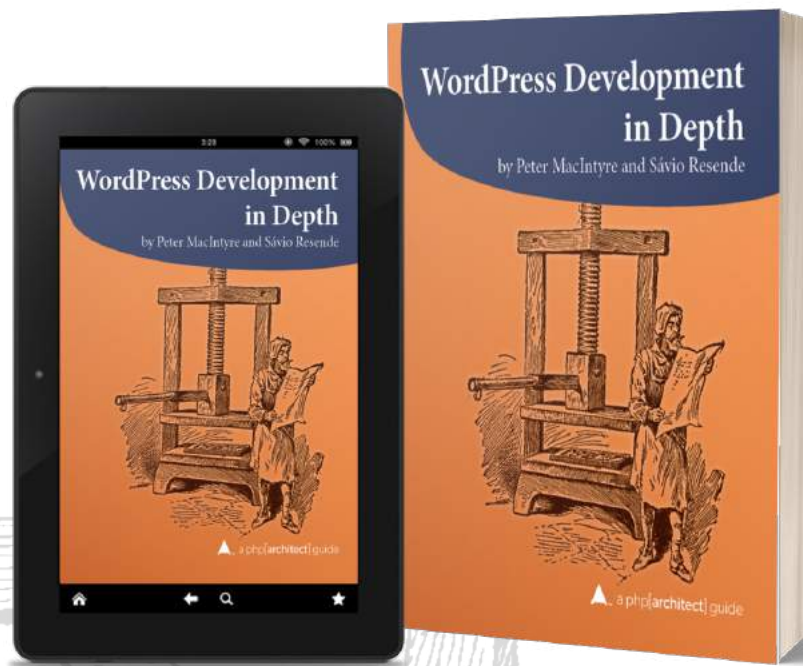


Learn how a Grumpy Programmer approaches testing PHP applications, covering both the technical and core skills you need to learn in order to make testing just a thing you do instead of a thing you struggle with.

The Grumpy Programmer's Guide To Testing PHP Applications by Chris Hartjes (@grmpyprogrammer) provides help for developers who are looking to become more test-centric and reap the benefits of automated testing and related tooling like static analysis and automation.

Available in Print+Digital and Digital Editions.

Order Your Copy
phpa.me/grumpy-testing-book



Build a custom, secure, multilingual website with WordPress

WordPress is more than a blogging platform—it powers one-fifth of all sites. Since its release, enthusiastic contributors have pushed the envelope to use it as a platform for building specialized social networks, e-commerce storefronts, business sites, and more. This guide helps you sort through the vast number of plugins available to find the ones that work for you. You'll also learn how to use CSS and PHP code to tailor WordPress to your needs further.

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

Read a sample and purchase your copy at the link below.

Order Your Copy
<http://phpa.me/wpdevdepth-book>

PHP 8 Bits and Git

Oscar Merida

As scheduled, PHP 8 was released at the end of November. Congratulations are in order for the release maintainers and all contributors involved. In this twelfth issue, we have an in-depth dive into the changes in PHP 8, how to write better code with calisthenics, powering up Git with aliases and hooks, and more.

While wrapping up this issue, news broke that Salesforce is acquiring Slack for \$28 billion. This morning, my Twitter feed had no shortage of messages about how PHP is a core part of the platform. Over at Commit Strip¹, they predict—or is it caution?—that PHP/jQuery/WordPress is a good bet to be around in a decade.

Let's enjoy this moment. PHP continues to mature, as evidenced by the recent release of PHP 8. The only value in any tech stack is tied to how well it helps people solve problems. In the last decade, PHP has shed language features that led to security issues and promoted poor coding practices. Internals has settled into an effective planning model via RFCs that enables consistent annual releases. While new technologies and opportunities to apply them will certainly open up, it would take a massive effort across the web to migrate away from PHP to something else.

Let's start in *PHP 8 Distilled* by Matthew Turland. He writes about new language features you can use, deprecations and removed extensions to watch out for, and tools to streamline upgrading your application code to it. Vinícius Campitelli returns with *Applying Best Coding Practices to PHP, Part Two* to look at how you can use practical rules along with the SOLID principles discussed last month. In *Power Up with Git*, Andrew Woods

explains how you can customize and create new Git commands to make your workflow more efficient. Even if you're a seasoned veteran, check this one out. You're bound to learn a new alias or approach to working with it.

