Exploring PHP 8.0

Larry Garfield

platform.sh

Exploring PHP 8.0

Larry Garfield

This book is for sale at http://leanpub.com/exploringphp80

This version was published on 2020-12-11

Contents

Introduction

PHP is a funny language. Not in the sense that the syntax is weird (although certainly some would argue that), but in the sense that it is one of the extremely few major Free Software projects that manages to survive and thrive without any major corporate backing.

There are thousands of companies in the PHP ecosystem, and many of them do support the PHP project in some way either financially or in-kind, but development of PHP itself has historically been almost entirely volunteer-driven. As of the end of 2020, I am aware of only one person who is paid to work on PHP itself full time. (You will probably guess who that is for yourself by the end of the book.)

Despite that, every year the PHP community manages to churn out a new release of the engine with performance and stability improvements, feature enhancements, and more polish than the version before. That's despite having one of the most open and public development processes of any language, inviting the largest number of cooks into the kitchen. (There are nearly 2000 people with voting rights on PHP RFCs.) The fact that all these cooks bustling about in this chaotic kitchen succeed in pulling something savory out of the oven year after year is nothing short of a miracle–a welcome one.

PHP 8.0 is the latest release of the web's most popular language, released just in time for PHP's 25th anniversary. Over 150 people contributed code to this latest release, and even more helped with reviews, feedback, and documentation, nearly all of them volunteers. For the past year PHP has had a steady trickle of new and exciting functionality being proposed, voted on, and merged. While a few headline features will no doubt take the lion's share of the attention, a huge assortment of "down ticket" improvements add up to one of the most exciting releases in years. Several long-requested features finally made it in, many long-standing logic bugs were addressed, and many new and exciting ideas were introduced.

If there is a theme to this release, I would say it's conciseness and consistency. Several new syntactic features serve primarily to make common patterns easier and shorter to write, while most of the bug fixes and minor changes involve making the language more consistent and less weird in various ways. The net result is a language that is more pleasant to use, but requires less typing. That's a net win.

This book attempts to summarize and describe the full breadth of changes in PHP 8.0. It covers every RFC (Request For Comments, or formal change proposals for PHP), as well as nearly all smaller changes that were deemed small enough to bypass the formal process. Measured as test coverage, I'd say it's easily 98%, which is better than most applications on the market.

This book is offered to the PHP community free of charge from Platform.sh, a continuous deployment cloud hosting company. We host applications in a half-dozen languages, but PHP remains the backbone of our business. While you can host PHP 8.0 with Platform.sh, most of this book is, of course, entirely valid and relevant for any PHP 8.0 environment. It is based on an 11-part epic series of blog posts leading up to PHP 8.0's release, but has been revised, edited, and expanded considerably from those posts. It is our gift to you.

This book is divided into four parts. In "Headline Features" we look at the top-billed new features that make PHP 8.0 exciting. In "Feature improvements" we go over other changes that may not get their names in lights but are still solid and welcome additions to the language. "Changes and backward compatibility challenges" covers the many examples of oddball, silly behavior that are no longer oddball and silly, which is fine unless you were relying on them being oddball and silly and so need to watch out for slight changes in behavior. Finally, "PHP 8 on Platform.sh" walks through the easiest way to get up and running with PHP 8.0 in production, using Platform.sh.

I would like to especially thank:

- My manager, Robert Douglass, for green-lighting the blog series that this book is based on.
- Christopher Lockheardt, from our marketing team, who has been diligently editing this whole project from the beginning and fighting me over every comma.
- Marina Gulova, from our design team, who provided the graphics for the blog series as well as this book.
- Chad Carlson, my colleague on the Developer Relations team, who wrote the original version of the "PHP 8.0 on Platform.sh" section as part of our Getting Started guide for customers.

And, of course, everyone who contributed in some way to PHP 8.0, from RFC authors to people who submit good bug reports. All of them helped to make this the most exciting PHP release in half a decade and set PHP on course for another 25 years of success.

May your ElePHPants always be smiling.

Headline features

The list of new functionality in PHP 8.0 is long, but there a are a few stand-out features that define every release. For PHP 8.0, there are four leading features that will gather the lion's share of attention:

- Constructor Property Promotion
- The Just-in-Time compiler
- Attributes
- Named Arguments

Let's look at each one in turn.

Constructor Property Promotion

Perhaps the largest quality-of-life improvement to PHP 8.0 is Constructor Property Promotion. The syntax has been borrowed largely from Hack, Facebook's PHP fork, although it exists in other languages like TypeScript and Kotlin with various syntaxes.

It's easiest to explain with an example. Consider your typical service class in a modern application:

```php
class FormRenderer
{
    private ThemeSystem $theme;
    private UserRepository $users;
    private ModuleSystem $modules;

    public function __construct(
        ThemeSystem $theme,
        UserRepository $users,
        ModuleSystem $modules
    ) {
        $this->theme = $theme;
        $this->users = $users;
        $this->modules = $modules;
    }

    public function renderForm(Form $form): string
    {
        // ...
    }
}
```

That class has three dependencies. Each one has to be named *four times*: once to declare the property, once for the constructor parameters, once when referencing the parameter, and once when referencing the property. That is ridiculously verbose, and while many IDEs have some code completion features to help, it's still a pain. I have known people to avoid classes and dependency injection purely because of that verbosity.

As of PHP 8.0, that can all be collapsed down to this:

```php
class FormRenderer
{
    public function __construct(
        private ThemeSystem $theme,
        private UserRepository $users,
        private ModuleSystem $modules
    ) { }

    public function renderForm(Form $form): string
    {
        // ...
    }
}
```

Moving the visibility marker into the constructor declaration tells PHP "this parameter is also a property, you should assign the passed value to it." The effect at runtime is exactly the same as the first example, but it requires one-fourth as much typing. That's a win.

It's still entirely possible to declare additional properties the traditional way, and the constructor can also still have a body. In this example the body is empty, but if there is more work to do in the constructor than "assign stuff to properties," that code can still be included and will execute after the assignment (or "promotion") is done.

```php
class FormRenderer
{
    private User $currentUser;

    public function __construct(
        private ThemeSystem $theme,
        private UserRepository $users,
        private ModuleSystem $modules
    ) {
        $this->currentUser = $this->users->getActiveUser();
    }

    public function renderForm(Form $form): string
    {
        // ...
    }
}
```

In practice, the vast majority of classes I've written in the last few years have had such "boring" constructors, which means property promotion will dramatically cut down the amount of typing needed in the future.

Valuing your data

Constructor property promotion is even more valuable on value objects. (Value objects and service objects should make up the vast majority of your classes; there are only a tiny few other good models in PHP.) Many value objects (or Data Transfer Objects, or various other names they go by) are in practice little more than structs: a collection of named properties with, possibly, getters and setters on them. In that case, the entire class can now be reduced to little more than a constructor:

```php
Class MailMessage
{
    public function __construct(
        public string $to,
        public string $subject,
        public string $from,
        public string $body,
        public array $cc,
        public array $bcc,
        public string $attachmentPath
    ) {}
}
```

Making the properties public may be uncomfortable for some. In general, I would argue that for an internal (not part of the API) object it's fine, but for an object that is part of a public API making a series of getter methods and making the properties private is worthwhile. It's still less typing than it was, and it's also faster and far more memory efficient than using arrays[1].

It's also entirely allowed to have only some constructor parameters promoted and not others. The ones that are not will behave the same as they always have.

As usual, there are a few caveats.

- The visibility keywords only apply for constructors; using them in any other function or method signature is an error.
- A property may not be defined both in the constructor signature and on its own. If the property needs more than can be included in the signature, do it the old style way. That still works fine.
- Properties cannot be typed as callable (for implementation complexity reasons we won't get into here), so neither can constructor promoted properties. They can be left untyped, however.
- Variadic arguments cannot be promoted, as that makes the types harder to juggle.

We have Nikita Popov to thank for this RFC[2]. Get used to him, because we'll be seeing his name a lot in this book.

[1]https://peakd.com/php/@crell/php-use-associative-arrays-basically-never
[2]https://wiki.php.net/rfc/constructor_promotion

Just-In Time-compilation

Some background

Computers don't actually understand programming languages; they understand very low level instructions no human could write by hand. There are many ways of getting from a human-readable language like PHP or Rust to a computer-understandable set of instructions.

The most basic, and usually most performant, way is to compile the human-friendly source code directly to CPU instructions "Ahead-of-Time" (AOT). Those instructions then get saved to a "binary" stand-alone executable file. Some common languages that take this approach include C, C++, and Rust.

The least performant, but usually easiest to write, translation method is interpreted languages, often called "scripting languages." In this case, there's an "interpreter" program that translates each statement of source code into machine code as it's "executed." PHP 3 worked like that, which is why PHP 3 was so amazingly slow compared to modern PHP.