Are you ready to try a puzzle? Sherri Wheeler has one in *PHP Puzzles: Grid Mapping*. She looks at a few solutions for rendering a map grid. Venturing into *The Workshop*, Joe Ferguson shares another way to automate things in *Git Hooks with CaptainHook*. It's handy if you need a straightforward way to share hooks with colleagues. In *Security Corner: Circuit Breakers*, Eric Mann explains how to use the circuit breaker pattern to make your application more resilient. If you depend on one or more third-party APIs, give it a read. Chris Tankersley shares *Using Factories and Hydration in Education Station*. He'll show you how to decouple your objects with factories and clarify the distinction between object creation and adding data to them. Eric Van Johnson has *An Interview with Andreas Heigl* in *Community Corner*. He learns about how Andreas got involved with PHP and his contributions towards moving the PHP documentation to Git. Do you need *Deep Problem Analysis*? Edward Barnard offers a case study in tackling a tricky problem to come up with a working solution. As this year *finally* ends, Beth Tucker Long suggests *Resolutions* we can make to use our tech skills for the common good.



Download the Code Archive:

http://phpa.me/December2020_code

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

¹ Commit Strip: <https://www.commitstrip.com/?p=21058>

PHP 8 Distilled

Matthew Turland

PHP 8 is a significant release for much more than just its version number: it's absolutely packed with shiny new language features, potential performance improvements, and fixes to many unintuitive behaviors and inconsistencies in previous iterations of the language.

This article won't provide a comprehensive review of every new addition or change to the language. For that, it's best to review the relevant RFCs¹, or migration guide². However, it provides an overview of major new features and notable changes, as well as direction on how to get started with your upgrade.

New Features

Constructor Prototype Promotion

One long-held annoyance with the PHP object model involves the amount of boilerplate required to declare instance properties in a class, receive values for them via constructor parameters, and then assign those parameter values to those instance properties.

Constructor prototype promotion³ deals with the typical use case for this situation. It offers a syntax that makes the explicit assignment statements in the constructor body implicit and consolidates the instance property declarations into the constructor parameter declarations, which are then called "promoted parameters." See Listing 1 for what this syntax looks like.

Before any logic in the constructor body executes (assuming it's not empty), assignments for promoted properties happen implicitly in these new constructors. These assignments require the constructor parameters to have the same names as their intended corresponding instance properties.

However, this feature does not support some use cases.

- Parameters with a callable typehint, <https://wiki.php.net/rfc/callable>
- Variadic parameters, <https://wiki.php.net/rfc/variadics>
- Parameters with a default value of null that don't include an appropriate nullable typehint, https://wiki.php.net/rfc/nullable_types
- Parameters in abstract classes and interfaces

1 relevant RFCs: https://wiki.php.net/rfc#php_80

2 migration guide: <https://php.net/en/migration80>

3 Constructor prototype promotion: https://wiki.php.net/rfc/constructor_promotion

4 proposal: https://wiki.php.net/rfc/named_params

Named Arguments

If you've ever used Python, you may already be familiar with this feature through its implementation in that language, known as keyword arguments.

Named arguments have been a long-disputed addition going back years. The original proposal⁴ made in 2013 saw significant updates and eventual acceptance in 2020. Without parser support, many library authors fall back on using arrays to pass in parameters, which is problematic for documenting and enforcing expectations.

To use this feature, you specify values for arguments passed to functions or methods with the name of the corresponding parameter from the function or method signature, rather than passing those arguments in the same positional order as their corresponding parameters in that definition.

Listing 1.

```
1. <?php
2.
3. /* Instead of this... */
4.
5. class Point {
6.     private float $x;
7.     private float $y;
8.     private float $z;
9.
10.     public function __construct(
11.         float $x = 0.0,
12.         float $y = 0.0,
13.         float $z = 0.0,
14.     ) {
15.         $this->x = $x;
16.         $this->y = $y;
17.         $this->z = $z;
18.     }
19. }
20.
21. /* ... you can now do this. */
22.
23. class Point {
24.     public function __construct(
25.         private float $x = 0.0,
26.         private float $y = 0.0,
27.         private float $z = 0.0,
28.     ) {}
29. }
```

Listing 2.

```

1. <?php
2.
3. /* Instead of having to explicitly specify the default
4.  value of the $flags argument, named arguments allow you
5.  to skip it by specifying the name of the following
6.  $double_encode parameter. */
7.
8. htmlspecialchars($string, double_encode: false);
9.
10. /* The usefulness of this becomes more obvious in functions
11.  with a lot of parameters with default values. It also
12.  makes scalar arguments more self-documenting. */
13.
14. setcookie('name', '', 0, '', '', false, true);
15. // versus
16. setcookie('name', httponly: true);

```

Doing so is useful when a function or method has many parameters. Another case is when it has a parameter before the end of the parameter list with a default value you don't want to pass in explicitly. It can also increase the readability of code for function and method calls by making it easier to assess which argument value corresponds to which defined parameter visually.