Another approach is to use a "virtual machine." A virtual machine works in a similar fashion to an interpreter, but first it converts the source code into a simpler language, essentially a very low-level scripting language, that can be then interpreted much faster. This approach is very popular these days as it offers a good balance between complexity, ease of development, and performance. Sometimes that "simpler language" conversion happens ahead of time—as in Java, C#, or Go—and sometimes it happens on-the-fly—as in PHP, Python, or Javascript. Java calls that simplified version "bytecode." PHP uses the term "opcodes." It's the same idea.

Just in Time

The latest trend in compilation is the introduction of a "Just in Time" compiler, or JIT. A JIT compiler starts with the simplified intermediary language, and rather than interpreting it, it converts it on-the-fly to machine code, stores that machine code in memory, and executes that.

JIT compilers are very tricky, because in order to get good performance out of them you generally need to be selective in which parts of the intermediary language are compiled to machine code and which aren't. It's not always faster to convert to machine code, depending on the specific details of the code and the language in question. Additionally, the process of converting the simplified code to native machine code may take longer than just running the simplified code once and being done with it.

For that reason, most JIT compilers analyze the code as it's running to identify what parts would give the best bang for the buck and then compile just those bits. The net result, in theory, is that the program literally gets faster as it runs and as the JIT compiler in the virtual machine learns what parts of the code to bother optimizing.

Java was the first widespread language to include a JIT in its virtual machine. Most major Javascript engines now do as well. And, as of PHP 8.0, PHP has joined that list.

The PHP JIT

PHP's new JIT has been a long time coming. It's actually been under development for several years and nearly shipped in an earlier form in PHP 7.4. Work toward making PHP JIT-capable was the impetus that led to the major rewrite of the engine that gave 7.0 its massive performance boost.

The PHP JIT is built as an extension to the opcode cache. That means it can be enabled and disabled either when building PHP itself or at runtime, via `php.ini`. It also means the opcache must be enabled for it to work. The opcache is enabled by default for web requests, but not for CLI commands. To enable it in CLI commands, either add a line to your CLI's `php.ini` file:

```
opcache.enable_cli = 1
```

or specify it explicitly on the command line:

```
php -dopcache.enable=1 myscript.php
```

How to configure the JIT

By default, the JIT is enabled, but the "buffer size" is set to 0, which has the same effect as being disabled. To activate it, set the `opcache.jit_buffer_size` setting to a non-zero value. That controls how much space in memory the JIT can fill up with its optimized machine code. More is not always better, though, as the JIT could also waste time compiling code that doesn't really benefit from being compiled.

The other main setting is `opcache.jit`, which controls the level of JIT aggressiveness. It can accept either a string or a numeric code for advanced usage.

The string options are

- `disable` (hard off)
- `off` (can still be enabled at runtime)
- `tracing` (the more aggressive option, and the default)
- `function` (a less aggressive option that doesn't optimize code across function boundaries)

For advanced usage, `opcache.jit` can also accept a 4-digit numeric sequence, although it's not a number. Each number represents a different aggressiveness setting: optimization level, trigger, register allocation, and CPU-specific optimization flags. The PHP documentation describes them all[3] in more detail, so I won't go into further detail here, other than to note that the `tracing` and `function` presets are already pretty aggressive. In practice, you'll likely only need to use those settings to scale the JIT back in case of bugs, not to push it harder.

There are several other `opcache.jit_*` settings to further fine-tune the JIT settings, but they're mostly for debugging or very advanced usage so I'll defer to the official documentation.

There is no universally best configuration for the JIT. As is often the case with advanced tools like this, you'll need to experiment with your own application and tune it appropriately.

Will it help?

But will the JIT improve performance? The predictable answer, as always, is "it depends." For web apps, kinda maybe. For PHP as an ecosystem, immensely.

PHP, by design, usually runs in a shared-nothing configuration. After each request is handled, the program exits entirely. That gives the JIT very little time to analyze and optimize code, especially since most code in a typical web request is only executed once as the request is handled linearly. Besides, the largest part of those applications is often I/O (talking to the database, mainly), and the JIT can't help with that at all. The benchmarks that have been published so far show the JIT offering only a marginal boost to performance in typical PHP applications run through PHP-FPM or Apache.

Where the JIT has the potential to be really helpful is in use cases where PHP is often not considered today. Persistent daemons, parsers, machine learning, and other long-running CPU intensive processes are where the real benefits lie. Much like the Foreign Function Interface (FFI) support added to PHP 7.4, the goal here is to allow PHP to break out of being a first-class web language and into a first-class general server language.

PHP-Parser[4] is "a PHP parser written in PHP." It's from the same Nikita Popov we've been mentioning throughout this book and is used by many static analysis tools on the market today, such as PHPStan[5]. Nikita has reported that PHP-Parser runs in some cases as much as twice as fast with the new JIT engine.

Persistent applications like React PHP[6] or AmPHP[7] will likely see a notable improvement as well, although not quite as much as they tend to do a lot of I/O (where the JIT is not helpful).

There are machine learning libraries available for PHP, such as Rubix ML[8] or PHP-ML[9]. They're not as widely used as their Python equivalents, in part because, being interpreted, they tend to be

[3]https://www.php.net/manual/en/opcache.configuration.php#ini.opcache.jit
[4]https://github.com/nikic/PHP-Parser
[5]https://github.com/phpstan/phpstan
[6]https://reactphp.org/
[7]https://amphp.org
[8]https://rubixml.com/
[9]https://php-ml.readthedocs.io/en/latest/

slower than the C libraries with nice Python wrappers. With a JIT, however, these CPU-intensive tasks may end up being just as fast or possibly even faster as those available in other languages.

PHP is no longer just the fastest of the major web scripting languages. It's now a viable high-performance general data processing language, which puts persistent workers, machine learning, and other high-CPU tasks into the hands of millions of existing PHP developers around the world.

We primarily have Dmitry Stogov and Zeev Suraski to thank for the multi-year effort to make this RFC[10] happen.

[10]https://wiki.php.net/rfc/jit

Attributes

A history of change

Some PHP changes are proposed, discussed, implemented, and approved in short order. They're uncontroversial, popular, and have a natural way to implement them.

And then there are the ones that are tried, fail, and come back multiple times before they are finally accepted. Sometimes the implementation takes a long time to sort out, sometimes the idea itself is only half-baked, and sometimes the community just hasn't warmed up to an idea yet—"it's not yet time."

Attributes fall squarely into the latter category. They were first proposed back in 2016[11], for PHP 7.1, but met with stubborn resistance and soundly lost the acceptance vote. Fast forward four years and a very similar if slightly reduced-scope proposal sailed through with only one dissenter. It's apparently an idea whose time has indeed come.

Getting meta

So what are attributes? Attributes are declarative metadata that can be attached to other parts of the language and then analyzed at runtime to control behavior.

If you've ever used Doctrine Annotations[12], attributes are essentially that, but as a first-class citizen in the language syntax. The terms "attributes" and "annotations" are used roughly interchangeably in the various languages that support them. PHP went with the term "attribute" specifically to reduce confusion with Doctrine Annotations, since they have a different syntax. Because attributes are built into the language, though, rather than being simply a user-space documentation convention, they can be linted and type checked by static analyzers, IDEs, and syntax highlighters.

The specific syntax was the subject of far more debate than the feature itself, but I'm going to skip over all of that and just look at the final result. Attributes in PHP 8.0 borrow their syntax from Rust:

[11]https://wiki.php.net/rfc/attributes
[12]https://www.doctrine-project.org/projects/doctrine-annotations/en/1.10/index.html

```php
#[GreedyLoad]
class Product
{

    #[Positive]
    protected int $id;

    #[Admin]
    public function refillStock(int $quantity): bool
    {
        // ...
    }
}
```

Those various #[...] blocks are attributes. At runtime, they do … absolutely nothing. They have no impact on the code itself. However, they are available to the reflection API, which allows other code to examine the `Product` class, or its properties and methods, and take additional action. What that action is, well, that's up to you.

As a side effect, attributes are interpreted as comments in older PHP versions and thus ignored, if they're single-line. That's more a useful side effect than a deliberate feature, but is worth noting. It's probably rendering as a comment in your browser, too. It will take a little while for syntax highlighters and IDEs to catch up, but expect that to happen soon enough.

An important note about attributes is that they're not limited to strings. In fact, in the overwhelming majority of cases they will be objects, which can take parameters.

Attributes can be attached to classes, properties, methods, functions, class constants, and even to function/method parameters. That's even more places than Doctrine annotations.

A practical example

All of this seems rather abstract, so let's take a practical example. Most uses of attributes/annotations in PHP today revolve around registration. That is, they're a declarative way to tell one system details of another system, such as a plugin or event system. To that end, I have updated my PSR-14 Event Dispatcher[13] implementation, Tukio[14], to support attributes. Here's how it works (somewhat simplified).