See Listing 2 for an example of this feature in action. When using a named argument, specify the argument name without the leading \$ included in the parameter name when defining the function or method. A colon (:) follows the name and is then followed by the value for that argument. A comma (,) delimits named arguments as it does traditional positional arguments.

This feature does have some constraints to be aware of.

- Specifying parameter names that are not part of a method or function signature produces an error. Concerning semantic versioning, this means that changing parameter names in method and function signatures are now backward-incompatible change.
- Positional arguments must precede named arguments in calls that use both. Otherwise, they produce a compile-time error.
- Both named and positional arguments must precede any unpacked arguments⁵. Where array keys were previously ignored when using argument unpacking or `call_user_func_*()`, they now map to named arguments.
- Passing the same argument multiple times results in an error. This is the case whether the offending argument is passed by name each time or is specified using both positional and named arguments that correspond to the same parameter.
- Variadic method and function definitions will collect unknown named arguments into the variadic parameter⁶.

5 unpacked arguments: https://wiki.php.net/rfc/argument_unpacking

6 variadic parameter: <https://phpa.me/rfc-variadics-population>

Attributes

Of all the new features in PHP 8, this feature is perhaps the most controversial. Its purpose is to offer a natively-supported way to add metadata to units of code (i.e., classes, methods, functions, etc.). It already sees adoption from projects like Psalm⁷. There's even been some discussion⁸ of trying to standardize attributes across projects with similar purposes to allow for cross-compatibility.

In the past, we've commonly added metadata to code units using docblock annotations, an old concept popularized by projects like:

- phpDoc, <https://www.phpdoc.org>
- PHPUnit, <https://phpunit.readthedocs.io>
- Symfony, <https://symfony.com>,
- Doctrine, <https://doctrine-project.org>,
- and later Psalm, <https://psalm.dev> and Psalm, <https://psalm.dev>

There have even been some attempts to standardize tags for phpDoc-like projects via PHP-FIG proposals PSR-5⁹ and PSR-19¹⁰. The usefulness of annotations has dwindled somewhat in recent years as new language features, such as parameter and return types¹¹, have gradually taken their place. The advantage of attributes is that we can inspect them using the reflection API; no third-party tools or manual DocBlock parsing is required.

The first proposal¹² for attributes came in 2016 but was ultimately declined. The idea went dormant for years before being proposed again¹³ in 2020. Even after its acceptance, it underwent some amendments¹⁴. Then it received a shorter syntax¹⁵. Then that shorter syntax was also amended¹⁶. You can rest assured that the ideas and implementation behind attributes were thoroughly discussed and evaluated.

To show how attributes work, let's look at an example that other language features haven't supplanted yet: the `@link`¹⁷ DocBlock tag, which associates a link to an external resource with the code it annotates. This tag has two parameters: the URI of the resource and an optional description. One potential use for this is linking to bug reports affecting dependencies used by the annotated code.

7 Psalm: <https://psalm.dev/articles/php-8-attributes>

8 some discussion: <https://phpa.me/colinodell-132338>

9 PSR-5: <https://phpa.me/phpdoc-proposed>

10 PSR-19: <https://phpa.me/psr19-doc-tags>

11 parameter and return types: <https://wiki.php.net/rfc/typechecking>

12 first proposal: <https://wiki.php.net/rfc/attributes>

13 proposed again: https://wiki.php.net/rfc/attributes_v2

14 some amendments: https://wiki.php.net/rfc/attribute_amendments

15 shorter syntax: https://wiki.php.net/rfc/shorter_attribute_syntax

16 was also amended:

https://wiki.php.net/rfc/shorter_attribute_syntax_change

17 @link: <https://phpa.me/phpdoc-link>

Listing 3.

```

1. <?php
2.
3. namespace MyNamespace\Attributes;
4.
5. #[Attribute]
6. class Link
7. {
8.     public function __construct(
9.         private string $uri,
10.        private ?string $description = null
11.    ) {}
12.
13.    public function getUri(): string
14.    {
15.        return $this->uri;
16.    }
17.
18.    public function getDescription(): ?string
19.    {
20.        return $this->description;
21.    }
22. }

```

First, we must define the attribute using a class like the one in Listing 3. This class is itself annotated with an attribute named `Attribute` defined by PHP core. `#[` and `]` demarcate the start and end, respectively, of the code for the attribute. This `Attribute` attribute informs PHP that the annotated class represents an attribute. Aside from this, the class looks and functions like any other class.

Next, we use the attribute in a separate class as in Listing 4. Since our attribute class definition exists in a different namespace, we import it with a `use` statement. We then use the attribute to annotate the class declaration, similarly to how we would invoke a function but within `#[` and `]`. We pass in two strings corresponding to the `$uri` and `$description` constructor parameters of the attribute class defined in Listing 4.

Lastly, we can programmatically locate and inspect instances of attributes within the codebase. You can find an example of this in Listing 5, which finds instances of our `Link` attribute used to annotate classes and outputs a list of them. This example is admittedly a bit contrived or incomplete, but its purpose is to provide a simple conceptual illustration of how code can analyze attributes through introspection.

Listing 4.

```

1. <?php
2.
3. namespace MyNamespace;
4.
5. use MyNamespace\Attributes\Link;
6.
7. #[Link('http://tools.ietf.org/html/rfc3986', '(the URI specification)')]
8. class Uri
9. {
10.     /* ... */
11. }

```

Attributes are a subject with enough complexity that they could probably have an entire article dedicated solely to them. If you want a deeper dive into attributes than this article can include, check out Brent Roose's blog post¹⁸ on them.

Union Types

Even if you don't realize it, you've probably already seen conceptual use of union types: before PHP 8, they existed within the `Type` parameter of `@param` DocBlock tags¹⁹; see Listing 6 for an example of what this looks like.

The difference between these union types and those used in DocBlock tags is that PHP uses these for type checking. Rather than leaving parameters untyped and then manually checking their types using the `instanceof` operator in a method or function body, you can use a union type to accomplish the same thing much more concisely and readably.

Listing 5.

```

1. <?php
2.
3. require_once __DIR__ . '/vendor/autoload.php';
4.
5. use MyAttributes\Namespace\Link;
6.
7. foreach (get_declared_classes() as $class) {
8.     $reflector = new \ReflectionClass($class);
9.     $attributes = $reflector->getAttributes(Link::class);
10.    foreach ($attributes as $attribute) {
11.        echo $class, ' - ',
12.            $attribute->getUri(), ' - ',
13.            $attribute->getDescription(), PHP_EOL;
14.    }
15. }

```

Listing 6.

```

1. <?php
2.
3. /**
4.  * @param array|\Traversable $list
5.  */
6. public function doThingWithList($list)
7. {
8.     /* ... */
9. }

```

¹⁸ Brent Roose's blog post: <https://stitcher.io/blog/attributes-in-php-8>

¹⁹ `@param` DocBlock tags: <https://phpa.me/phpdoc-params>

Union types denote that a variable may have a type from a list of two or more possible types. `|`, commonly called the pipe operator, represents the inclusive or bitwise operator²⁰ in other contexts but is also used to separate individual types within union types. For example, a variable that may hold an integer or a floating-point number could have the union type `int|float`.

Union types were first proposed²¹ and were rejected in 2015. Ironically, a little over a year after this rejection, there was a proposal for a single native union type that saw acceptance and implementation in PHP 7.1: the `Iterable` pseudo-type²². `Iterable` solved the issue illustrated in Listing 5 of explicitly supporting both array and `Traversable` values without union types by adding a pseudo-type to represent both of them.

2019 saw a second proposal²³ for union types, this time an accepted one. The implementation leaves a large amount of functionality to potential future scope, such as support for type aliasing. The proposal goes into more detail, but Listing 7 provides a summary of this feature's restrictions in code.

This feature also impacts the reflection API. Specifically, it adds a new subclass of `ReflectionType`, appropriately named `ReflectionUnionType`. This class contains a `getTypes()` method that returns an array of `ReflectionType` instances representing the individual types constituting the relevant union type.