Tukio provides a "listener provider" that aggregates and orders "listeners," that is, any type of callable to which an event can be passed. Normally, you can register a listener like so:

[13]https://www.php-fig.org/psr/psr-14/
[14]https://github.com/Crell/Tukio

```php
function my_listener(MyEventType $event): void { ... }

$provider = new OrderedListenerProvider();

$provider->addListener('my_listener', 5, 'listener_a', MyEventType::class);
```

Those parameters are the callable to add (in this case the function name) and then optionally a priority for ordering, a custom ID, and the type of event to listen to. The latter two are generally derived automatically, but I'm including them here to demonstrate that you can specify them on the fly. There are also methods to add listeners before or after another listener, based on its ID.

Specifying all of those on the fly is a pain, though. It would be especially nice to specify the ID along with the listener, for instance, so it's consistent. Attributes allow us to do that.

First, we define our new attribute. Attributes are a normal PHP class that themselves have a special attribute:

```php
namespace Crell\Tukio;

use \Attribute;

#[Attribute]
class Listener implements ListenerAttribute
{
   public function __construct(
       public ?string $id = null,
       public ?string $type = null
   ) {}
}
```

The `#[Attribute]` attribute tells PHP "yes, this class can be loaded as an attribute." Also note that, as attribute is itself a class, it is subject to namespace rules and needs to be `use`d at the top of the file.

The class then also defines a constructor … using the new constructor promotion syntax. This is strictly an internal data class, so we'll just give it two optional public properties that are populated from the constructor. (In PHP 7.4, the same code would be twice as long.)

Let's go a step further; it doesn't make any sense to put a `Listener` attribute on a parameter or a class, just on functions and methods. We can therefore tell the class that it's a valid attribute only on functions and methods, using a series of bit flags:

```php
namespace Crell\Tukio;

use \Attribute;

#[Attribute(Attribute::TARGET_FUNCTION | Attribute::TARGET_METHOD)]
class Listener implements ListenerAttribute
{
   public function __construct(
       public ?string $id = null,
       public ?string $type = null
   ) {}
}
```

Now, if we try to put a `Listener` attribute on a not-function, not-method, it will throw an error when we try to use it.

That also suggests how we'll use the `Listener` attribute to pass in parameters.

```php
use Crell\Tukio\Listener;

#[Listener('listener_a')]
function my_listener(MyEventType $event): void { ... }

$provider = new OrderedListenerProvider();

$provider->addListener('my_listener', 5);
```

By itself that does nothing. But once given the name of the function, we can use the reflection API to pull out that information. A (very) simplified version of what Tukio does here is:

```php
Use Crell\Tukio\Listener;

/// A string means it's a function name, so reflect on it as a function.
if (is_string($listener)) {
    $ref = new \ReflectionFunction($listener);
    $attribs = $ref->getAttributes(
        Listener::class,
        \ReflectionAttribute::IS_INSTANCEOF,
    );
    $attributes = array_map(
        fn(\ReflectionAttribute $attrib) => $attrib->newInstance(),
        $attribs,
    );
}
```

In this example, `$attribs` is an array of `\ReflectionAttribute` objects. There are a few things you can do with one of those. In particular, the attribute doesn't *have* to be a class! You can get just the name and arguments (if any) as a string and array, respectively. That can be helpful for static analysis or for simple flag attributes whose existence already tells you everything you need to know.

The `getAttributes()` method can also filter the attributes on a value for you. In this case, we're limiting it to only return `Listener` attributes and to allow subclasses of `Listener` to be included as well. While optional, I strongly recommend doing both as it naturally filters out attributes from other libraries you may not expect, and allowing subclasses or implements means you can cluster a series of attributes together using an interface.

The last line maps over that array and calls `newInstance()` on each attribute. That directly calls the `Listener` constructor with the parameters specified, then returns the result. The object that comes back is identical to what you would get had you just written `new Listener('listener_a')`. Beyond that constructor call the attribute class can do or have anything any other class can do or have. You could give it complex internal logic or not, handle default values, make certain parameters required, etc., as your situation requires.

Now, Tukio can combine the data from the `addListener()` method call and the `Listener` attribute however it wants. In this case, the effect is the same as if you had specified the ID in the method call rather than in the attribute.

A single language item can be tagged with multiple attributes, even attributes from entirely different libraries. Attributes can also allow themselves to be repeated on a single language item or not. Sometimes it may make sense to allow the same attribute twice, other times not. Both can be enforced.

There's a lot more that you can do, and Tukio actually has multiple attributes it supports for different use cases and situations that I won't go into here for brevity, but that should give you a taste of what's possible.

Coming soon to a framework near you

Attribute support is already appearing in frameworks. Symfony 5.2 includes attribute versions of route definition[15], for instance. That means you can now declare a controller and wire it up to a route like so:

[15]https://symfony.com/blog/new-in-symfony-5-2-php-8-attributes

```php
use Symfony\Component\Routing\Annotation\Route;

class SomeController
{
    #[Route('/path', 'action_name')]
    public function someAction()
    {
        // ...
    }
}
```

For the most part, that's just swapping Doctrine annotations syntax for native attributes to achieve the same goal. However, it's also going a step further. You can even control what dependencies get passed to your controller parameters[16] via attributes on the parameters, something Doctrine couldn't do.

```php
namespace App\Controller;

use App\Entity\MyUser;
use Symfony\Bundle\FrameworkBundle\Controller\AbstractController;
use Symfony\Component\Security\Http\Attribute\CurrentUser;

class SomeController extends AbstractController
{
    public function index(#[CurrentUser] MyUser $user)
    {
        // ...
    }
}
```

That tells the argument resolvers to pass the current user, specifically, to the `$user` parameter.

No doubt we'll see more use of attributes in other frameworks and libraries as well now that it has first-class native support in the language.

Promoted arguments

It's also important to note how attributes work on promoted constructor arguments. Recall that in PHP 8.0, it's now possible to specify an object property and constructor argument at the same time when one just maps to the other, like so:

[16]https://symfony.com/blog/new-in-symfony-5-2-controller-argument-attributes

```php
class Point
{
    public function __construct(public int $x, public int $y) {}
}
```

Both arguments and properties can have attributes, however. So in this example, does the attribute refer to the argument or the property?

```php
class Point
{
    public function __construct(
        #[Positive]
        public int $x,
        #[Positive]
        public int $y
    ) {}
}
```

Technically it could be either, but it may not always make sense for one or the other. The engine can't know which one it's supposed to be.

For 8.0, the decision was to do both. The attribute will be available on both the property and the argument. If that doesn't make sense for your application, either filter it out in your own application logic or don't use constructor promotion for that one parameter, so you can attribute it in just one place and not the other.

Native attributes

In 8.0, there are no attributes that mean anything to the PHP engine, aside from the `\Attribute` class itself. However, it's unlikely to stay that way. The RFC called out several possible future attributes that could have meaning to the engine in the future.

For example, a `#[Memoize]` attribute could tell the engine that a function or method is safe to cache, and the engine could then automatically cache the results of that function. Another option would be a `#[Jit]` attribute that could provide hints to the JIT engine that a given function or class is a good or bad candidate for JIT compilation or tweak how it gets compiled. Perhaps even an `#[Inline]` attribute that could tell the compiler it should try to inline a function call, thus saving time on the call itself.

None of these engine attributes exist yet, but they're the sort of ideas that are now available to engine developers to further enhance the language in future versions.

A long list of credits

Attributes were developed and modified over the course of a couple of RFCs.

The original RFC[17] comes courtesy of Benjamin Eberlei and Martin Schröder and used a different syntax borrowed from older versions of Hack. A follow-up RFC[18] from the same people tweaked the features a bit.

The syntax was changed twice after that, in RFCs from Theodore Brown and Martin Schröder[19] and then from Derick Rethans and Benjamin Eberlei[20] to settle on the final syntax.

[17]https://wiki.php.net/rfc/attributes_v2
[18]https://wiki.php.net/rfc/attribute_amendments
[19]https://wiki.php.net/rfc/shorter_attribute_syntax
[20]https://wiki.php.net/rfc/shorter_attribute_syntax_change

Named arguments

In every programming language that has functions (that is, all of them), functions can be called by passing an ordered list of arguments to them as input parameters. That works quite well, but does have some edge cases where it's less than ideal. Specifically, when there's more than one parameter it's easy to forget what the order of them is or what a given parameter means. (And even when there's just one, it may not be self-evident at the call site what it means.)

There are various workarounds that are possible, such as passing an associative array instead of discrete parameters, but those all introduce their own problems. Passing an associative array, for instance, bypasses all type safety.

A few languages address that problem by allowing (usually optionally) callers to specify parameters by name, rather than by position. PHP 8.0 is now one of those languages.

Named parameters

In PHP, named parameters are entirely controlled by the caller. All functions and methods are, automatically, named-parameter supporting. When the function is called, the caller can choose to use a positional argument, a named argument, or even a combination of them.

As an example, PHP's `array_fill()` function produces a new array of a specified size where all elements are set to the same starting value. Like so:

```php
$new = array_fill(0, 100, 50);
```

Just from reading that … what do those numbers mean? Will `$new` be a 100 element array of `50` or a 50 element array of `100`? It's not obvious unless you know how `array_fill()` works.

With named parameters, you can call it like this instead:

```php
array_fill(start_index: 0, count: 100, value: 50);
```

Now it's clear what we'll get back: 100 array elements, with keys starting at `0`, with all values set to `50`. But you can also order the parameters however you want:

```
array_fill(
  value: 50,
  count: 100,
  start_index: 0
);
```

The argument names are matched to the parameter names in the function definition, regardless of order. In this example we've also shown the parameters in vertical form, which is fine.

Parameter names must be literals; you cannot have a variable as a parameter name.

It's also possible to specify only certain parameters by name and others positionally. More specifically, you can list parameters positionally until you switch to named arguments and then use named thereafter. Thus, this is perfectly fine:

```
array_fill(0,
  value: 50,
  count: 100
);
```

But this is not:

```
array_fill(
  value: 50,
  0,
  count: 100
);
```

Named variadics

One tricky question with named parameters is around variadics. "Variadics" are the fancy name for the ability to pass a variable number of parameters to a function. That is, you've long been able to do this in PHP:

```php
include_these($product, 1, 2, 3, 4, 5);

function include_these(Product $product, int ...$args)
{
    // $args is now an array of ints.

    $vals = ['a', 'b'];
    do_other_thing(...$vals);
}
```

Here, the ... operator, affectionately known as "splat," either collects arguments up into an array or spreads them out from an array, depending on the context. But how does that interact with named arguments?

The way they interact is, I would argue, what you'd logically expect. An indexed array that is spread out will spread out as positional arguments. An associative array that is spread out will spread out as named arguments. To continue the earlier example:

```php
// This is a named array, so the values will map to named parameters.
$params = ['count' => 100, 'start_index' => 0, 'value' => 50];
array_fill(...$params);
```

When collecting variadic arguments, they'll be collected as either numerically indexed array values if passed positionally or as string-keyed array values if passed by name. Be aware that, if you have a variadic parameter, that means you may have an array that is a mixture of numeric and named keys.

That also leads to interesting possibilities to build up a function call dynamically, by first dynamically building an associative array and then calling the function with it using splat.

```php
$args['value'] = $request->get('val') ?? 'a';
$args['start_index'] = 0;
$args['count'] = $config->getSetting('array_size');

$array = array_fill(...$args);
```

Limitations

The main pushback against named arguments was, and always has been, that making it "too easy" to have functions with lots of arguments would encourage poor API design. For example, this method call is unquestionably hard to understand:

```
read_value($object, true, false, false, false, true, true, 7, false, true);
```

Such an API really should be redesigned. The fear is that named parameters would allow API designers to get away with saying "well, just use names on them, then you can even skip all the defaults you don't care about that way." That's true, but also misses the point that the problem is the API is too complicated.

There's no strong preventative here other than "don't use this as a crutch to make bad APIs," which could be said about nearly any language feature.

But … why?

Named parameters were first seriously proposed all the way back in 2013, but didn't gain traction until now. What really brought them to the forefront again was several discussions around how to make object construction easier. Several proposals were floated in early 2020 for dedicated, one-off syntaxes for easier "struct" objects, that is, objects that are just a collection of probably-public properties.

Languages like Go and Rust make it very easy to create structs with named parameters, because they don't technically have objects the way PHP does. They have a struct with properties, period, that you can define even as a struct literal, and then you can hang methods off of them. The net result is similar but not quite the same as the Java/C++/PHP style of object. Still, plenty of people have been justifiably jealous of how easy those languages make it to create new complex structures.

None of the one-off syntaxes that were proposed really addressed the problem in a way that "fit" with PHP. Several people, however, myself included, noted that a more robust approach would be to solve the underlying problems in a way that allowed a nicer object construction syntax to just "fall out" naturally.

I laid out the argument for that in detail in a blog post[21] back in March. In short, constructor property promotion plus named arguments would, together, effectively give us a struct-initialization syntax. The whole would be greater than the sum of its parts. Fortunately, the ever-busy Nikita Popov agreed with me and picked up the ball and ran with it, implementing both RFCs. (Still, I'll claim 0.1% credit for both constructor promotion and named arguments.)

That brings us to one of the few places where named arguments really should be used: constructing struct objects.

To demonstrate, consider a value object for a PSR-7[22] URL object.

[21]https://peakd.com/php/@crell/improving-php-s-object-ergonomics
[22]https://www.php-fig.org/psr/psr-7/

```php
class Url implements UrlInterface
{
    public function __construct(
        private string $scheme = 'https',
        private ?string $authority = null,
        private ?string $userInfo = null,
        private string $host = '',
        private ?int $port = 443,
        private string $path = '',
        private ?string $query = null,
        private ?string $fragment = null
    ) {}
}
```

We have already seen how constructor promotion makes that vastly easier to write. Named arguments also makes it vastly easier to use, since many of those parameters can legitimately be absent in a valid URL, including ones early in the list like `$authority`. In PHP 7.4, you'd have to use it like so:

```php
$url = new Url('https', null, null, 'platform.sh', 443, '/blog', null, 'latest');
```

Which is … gross, but you have to do that in order to get to the later arguments. With named parameters, that collapses to this shorter and more self-documenting alternative:

```php
$url = new Url(host: 'platform.sh', path: '/blog', fragment: 'latest');

// Or if you prefer vertical:

$url = new Url(
    path: '/blog',
    host: 'platform.sh',
    fragment: 'latest'
);
```

And you can then put the parameters in any order to boot. The two features complement each other to make working with lightweight objects vastly cleaner than in previous versions. Value object constructors are, I would argue, one of the three key places to use named arguments.

Arguments in attributes

The second key target is in attributes. Recall from the last chapter:

```php
class SomeController
{
    #[Route('/path', 'action')]
    public function someAction()
    {
        // ...
    }
}
```

Attributes are syntactically an object constructor call, which gets called lazily through the reflection API as needed. Because it's a constructor call, it can do (almost) anything a constructor call can do, including use named arguments:

```php
class SomeController
{
    #[Route(path: '/path', name: 'action')]
    public function someAction()
    {
        // ...
    }
}
```

In fact, I predict that most attribute usage will use named arguments. The whole point is to make flexible metadata available within the syntax of the language itself. Many Doctrine annotation usages today rely very heavily on named keys, as they are more self-documenting and more flexible when there are many optional arguments. It's logical to expect (and encourage) attributes to follow the same pattern.

Named arguments about arguments

The third place I expect named arguments to see heavy use is on functions where the parameters are legitimately needed, and legitimately confusing, and there's no self-evident way around that. The `array_fill()` example earlier is a good example.

Another good example? Everyone's favorite canard, `$haystack, $needle` vs. `$needle, $haystack`. String functions generally use one order, array functions generally use the other, for reasons that make sense when you're modeling your API on C but not otherwise. Now, you don't need to remember what the order is.

```
if (in_array(haystack: $arr, needle: 'A')) {
    // ...
}
```

Is that the order those parameters are in the function, positionally? Who cares? The function call specifies exactly what is being passed, making the order irrelevant. (They're not in order, in case you were wondering.)

At last, we can stop complaining about parameter order and citing that as a reason why PHP is bad. (I'm sure people will still continue to do so; they'll just be even more wrong than they already were.)

API implications

One warning, however. As noted, named arguments work for all functions and methods automatically. That means the parameter names in functions, methods, and interfaces are now API significant. Changing them may break users who are calling them by name. (This would be a good time to revisit your parameter names, while checking to make sure your libraries are ready for PHP 8.0.)

Variable name significance is not, at this point, enforced in inheritance. That's a pragmatic decision to avoid breaking too much existing code, as the vast majority of methods won't be called with named parameters 99.9% of the time, so creating even more breakage for existing code that might be renaming arguments for entirely logical reasons didn't make sense. Python and Ruby take the same approach and haven't run into serious issues, which PHP took as a good sign that it was safe to do the same.

As mentioned, the named arguments RFC[23] come to us courtesy of Nikita Popov.

[23]https://wiki.php.net/rfc/named_params

Feature improvements

While the headline features will garner the most press, there are many other improvements to PHP 8.0 that will make the life of the daily developer easier. Each individually won't define a release, but taken together they add up to a substantially nicer language to work with.

Union types

The biggest type system change is the introduction of *union types*. *Union types* are a virtual type that are the "union" (a logical "OR") of two other types. For example:

```php
function mangleUsers(string|array $users): array
{
    If (is_string($users)) {
        $users = [$users];
    }
    // ...
}
```

Previously, if you wanted to support the "one or many" parameter style like that you couldn't type the parameter. Now, you can specify `string|array` and either a string or array variable is acceptable. Integers, objects, and so on will still be rejected.