20 bitwise operator: <https://php.net/language.operators.bitwise>

21 first proposed: https://wiki.php.net/rfc/union_types

22 `Iterable` pseudo-type: <https://wiki.php.net/rfc/iterable>

23 second proposal: https://wiki.php.net/rfc/union_types_v2

Listing 7.

```

1. <?php
2.
3. class Number {
4.     /* Union types work for properties... */
5.     private int|float $number;
6.
7.     /* ... parameters ... */
8.     public function setNumber(int|float $number): void {
9.         $this->number = $number;
10.    }
11.
12.    /* ... and return types. */
13.    public function getNumber(): int|float {
14.        return $this->number;
15.    }
16. }
17.
18. /* void cannot be part of a union type. */
19. public function doThing(): int|void; /* This doesn't work. */
20.
21. /* These do the same thing. */
22. public function doThing(): Number|null;
23. public function doThing(): ?Number;
24.
25. /* false functions as a subtype of boolean, true does not. */
26. public function doThing(): int|false; /* This works. */
27. public function doThing(): int|true; /* This doesn't work. */
28.
29. /* Redundant types aren't allowed. None of these work. */
30. public function doThing(): int|int; /*
31. public function doThing(): bool|false;
32. use A as B;
33. public function doThing(): A|B;

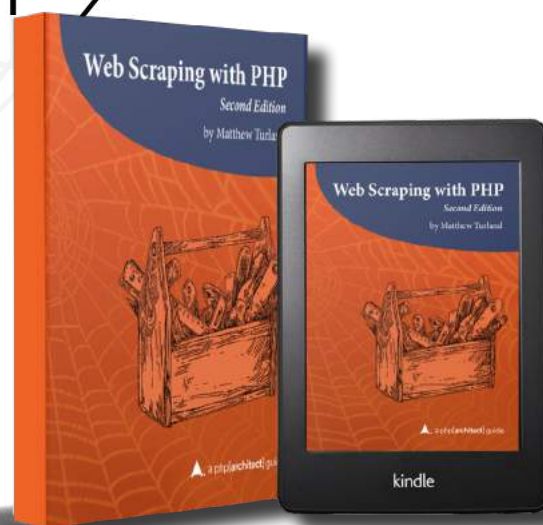
```

What if there's no API?

Web scraping is a time-honored technique for collecting the information you need from a web page. In this book, you'll learn the various tools and libraries available in PHP to retrieve, parse, and extract data from HTML.

Order Your Copy

<http://phpa.me/web-scraping-2ed>



Listing 8.

```

1. <?php
2.
3. /* Instead of this... */
4. $country = null;
5. if ($session !== null) {
6.     $user = $session->user;
7.     if ($user !== null) {
8.         $address = $user->getAddress();
9.         if ($address !== null) {
10.            $country = $address->country;
11.        }
12.    }
13. }
14.
15. /* ... do this. */
16. $country = $session->user->getAddress()->country;

```

Nullsafe Operator

If you need to access an object property or method that's nested within other objects in your hierarchy, it can require verbose checks to ensure that the property or method return value at each level of the call chain is not null.

The nullsafe operator²⁴ `?->` effectively adds a null check against the current value in the chain. If that value is null, further evaluation of the entire expression stops and instead resolves to null. See Listing 8 for an example of what this looks like.

This feature does have some limitations. You can use it in read contexts, but not write contexts. You also cannot return the value of a nullsafe chain by reference²⁵. See Listing 9 for examples of each of these.

Throw Expressions

Before PHP 8, throwing an exception involved a statement using the `throw`²⁶ keyword. A proposal²⁷ changed this situation to make `throw` an expression²⁸ instead.

This change is significant because it makes `throw` usable where it wasn't before, such as inside arrow functions. See Listing 10 for some examples.

Match Expressions

The `switch` statement²⁹ is useful but can be difficult to get right when dealing with many case statements or when some of them omit a `break` or `return` statement.

Listing 9.

```

1. <?php
2.
3. /* Nullsafe chains aren't usable for assignments or
4.    unsetting variables. */
5. $foo?->bar->baz = 'baz';
6. foreach ([1, 2, 3] as $foo?->bar->baz) {}
7. unset($foo?->bar->baz);
8. [$foo?->bar->baz] = 'baz';
9.
10. /* Nullsafe chains are usable in contexts that read an
11.    expression value. */
12. $foo = $a?->b();
13. if ($a?->b() !== null) {}
14. foreach ($a?->b() as $value) {}
15.
16. /* Returning a nullable chain by reference is
17.    not supported. */
18. $x = &$foo?->bar;
19. function &return_by_ref($foo) {
20.     return $foo?->bar;
21. }

```

Listing 10.

```

1. <?php
2.
3. $callable = fn() => throw new Exception;
4.
5. $value = $nullableValue ?? throw new InvalidArgumentException;
6.
7. $value = $falsableValue ?: throw new InvalidArgumentException;
8.
9. $value = !empty($array) ? reset($array)
10.                  : throw new InvalidArgumentException;
11.
12. $condition && throw new Exception;
13.
14. $condition || throw new Exception;

```

After being proposed³⁰, declined, proposed again³¹, and accepted, `match` expressions now provide an alternative. In some ways, in their final form, `match` expressions became for `switch` statements what arrow functions³² are to anonymous functions. They add a syntax that replicates more verbose code using older language constructs.