Union types work on parameters, return types, and property types, and they support any type PHP supports: primitives, objects, user-defined classes, etc. While they can be used to produce all kinds of grody, inconsistent APIs, those were already possible. At least now they can be documented. More importantly, it opens up some interesting use cases that were previously impossible to document in the type system itself.

For example, a function that requires a number but can work on both ints or floats can now say exactly that:

```php
function doMath(int|float): int|float
{
    // ...
}
```

It's also possible to explicitly say that a function or method may return objects of different types. For example, a function that takes a PSR-7[24] request object and either returns a new request or a corresponding response would look like this:

[24]https://www.php-fig.org/psr/psr-7/

```php
function handleRequest(RequestInterface $req): RequestInterface|ResponseInterface
{
    // ...
}
```

In inheritance, union types follow the same covariant/contravariant rules as any other type. That is, a subclass's method parameters are allowed to accept a broader type definition than its parent (so adding a `|Foo` is OK, but not removing it), while a return value is allowed to specify a narrower type definition than its parent (so removing `|Foo` is OK, but not adding it).

The reflection API has also been updated to support inspecting union types on functions and properties.

More details are available in the original RFC[25]. Thanks to Nikita Popov for this one.

Specialty unions

PHP already has a number of one-off union types defined in the language. `iterable`, for instance, is effectively a union type for `array|\Traversable`. Nullable types, such as `?Product`, are essentially a union type of `null|Product`.

In fact, it is now possible to specify `null` as a type in a union, but only in a union. So `null|RequestInterface|Respons` is a legal type definition. The `?` nullable flag is now a shorthand for `null|` when there's only a single other value allowed.

Another union-type-only option is `false`. That is mainly to support existing PHP internal functions that, due to errors of judgement made back in the 1990s, return "a value, or false on error." Those functions can now be typed as:

```php
function base64_decode(string $str, bool $strict=false): string|false {}
```

That capability is only to support legacy code that can't change its API to be less bad. Please don't use it for any new code. Like `null`, `false` cannot be used as a stand-alone type declaration. Also, the `void` type declaration cannot be combined with anything, as it literally means "nothing at all is returned, period," so it makes little sense to combine with other types.

PHP 8 also introduces a new built-in union type. The new `mixed` type is equivalent to `array|bool|callable|int|floa` That's not *quite* the same as omitting the type entirely. Omitting the type could mean that the developer just forgot, or that the type system is not robust enough to describe the possible allowed values. Using the `mixed` type explicitly says to the engine and other developers, "I accept anything here, and I mean it." There are now extremely few cases where it's not possible to type a parameter, return, or property in PHP 8.

Thanks to Máté Kocsis and Dan Ackroid for this RFC[26].

[25] https://wiki.php.net/rfc/union_types_v2
[26] https://wiki.php.net/rfc/mixed_type_v2

The Stringable interface

PHP objects have long been able to define a way that they can be cast to a string, using the `__toString()` magic method[27]. However, the `string` type hint doesn't accept string-castable objects in strict mode, rendering that less useful since PHP 7.0 introduced scalar types.

PHP 8 now introduces a `Stringable` interface that corresponds to an object having the `__toString()` magic method. It also does so in a clever, backward-compatible way.

As of 8.0, a class *may* implement a `Stringable` interface that defines a method `public function __toString(): string`. If it doesn't, but still implements that method, *the engine will add the method and return type automatically*. That means any stringable object may now be type-checked, including as part of a union type, like so:

```php
function show_message(string|Stringable $message): void
{
    // ...
}
```

Boom! `__toString` is useful again.

Because the new interface is, technically, slightly more strict than the original `__toString()` method (since it enforces a string return rather than just assuming it), the Symfony team has included a polyfill for it in their symfony/polyfill-php80[28] package. That allows developers to start using it now to ensure their types are correct and will be forward-compatible for PHP 8 immediately.

Mind you, just because `__toString()` is more usable now doesn't mean you should go overboard. Most objects should not have a `__toString()` method and should have meaningful methods that return strings instead. Where `__toString()` is useful is for value objects that have one and only one possible meaningful string representation, because the object is essentially just a fancy string. A good example is an `IPv4Address` object, which would only make sense to cast to a string as `1.2.3.4` type strings.

Thanks to Nicolas Grekas for the Stringable RFC[29].

[27] https://www.php.net/manual/en/language.oop5.magic.php#object.tostring
[28] https://github.com/symfony/polyfill
[29] https://wiki.php.net/rfc/stringable

`match()` expressions

PHP has had a `switch` statement since the dawn of time, modeled on the same construct in C. You've surely seen it before:

```php
switch ($var) {
    case 'a':
        $message = "The variable was a.";
        break;
    case 'b':
        $message = "The variable was c.";
        break;
    case 'c':
        $message = "The variable was c.";
        break;
    default:
        $message = "The variable was something else.";
        break;
}

print $message;
```

While it gets the job done, the classic switch has a number of limitations. Most newer languages like Go or Rust have opted for more robust, less error-prone structures that go by a variety of names. And now, PHP has one, too.

Enter `match`. Unlike `switch`, `match` is an expression. That means it evaluates to a value, like so:

```php
$message = match($var) {
    'a' => 'The variable was a',
    'b' => 'The variable was b',
    'c' => 'The variable was c',
    default => 'The variable was something else',
};

print $message;
```

There are several things to point out here:

1. With `switch`, each `case` is compared with loose equality ==. With `match`, each branch is compared with strict equality, ===. That means the types must match, too.
2. Each branch of the `match` is a single expression only. No multi-line statements, just a single expression that gets evaluated. If you need complex logic, make it a function or method that you call.
3. There is no fall through from one branch to the next, so there is no need for an explicit `break`.
4. `match` returns a value. That's its whole reason to exist.
5. Because it's an expression, a closing `;` is needed at the very end, just like with closure definitions.
6. `match` is exhaustive. If `$var` is not === any of the provided values and there is no `default`, an error will be thrown.

`match` branches can also be compound and comma-delimited for "OR" like behavior, like so:

```php
echo match($operator) {
    '+', '-', '*', '/' => 'Basic arithmetic',
    '%' => 'Modulus',
    '!' => 'Negation',
};
```

`match` was pitched as a modernized, more useful `switch`, but I am not sure that's accurate. Rather, I think of `match` as a more powerful ternary.

In the past, I've often found myself in binary cases where a variable can be assigned to one of two values depending on some condition. The easiest way to write that, I've found, is like this:

```php
$display = $user->isAdmin()
    ? $user->name() . ' (admin)'
    : $user->name() . ' (muggle)'
;
```

That's completely valid code and quite useful. If the logic in the condition or in either of the branches is too complex to be easily readable, that's a hint the logic should be refactored out to its own function or method. I've found that to be a really good heuristic for situations like this, as well as leading to very readable, compact, and testable code.

That only works for `true`/`false`, however. `match`, in contrast, works for any number of options but has the same incentives to nudge you toward well-factored, clean code.

If you need more complex conditions than simple identity, you can use `match` on `true`:

```
$count = get_count();
$size = match(true) {
    $count > 0 && $count <=10 => 'small',
    $count <=50 => 'medium',
    $count >50 => 'huge',
};
```

Note that since there is no `default`, a non-match (in this case if `$count` is `0` or negative) will result in a thrown error rather than silently assigning to null and moving on. That's good, because errors can't propagate to pollute later code.

The match expression RFC[30] comes to us courtesy of Ilija Tovilo.

[30]https://wiki.php.net/rfc/match_expression_v2

Trailing commas in function definitions and closures

A long time ago on a mailing list far far away (that is, back in 2017), there was a proposal to allow a trailing comma[31] in all places where PHP supported a list of values. That, unfortunately, mostly failed, but subsequent RFCs to add comma support to each list construct independently have passed. Go figure.

The latest additions for PHP 8 are in function definitions and closures. For function definitions, it's now possible to put a comma after the last parameter definition, which is mainly useful when many parameters are split across several lines.

```php
Class Address
{
    public function __construct(
        string $street,
        string $number,
        string $city,
        string $state,
        string $zip,     // This is new.
    ) {
    // ...
    }
}
```

Trailing commas in function calls were already added in PHP 7.3.

This update[32] is thanks to the ever-popular Nikita Popov. It was added around the same time as Constructor Promotion, as the combination of Constructor Promotion and Named Arguments is likely to result in a lot more vertical lists of parameters and arguments. To repeat the example from an earlier chapter:

[31] https://wiki.php.net/rfc/list-syntax-trailing-commas
[32] https://wiki.php.net/rfc/trailing_comma_in_parameter_list

```php
class Url implements UrlInterface
{
    public function __construct(
        private string $scheme = 'https',
        private ?string $authority = null,
        private ?string $userInfo = null,
        private string $host = '',
        private ?int $port = 443,
        private string $path = '',
        private ?string $query = null,
        private ?string $fragment = null,
    ) {}
}

$url = new Url(
    path: '/blog',
    host: 'platform.sh',
    fragment: 'latest',
);
```

The commas on the last items just make the code a little more consistent and easier to maintain.

Trailing commas are now also allowed in `use` clauses in closures, for the same reason:

```php
$a ='A';
$b = 'B';
$c = 'C';

$dynamic_function = function(
  $first,
  $second,
  $third,       // This is new, as above.
) use (
  $a,
  $b,
  $c,             // This is also new.
);
```

This change comes courtesy[33] of Tyson Andre.

[33]https://wiki.php.net/rfc/trailing_comma_in_closure_use_list

Weak Maps

PHP 7.4 introduced the concept of Weak References[34], which allow an object to be referenced without incrementing its reference counter. That's a bit obscure and in practice not all that useful in most cases. What we were really waiting for is Weak Maps, which have landed in PHP 8.0.

Weak Maps are a little nerdy to explain, so bear with me. Normally, when you create an object and assign it to a variable, what happens is the object is created in memory and then the variable is created as a *reference* to it. Think of it as the variable just having the ID of the object, not the object itself. If you assign another variable to the same object, there's still only one object, but now there are two variables with the object's ID.

```php
$a = new Foo();
$b = $a;

// $b and $a are now separate variables that
// both point to a Foo object in memory somewhere.
```

Every time a variable is removed, PHP checks to see if there are any other variables still referencing that object. If there are none, it knows it's safe to delete that object for you. This process is called "garbage collection," and I've greatly over-simplified it here because this is enough for our purposes.

A Weak Reference, or a Weak Map, is a way of creating a variable that acts like any other, but when PHP checks to see if any variables still point to an object those "weak" variables don't count. So if there are still three weak references pointing at an object, but no normal variables, PHP will happily delete the object and set the remaining variables to null instead.

It will make more sense to see it in action. Suppose you have a series of `Product` objects, written by someone else in a library of some kind. You can't modify them, but you're going to use them.

For each `Product`, you want to track additional information that isn't on the original object, say, a list of `Review` objects on that product. Subclassing the `Product` to include reviews is possible but messy, and inheritance often runs into problems when trying to combine multiple modules together.

Instead, we'll make a separate `ReviewList` object that contains a Weak Map of `Review` objects, lazy-loading them as needed and tracking them by `Product`. Once a `Product` is removed from memory, when all of the variables that reference it go out of scope, we don't need to keep those `Review` objects around. A `WeakMap` acts as a self-cleaning cache in this case, and it works similarly to `ArrayObject`.

[34] https://www.php.net/manual/en/class.weakreference.php

```php
class ReviewList
{
    private WeakMap $cache;

    public function __construct()
    {
        $this->cache = new WeakMap();
    }

    public function getReviews(Product $prod): string
    {
        return $this->cache[$prod] ??= $this->findReviews($prod->id());
    }

    protected function findReviews(int $prodId): array
    {
        // ...
    }
}


$reviewList = new ReviewList();
$prod1 = getProduct(1);
$prod2 = getProduct(2);

$reviews_p1 = $reviewList->getReviews($prod1);
$reviews_p2 = $reviewList->getReviews($prod2);

// ...

$reviews_p1_again = $reviewList->getReviews($prod1);

unset($prod1);
```

In this example, `ReviewList` has an internal "weak cache" keyed off of `Product` objects. When `getReviews()` is called, if the desired value is already in the cache it will get returned. If not, it will be loaded into memory, saved in the `WeakMap` cache, and then returned. (The `??=` bit there is null-coalesce-assign, introduced in PHP 7.4, and is the bee's knees for exactly this sort of case.) Later on, when we create `$reviews_p1_again`, the value will be looked up in the cache instead.

However, at some point in the future we unset `$prod1`. Generally that won't be done manually, but the variable will go out of scope and get garbage collected. Since there are no more normal references to the product 1 object, the reference to that object in the `$cache` Weak Map will get removed automatically. That will also cause the corresponding list of `Review` objects to get cleared automatically, too. The memory is saved and no further work is needed. It "Just Works(tm)."

If you tried to do the same with a normal array, there would be two problems:

1. Arrays can't use objects as keys, so it would need to be keyed off of the product ID or similar.
2. That means the cache wouldn't know to prune itself when the object it's a cache for gets garbage collected. You might be able to implement some complex logic using destructors, global variables, and other black magic, but … please don't. The odds of getting it wrong are high, and the level of complexity it introduces isn't worth it.

Caching scenarios like that are really the only strong use case for `WeakMap`, but when you need it, it's a big memory and code saver.

Once again, we have Nikita Popov to thank for the WeakMap RFC[35].

[35] https://wiki.php.net/rfc/weak_maps

New string functions

PHP has a huge array of string functions (sorry, too easy), from the very useful to the rather obscure. PHP 8.0 brings three new functions to the table, all of which replace common past idioms.

`str_starts_with()` and `str_ends_with()` came as a matched set and do exactly what they say on the tin. Both take a string to check and a partial string to test and return a boolean if the first string begins with (or ends with) the second.

```php
$needle = 'hello';
if (str_starts_with('hello world', $needle)) {
    print "This must be your first program.";
}
```

Both functions do an exact match, in ASCII, and have no case-insensitive option as that was deemed to complicate it too much. If you need a case-insensitive check, run `strtolower()` on both strings first. These functions come to us courtesy of an RFC[36] by Will Hudgins.

In a similar vein, `str_contains()` is also fairly self-explanatory. It checks if the first string contains the second string anywhere within it at all.

```php
$needle = 'on';
if (str_contains('peace on earth', $needle)) {
    print "Yes, let's do that.";
}
```

As with the earlier functions, `str_contains()` is a case-sensitive ASCII comparison. The RFC[37] for this addition comes from Philipp Tanlak.

All three functions were possible to mimic in user-space via clever uses of `strpos()`, where "clever" is a polite way of saying "error-prone." One of the most common newbie PHP developer mistakes (made by many very experienced developers) was to do a `str_contains()` type check like so:

```php
$its_in_there = strpos($haystack, $needle) == false;
```

However, due to PHP's type juggling, that basic-equality (==) check is insufficient, because a position of 0 (meaning the `$needle` is the first part of `$haystack`) would be == `false`. The check needs to be identity (===), which is easy to forget. It's also a perfect example of why "return a value or false on error" is a terrible pattern that should never be used, ever.

Fortunately, with PHP 8.0 all of that cleverness is no longer needed, as the trio of new functions type-safely return the boolean value you wanted in the first place.

[36] https://wiki.php.net/rfc/add_str_starts_with_and_ends_with_functions
[37] https://wiki.php.net/rfc/str_contains

Token objects

PHP's `token_get_all()`[38] function lets PHP parse PHP code files and get back an array of arrays, representing the stream of tokens in that file. However, a giant array of arrays is not always the most useful way to represent, well, anything. For one, PHP arrays are very memory hungry[39]. For another, the nested arrays are positional, and the meaning of each position varies with the type of token. That architecture pattern is known in academic literature as "fugly."

PHP 8.0 introduces a new class, `PhpToken`, with a corresponding `tokenize()` static method. `PhpToken::tokenize()` is passed a string of PHP code and returns an array of `PhpToken` objects. Because it's using objects, this method is faster, uses half as much memory, and is easier to use than `token_get_all()`. (Tastes great *and* is less filling!)

`PhpToken` also includes a few utility methods to get information about the token, such as `getTokenName()` or `isIgnorable()`. (The documentation has more details[40].) There's even a `__toString()` implementation that renders the object back to its source code string form. What's more, it's also possible to extend the class and add your own custom methods.

```php
$code = '...'; // Some PHP source code.

class LowerCaseToken extends PhpToken implements Stringable {
    public function getLowerText() {
        return strtolower($this->text);
    }

    public function __toString(): string {
        return $this->getLowerText();
    }
}

$tokens = LowerCaseToken::tokenize($code);

print implode($tokens);
```

In this (extremely contrived) example, `LowerCaseToken::tokenize($code);` returns an array of `LowerCaseToken` objects. `implode()` then converts each to a string and concatenates them, with no extra separator. "Converts to a string" means that `__toString()` is called, which returns a lower-case

[38] https://www.php.net/manual/en/function.token-get-all.php
[39] https://peakd.com/php/@crell/php-use-associative-arrays-basically-never
[40] https://www.php.net/manual/en/class.phptoken.php

version of the string representation of the token. The net result is to take a piece of code and force it to all lowercase without otherwise modifying the code or its structure.

This is admittedly not the most practical example, but one could imagine reading in a file of PHP source code, modifying the array in some structured way, and then trivially serializing it back out in modified form, while still respecting the structure and alignment of the text. Alternatively, one could add additional methods to help with advanced parsing for code formatting, static analysis, or other meta-coding tasks.

It should be no surprise that this RFC[41] comes to us courtesy of Nikita Popov. Again.