One significant difference between `match` expressions and `switch` statements is that the former, being expressions, always resolve to a value. Thus, we can use them in assignment statements and anywhere else that an expression is allowed.

Another difference is that `switch` uses a loose comparison (i.e. `==`) while `match` uses a strict comparison that takes the value type into account (i.e. `===`).

24 nullsafe operator: https://wiki.php.net/rfc/nullsafe_operator

25 by reference: <https://php.net/language.references.return>

26 throw: <https://php.net/language.exceptions#example-289>

27 proposal: https://wiki.php.net/rfc/throw_expression

28 expression: <https://php.net/language.expressions>

29 switch statement: <https://php.net/control-structures.switch>

30 proposed: https://wiki.php.net/rfc/match_expression

31 proposed again: https://wiki.php.net/rfc/match_expression_v2

32 arrow functions: https://wiki.php.net/rfc/arrow_functions_v2

Finally, once a `match` expression finds a match, it returns the corresponding value. This behavior contrasts with a `case` statement, which may allow execution to “fall through” if a `break`, `return`, or other terminating statement isn’t present.

See Listing 11 for an example of what `match` expressions look like and how they work compared to `switch` statements.

The outer parts of a `match` expression are like those of `switch` statements: they begin with a keyword (`match`) followed

by a parenthesized expression and then an open curly brace and end with a closing curly brace.

Inside the braces, instead of `case` statements, there are:

1. comma-delimited lists of one or more expressions,
2. a rocket or double-arrow (`=>`),
3. the `match` expression value to return if the parenthesized value matches any values to the left of the arrow,
4. and a trailing comma (,).

As with `switch` statements, the default keyword specifies a fallback case when the parenthesized expression doesn’t match any preceding values. If a `match` statement has no default case and does not match a value, PHP throws an `UnhandledMatchError` instance.

Due to the transition of `throw` from a statement to an expression—described in the previous section—a `match` statement can resolve to a `throw` expression.

Static Return Type

The implementation of late static binding³³ (commonly appreciated as LSB) in PHP 5.3 came at a time when PHP lacked support for return type declarations³⁴, which came later in PHP 7.0. As a result, the `static` keyword was usable in most reasonable contexts in the original LSB implementation except in return types. A proposal³⁵ accepted for PHP 8 fills this gap in the implementation.

See Listing 12 for an example of how this works. A superclass `Foo` declares a method `get()` with a static return type. A `Bar` subclass of `Foo` inherits and does not override this method. Later code calls that method on an instance of `Bar`, and the method’s return value type is an instance of `Bar` rather than `Foo`, which return type checking confirms.

³³ late static binding:
<https://php.net/language.oop5.late-static-bindings>

³⁴ return type declarations:
https://wiki.php.net/rfc/return_types

³⁵ A proposal:
https://wiki.php.net/rfc/static_return_type

JIT

The JIT³⁶, or Just-In-Time Compiler, was initially proposed as an experimental addition to PHP 7.4 but was instead delayed until in PHP 8. There’s quite a bit to say about this feature, but here’s what you need to know.

Several³⁷ source³⁸ have conducted benchmarks and confirmed that many web applications should see minor, if any, performance improvement by enabling this feature. It is significantly more useful to CPU-bound applications: complex mathematical operations such as calculating Mandelbrot fractals³⁹, long-running processes such as applications running on ReactPHP⁴⁰ and other similar asynchronous frameworks, etc.

The RFC details related configuration settings⁴¹ for the JIT; see this deep dive⁴² for explanations of commonly used configurations.

Migrating

Now that you’ve seen some of the shiny new features in PHP 8, let’s talk about how you can ensure a successful migration to it.

Tests

You’ve got automated tests, right?

If you do, try running them against PHP 8. Assuming they have decent code coverage, they should be your first line of defense in uncovering the specific changes in PHP 8 that will impact your codebase.