(Note that in the RFC, `tokenize()` is called `getAll()`. It was changed to `tokenize()` very late in the RC phase without a follow-up RFC, as the original name didn't really make much sense.)

[41]https://wiki.php.net/rfc/token_as_object

Variable dereferencing

PHP 7.0 included a change called "Uniform Variable Syntax," which standardized the way PHP interpreted complex variable expressions. It was a good change overall, but there were a few gaps missed. PHP 8.0 fixes most of those. Technically these are extreme edge cases, and some of them aren't even meaningful without certain obscure extensions, but someone (we won't say who, because you know who) has been on a consistency kick in PHP for some time and decided they needed to be polished off.

First, in PHP 7.4 `"foo"[0]` is legal syntax and will return "f". However, `"foo$bar"[0]` is not, even though it's also a string that's being dereferenced to return its first character. In PHP 8.0, that now works.

The same is true of magic constants. `FOO[0]` is valid, but `__FUNCTION__[0]` is not. There's no good reason for that, so in PHP 8.0 the latter now works, too.

To be even more obscure, both of the above points apply to dereferencing a string or constant with `->` as well. Since `->` applies to objects, which strings and constants are not, that doesn't mean anything in practice. For consistency, though, `"foo"->bar()` will now parse as valid and return a "calling method on non-object" error instead of a parse error. Because why not?

PHP also supports a `{}` dereferencing syntax in addition to `[]`. They mean the same thing when used on arrays and strings, even if almost no one uses them. At least, that was true for everything except constants. As of 8.0, that's true of constants as well.

Even more obscurely, literal class constants do not support dereferencing static variables off of them, but dynamic class constants do. That is, `Foo::$bar::$baz` is legal but `Foo::BAR::$baz` is not in PHP 7.4. In PHP 8.0, they are both legal, even if the odds of you running into it are vanishingly small.

You'll likely never run into any of the above situations. One you may run into, however, is expressions for `instanceof` and `new`. Often it's useful to create a new object of a dynamically named class or check an object against a dynamic class name. That is:

```php
$o = new $class();
if ($o instanceof $class) { ... }
```

However, there was no way to build the class name dynamically in place. It had to be assigned to a temp variable in advance. Something like this wasn't allowed before:

```php
$o = new ("Driver_" . $driver_name);
if ($o instanceof ('Driver\\' . $driver_name)) { ... }
```

As of PHP 8.0, those are now legal.

To the surprise of no one, these changes[42] all come courtesy of Nikita Popov.

[42]https://wiki.php.net/rfc/variable_syntax_tweaks

Type-respecting variadics

PHP has supported variadic arguments since PHP 5.6. However, they were rather picky about types. Most of the time that was fine, as variadic arguments would almost always be of the same type. For example:

```php
// Untyped variadic.
function add_to_list(List $l, ...$to_add) {
  // ...
}

// Typed variadic.
function add_to_list(List $l, ...Item $to_add) {
  // ...
}
```

One place that is not necessarily the case is in inheritance. If a parent method listed out parameters explicitly, it was not permitted for a child method to use a variadic instead. That includes interfaces. That has been fixed in PHP 8.0, meaning this code is now allowed:

```php
interface ThingDoer
{
    public function doStuff(int $i, string $s);
}

class DoThing implements ThingDoer
{
    public function doStuff(int $i, string $s)
    {
        print "Ran" . PHP_EOL;
    }
}

class ThingLogger implements ThingDoer
{
    public function __construct(private ThingDoer $do) {}

    public function doStuff(...$args)
    {
        print("Stuff is getting done.\n");
        $this->do->doStuff(...$args);
    }
}
```

This change is mainly to support this sort of “pass through” cases, where one method delegates all of its parameters to another function with the same signature.

There was no RFC for this feature, but Nikita Popov was responsible for it anyway. Shocking, I know.

Safe division

We all know from math(s) class that division by zero makes no sense at all, and you shouldn't do it. What should happen if you try to do it anyway is a subject of slightly more debate.

In PHP 7, `5 / 0` results in a fatal error. That is not always ideal, especially when dealing with user-supplied data. (Fun fact: Users are the #1 source of invalid data.) There is, in fact, an established standard for what division by zero should do in programming languages called IEEE 754[43]. Specifically, it should return positive or negative infinity, which is (approximately) what mathematically division by zero means.

PHP 8.0 introduces the `fdiv()` function that does exactly that. It takes two parameters and returns the first divided by the second, as you'd expect. However, if the denominator (second parameter) is `0` then it returns `INF`, `-INF`, or `NAN` as sentinel values rather than killing the program. As an extra special case, zero divided by zero isn't a number, and it doesn't even try.

```
INF == fdiv(5, 0); // true
-INF == fdiv(-5/0); // true
NAN == fdiv(0, 0); // true
```

`INF` and `NAN` are constants for floating point numbers, but have no numeric representation (being that they're infinite and all). If you print them, you'll just get back their names.

In those cases where you're dividing by a user-supplied value, it's safer now to use `fdiv()` than the division operator. The output of `fdiv()` is also safe to print, as long as you're OK with `INF` in user output.

One warning, though: As is traditional, `NAN`, being not a number, is not equal to anything, including itself. That does make checking against it a bit more difficult, but at least that's consistent across languages.

`fdiv()` also snuck in without an RFC courtesy of everyone's favorite Nikita Popov.

Meanwhile, dividing by zero with `/` has been promoted to throw a `DivisionByZeroError`, making it slightly less fatal than before.

[43]https://en.wikipedia.org/wiki/IEEE_754

Class and object improvements

Classes and objects are the lifeblood of most PHP applications today. While PHP 8.0 doesn't fundamentally change any part of the way objects work, it does contain a number of incremental improvements to make OOP code more consistent and predictable.

Object class constant

Objects now have a magic constant that specifies their class, just as class names do. `$object::class` is a string containing the class name, such as `App\Form\FormDef`. It's the same as `get_class()`, but easier to use.

Credit for this feature[44] goes again to Nikita Popov.

Static return types

A personal favorite feature of mine, methods can now have a return type of `static`. That's mainly useful for interfaces with chained methods. That means the following is now possible:

```php
Interface TaskBuilder
{
    public function addStep(Step $s): static;
}

class  ImportantTask implements TaskBuilder
{
    public function addStep(Step $s): static
    {
        $this->steps[] = $s;
        return $this;
    }
}
```

And now `ImportantTask::addStep()` is typed to return an instance of `ImportantTask`. Previously with a return type of `self` it would only indicate that it's returning `TaskBuilder`.

Credit for this improvement[45] goes yet again to Nikita Popov.

[44]https://wiki.php.net/rfc/class_name_literal_on_object
[45]https://wiki.php.net/rfc/static_return_type

Private inheritance

(Insert obvious legal jokes here.)

In prior versions of PHP, private methods are not inherited by a child class, as one would expect. However, if the child class contains a method of the same name as the private method in the parent, it still tries to enforce some inheritance rules, such as visibility, but not others, like parameter order.

PHP 8.0 makes that more consistent by freeing child methods of any constraints imposed by their parent if there's a matching private method. In particular, that means a `final private` method can be overridden in a child, because … it's not really being overridden, as it was never inherited anyway. That also means that a `final private` method makes no actual sense and will now trigger a Warning.

Or in code:

```
class Ancestor
{
    pubilc function doAThing()
    {
        return $this->aPrivateMethod();
    }

    final private function aPrivateMethod()
    {
        return "Ancestor";
    }
}

class Child
{
    pubilc function doAThing()
    {
        return $this->aPrivateMethod();
    }

    private function aPrivateMethod()
    {
        return "Child";
    }
}

$c = new Child();
print $c->doAThing();
```

This code in PHP 7 would have produced a fatal error. In PHP 8.0, it will issue a Warning that the `final private` in the `Ancestor` class is pointless, and then dutifully print “Child” without further complaint.

The one exception is the `__construct()` method. That can still be `final private`, because constructors are weird and it’s the only reasonable way to forbid a child class from having a constructor at all. (There are some edge cases where that may be useful.)

It’s unlikely that you’ll ever run into this change, but if you do, you can thank Pedro Magalhães for this RFC[46].

Trait method validation

Traits are a PHP feature to circumvent the lack of multiple inheritance in the language. They are implemented as a “compile-time copy and paste” and allow you to provide methods and properties that get replicated into a class. They may also include abstract methods, the same way abstract classes can, but their interaction with inheritance has been … buggy and quirky and inconsistent. PHP 8.0 makes them less buggy and quirky.