³⁶ JIT: <https://wiki.php.net/rfc/jit>

³⁷ Several:
<https://phpa.me/sticher-jit-real-life>

³⁸ Sources:
<https://droptica.com/blog/jit-compiler-php-8/>

³⁹ Mandelbrot fractals:
https://en.wikipedia.org/wiki/Mandelbrot_set

⁴⁰ ReactPHP: <https://reactphp.org>

⁴¹ related configuration settings:
https://wiki.php.net/rfc/jit#phpini_defaults

⁴² this deep dive:
<https://thephp.website/en/issue/php-8-jit/>

Listing 11.

```
1. <?php
2.
3. /* Instead of this... */
4. switch ($x) {
5.     case 1:
6.     case 2:
7.         $result = 'foo';
8.         break;
9.     case 3:
10.    case 4:
11.        $result = 'bar';
12.        break;
13.    default:
14.        $result = 'baz';
15.        break;
16. }
17.
18. /* ... do this. */
19. $result = match ($x) {
20.     1, 2 => 'foo',
21.     3, 4 => 'bar',
22.     default => 'baz',
23. };
```

Listing 12.

```
1. <?php
2.
3. class Foo
4. {
5.     public function get(): static
6.     {
7.         return new static;
8.     }
9. }
10.
11. class Bar extends Foo { }
12.
13. $bar = new Bar;
14. $result = $bar->get();
15. /* $result is a type-checked
16.    instance of Bar */
```


If you don't, well, this may be a good time to start writing some⁴³.

Static Analysis

The PHPCompatibility⁴⁴ project leverages the PHP CodeSniffer⁴⁵ static analyzer to detect compatibility issues between PHP versions. They are working on adding support for PHP 8⁴⁶. I'm sure they would love some help!

You can also lean on other static analyzers like Psalm and PHPStan. Bear in mind that these focus more on general code quality than on PHP 8 compatibility specifically, so they may not help as much in the latter regard.

Deprecations and Removals

Features deprecated in the 7.2⁴⁷, 7.3⁴⁸, and 7.4⁴⁹ branches of PHP may not exist in PHP 8 or may face removal in a future major version. These may clue you in to specific areas of your codebase to inspect. Here are some specific deprecated features that have been removed in PHP 8.

- Curly braces for offset access have been removed, <https://phpa.me/2VmyM9Q>.
- `image2wbmp()` has been removed, <https://wiki.php.net/rfc/image2wbmp>.
- `png2wbmp()` and `jpeg2wbmp()` have been removed, <https://wiki.php.net/rfc/deprecate-png-jpeg-2wbmp>.
- `INTL_IDNA_VARIANT_2003` has been removed, <https://phpa.me/3o8VrTn>.
- `Normalizer::NONE` has been removed.
- `ldap_sort()`, `ldap_control_paged_result()`, and `ldap_control_paged_result_response()` have been removed.
- Several aliases for mbstring extension functions related to regular expressions have been removed.
- `pg_connect()` syntax using multiple parameters instead of a connection string is no longer supported.
- `pg_lo_import()` and `pg_lo_export()` signatures that take the connection as the last argument are no longer supported.
- `AI_IDN_ALLOW_UNASSIGNED` and `AI_IDN_USE_STD3_ASCII_RULES` flags for `socket_addrinfo_lookup()` have been removed.
- DES fallback for `crypt()` has been removed; unknown salt formats will now cause `crypt()` to fail.

- The `$version` parameter of `curl_version()` has been removed.

One specific change to be aware of is that the XML-RPC extension⁵⁰ now lives in PECL⁵¹ rather than core; see the related RFC⁵² for details. If you use this extension, you'll need to install it on your server using PECL as part of your upgrade process.

RFCs

The RFC process drives much of the feature development in PHP these days. As such, RFCs are a great source of information about new features and changes to the languages.

Below is a list of some specific RFCs you may want to review involving backward-incompatible language changes. Tests and PHPCompatibility will probably automate a lot of the process of finding instances where these changes will affect your codebase. That said, it still helps to be aware of specific language changes yourself to recognize potential culprits when you encounter related issues.

- Reclassifying engine warnings, https://wiki.php.net/rfc/engine_warnings
- Consistent type errors for internal functions, https://wiki.php.net/rfc/consistent_type_errors
- Ensure correct signatures of magic methods, <https://wiki.php.net/rfc/magic-methods-signature>
- Always generate a fatal error for incompatible method signatures, https://wiki.php.net/rfc/lsp_errors
- Change Default PDO Error Mode, https://wiki.php.net/rfc/pdo_default_errmode
- Variable Syntax Tweaks, https://wiki.php.net/rfc/variable_syntax_tweaks
- Saner numeric strings, <https://wiki.php.net/rfc/saner-numeric-strings>
- Saner string to number comparisons, https://wiki.php.net/rfc/string_to_number_comparison
- Locale-independent float to string cast, <https://phpa.me/3lov9U>
- Change the precedence of the concatenation operator, https://wiki.php.net/rfc/concatenation_precedence
- Stricter type checks for arithmetic/bitwise operators, https://wiki.php.net/rfc/arithmetic_operator_type_checks
- Make sorting stable, https://wiki.php.net/rfc/stable_sorting

Upgrade Notes

RFCs don't cover all major changes to PHP. For a source that does, look no further than the migration guide⁵³. These are as comprehensive and detailed as you'll find, so combing

43 start writing some: <https://phpa.me/phpunit-writing-tests>

44 PHPCompatibility: <https://github.com/PHPCompatibility/PHPCompatibility>

45 PHP CodeSniffer: https://github.com/squizlabs/PHP_CodeSniffer

46 adding support for PHP 8:

<https://github.com/PHPCompatibility/PHPCompatibility/issues/809>

47 7.2: https://wiki.php.net/rfc/deprecations_php_7_2

48 7.3: https://wiki.php.net/rfc/deprecations_php_7_3

49 7.4: https://wiki.php.net/rfc/deprecations_php_7_4

50 XML-RPC extension: <https://php.net/book.xmlrpc>

51 PECL: <https://pecl.php.net>

52 related RFC: https://wiki.php.net/rfc/unbundle_xmlrpc

53 migration guide: <https://php.net/en/migration80>

through it can be a bit tedious. As such, you'll generally want to rely more on methods mentioned in previous sections of this article first before resorting to consulting this reference.

One example of a change that didn't involve an RFC is the deprecation of the Zip extension procedural API⁵⁴. Another is that many core extensions now return objects where before they returned resources⁵⁵. This change should be transparent in most circumstances except those involving `is_resource()` checks on the affected functions' return values. Below is a list of extensions affected by this change; the upgrade notes cover the affected methods' specifics.

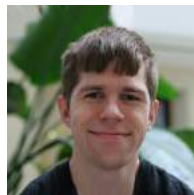
- cURL, <https://php.net/book.curl>
- Enchant, <https://php.net/book.enchant>
- GD, <https://php.net/book.image>
- OpenSSL, <https://php.net/book.openssl>
- Shmop, <https://php.net/book.shmop>
- Sockets, <https://php.net/book.sockets>
- Semaphore, <https://php.net/book.sem>
- XML, <https://php.net/book.xml>
- XMLWriter, <https://php.net/book.xmlwriter>
- XML-RPC, <https://php.net/book.xmlrpc>
- Zlib, <https://php.net/book.zlib>

⁵⁴ deprecation of the Zip extension procedural API:
<https://github.com/php/php-src/pull/5746>

⁵⁵ now return objects where before they returned resources:
<https://github.com/php/php-tasks/issues/6>

Fin

This article informs you of the multitude of reasons to upgrade and available tools to use as you start on your migration journey, and it calls out potential pitfalls to watch out for. Go forth, happy upgrading, and enjoy PHP 8!



Matthew Turland has been working with PHP since 2002. He has been both an author and technical editor for php[architect] Magazine, spoken at multiple conferences, and contributed to numerous PHP projects. He is the author of php[architect]'s "Web Scraping with PHP, 2nd Edition" and co-author of SitePoint's "PHP Master: Write Cutting-Edge Code." In his spare time, he likes to bend PHP to his will to scrape web pages and run bots. @elazar

Related Reading

- *finally{}: What's in PHP Eight?* by Eli White, May 2020.
<https://www.phparch.com/?p=13927>
- *Community Corner: PHP 8 Release Managers: Interview with Sara Golemon and Gabriel Caruso, Part 1* by Eric Van Johnson, July 2020.
<https://www.phparch.com/?p=14152>

Listen to Ep. 44:

Listen to Eric van Johnson, John Congdon, and Oscar Merida discuss practical uses for scalar type hints in PHP. SOLID principles for programming, the peculiarities of floating point math and handling money calculations as a result, and more.



<https://phpa.me/podcast-ep-44>



Borrowed this magazine?

Get php[architect] delivered to your doorstep or digitally every month!

Each issue of php[architect] magazine focuses on an important topic that PHP developers face every day.

We cover topics such as frameworks, security, ecommerce, databases, scalability, migration, API integration, devops, cloud services, business development, content management systems, and the PHP community.



**Digital and Print+Digital
Subscriptions
Starting at \$49/Year**

http://phpa.me/mag_subscribe