More specifically, it used to be that an abstract method in a trait did not require a `use`ing class (one that imports the trait) to match that abstract method’s signature, only its name. However, it did require the method to match the trait’s abstract method if a child of the `use`ing class or a parent of the `use`ing class implemented the method. If that doesn’t make sense to you, that’s because it doesn’t make sense. The old behavior was a bug and has now been fixed. That means the following is now syntactically invalid, instead of just making no sense:

```php
trait HashUtil
{
    public function compoundHash(string $a, string $b): string
    {
        return $this->innerHash($a) . $this->innerHash($b);
    }

    abstract protected function innerHash(string $a): string;
}

class Encryptor
{
    use HashUtil;

    // This doesn't match the signature of HashUtil::innerHash(),
    // but PHP 7.4 wouldn't object to that. PHP 8.0 does.
```

[46]https://wiki.php.net/rfc/inheritance_private_methods

```
    protected function innerHash(int $a): int
    {
        // ...
    }
}
```

Moreover, it’s now possible to have an `abstract private` method in a trait. Normally `abstract private` methods wouldn’t make any sense at all, as they have no implementation and you cannot add one. Traits, however, operate sideways to inheritance, so the example above could now be written as:

```
trait HashUtil
{
    public function compoundHash(string $a, string $b): string
    {
        return $this->innerHash($a) . $this->innerHash($b);
    }

    abstract private function innerHash(string $a): string;
}
```

That means any class that `uses` that trait gets access to the `compoundHash()` method, but has to provide its own private implementation of `innerHash()`.

As this is a “cleanup and orderly code” RFC[47], you probably already guessed that it came from Nikita Popov.

[47]https://wiki.php.net/rfc/abstract_trait_method_validation

Date and time handling

Dating is hard, but PHP makes it bearable. PHP's `DateTime` extension is one of the better date-and-time handling libraries available in any language. In PHP 8.0, it got a wee bit better, too.

`p` date formatter

PHP has a variety of functions that accept a date format string to turn a `DateTime` object (or similar) into one of the 14 million possible ways to express dates and times in text. PHP 8.0 adds another formating code, `p`.

The `P` code, which already exists, shows the time offset of a given `DateTime` instance. For example:

```php
$d = new DateTimeImmutable('now', new DateTimeZone('America/Chicago'));
print $d->format('P');
// Prints "-06:00"
```

In UTC time, `P` would print "+00:00".

The new `p` formatting code works exactly the same, but represents UTC time as `Z` instead. (Both are valid ISO 8601 ways to represent that timezone.) For example:

```php
$d = new DateTimeImmutable('now', new DateTimeZone('UTC'));
print $d->format('P');
// Prints "+00:00"
print $d->format('p');
// Prints "Z"
```

This feature snuck in the back door without an invitation or RFC, but brought punch so was allowed to stay anyway.

Describing a time with an offset is still an incorrect way to record time, because it is not consistent. Many areas are not on one-hour offsets, daylight saving time means they sometimes change throughout the year, timezones change when governments feel like it, two areas can have the same offset part of the year but not others, etc. The only correct way to record a time is as a timestamp with region code, like "America/Chicago" or "Europe/Paris", for which the formatting code is `e`. All other answers are wrong.

`DateTimeInterface::createFromInterface()`

This is another "gap filler" improvement. PHP's Date and Time handling is one of the better implementations of any language, but for historical reasons comes in two variants: `DateTime` objects and `DateTimeImmutable` objects. The latter is, unsurprisingly, immutable, while the former is mutable. In a just world only the latter would exist, but the former was implemented first before the value of immutable objects really caught on in PHP land. In order to allow developers to support both, there is also a `DateTimeInterface` interface that, while not extensive, covers both objects and the read-only methods they have in common.

`DateTime` has a `createFromImmutable()` static method that takes an immutable object and returns a new mutable `DateTime`. `DateTimeImmuable`, meanwhile, has a `createFromMutable()` static method that takes a mutable object and returns a new immutable `DateTimeImmutable`. These are both useful, but seem to forget that the interface exists.

Enter PHP 8.0, and both classes now have a new `createFromInterface()` method that will accept either version of the object. It's a little thing, but it removes a conditional from everyone's code. So, for example, if you want to ensure that you have an immutable object to work with you can do the following:

```
function do_date_stuff(DateTimeInterface $d)
{
    $date = DateTimeImmutable::createFromInterface($d);
    // ...
}
```

Now, you can accept either `DateTime` or `DateTimeImmutable` but guarantee that you are working with the immutable version for the rest of the function. Which you should always do, because `DateTimeImmutable` is superior in all circumstances and `DateTime` should not be used, ever.

This change also slipped in below the bar needed for an RFC, but can be credited to Mike Simonson.

New formatting options for Intl extension

Another edge-case-y addition, the Intl extension now has four new formatting constants:

- `IntlDateFormatter::RELATIVE_FULL`
- `IntlDateFormatter::RELATIVE_LONG`
- `IntlDateFormatter::RELATIVE_MEDIUM`
- `IntlDateFormatter::RELATIVE_SHORT`

They work the same as the existing `FULL`, `LONG`, `MEDIUM`, and `SHORT` constants, except for three days. Today, tomorrow, and yesterday will get formatted as `today`, `tomorrow`, and `yesterday`, respectively. For example:

```php
function printFormat(
    int $dateFormat,
    int $timeFormat,
    DateTimeImmutable $time,
) {
    $formatter = new IntlDateFormatter(
        "en_US",
        $dateFormat,
        $timeFormat,
        "America/Los_Angeles",
        IntlDateFormatter::GREGORIAN,
    );

    echo $formatter->format($time) . "\n";
}

printFormat(
    IntlDateFormatter::RELATIVE_FULL,
    IntlDateFormatter::NONE,
    new DateTimeImmutable('-1 day'),
);
```

That will print “yesterday”. If the timeframe given was `-2 days`, then it would print the same output as `IntlDateFormatter::FULL`.

This stealth change was snuck in by Máté Kocsis.

Locale improvements

Locales are weird and annoying. There, I said it.

For those who haven't had to deal with them (mainly English speakers), locales are a concept inherited from C to, somewhat, localize the programming language itself. It changes the way, for example, character sorting works to match different languages and character sets. As is common with older APIs inherited conceptually from C, it's largely driven by a global setting[48].

A few small changes make locales more flexible and less aggressive in PHP 8.0.

Locale-independent float casting

One of the many things locales influence is decimal separators. For "historical reasons," some parts of the world use a `.` as a decimal separator, like `3.14159`. Others use a `,` separator, like `3,14159`. Computers internally are virtually all programmed to use `.`, which is why that's what you're used to when writing code. (A case of decimal imperialism.) The locale setting, however, determines which punctuation is used when converting a floating-point number to a string.

That creates an interesting problem, however, because it's not entirely consistent. Some APIs, like PDO or `json_encode()`, already don't use commas, ever, for compatibility with external systems. Also, consider casting a float to a string and back again. With a locale set to a region that uses comma-decimals, `3.14159` would cast to `"3,14159"`, but then casting that string back to a float would yield only `3`, because the string->float cast is not affected by the locale. That's … wrong.

For PHP 8.0, float casting has been removed from the locale configuration. That means `(string)3.14159` will yield a string with a `.` in it, not a `,`, regardless of the locale setting.

That may cause slight presentation changes in applications that expected to print `3,14159` and will now show it with a period. However, the code accuracy is now more reliable, and that's more important.

If you still want to display comma-decimals, you have three options:

- The `number_format()` function lets you explicitly specify the decimal separator to be any character you want, as well as the thousands separator.
- The `NumberFormatter` class (part of the intl extension) offers the same customizability.
- `printf()`, `sprintf()`, and related functions already have separate placeholders for locale-sensitive (`%f`) and not-locale-sensitive (`%F`) float formatting. Those have been left unchanged, so `printf('%.2f', 3.14159)` while in a German locale will still yield `3,14`.

This bit of confusion-squishing[49] comes to us courtesy of George Peter Banyard and Máté Kocsis.

[48]http://www.php.net/setlocale
[49]https://wiki.php.net/rfc/locale_independent_float_to_string

New `printf()` formatting codes

The `printf()` function (and its relatives `sprintf()` and `fprintf()` and assorted others) formats values according to a pattern string, which can include both literal text and placeholders. Those placeholders can be simple (like `%s` for “put string here”) or robust and context-aware layout instructions.

In particular, the `%g` and `%G` formatting codes format a floating-point number, sometimes using scientific notation if appropriate. (The only difference is the case of the `e`/`E` that gets used.) However, those codes are locale-aware, and so will sometimes produce a `.` for the decimal part and sometimes a `,`, depending on the locale.

PHP `8.0` introduces `%h` and `%H`, which work the exact same way but are not locale-sensitive. They will always produce `.`-decimal floats.

Error handling improvements

Your code has errors. Everyone's code has errors. That's what it means to be code.

Handling errors should be as straightforward as possible, however. PHP 8.0 introduces a number of new features to help with Error Experience (EX), both in making them and understanding them.

`throw` expressions

PHP supports exceptions via the `throw` keyword. For various internal implementation reasons, `throw` was implemented as a *statement*, meaning it didn't return a value and just "was a thing."

In PHP 8, `throw` was converted to an *expression*, meaning it technically can return a value. That's not all that useful (`throw` breaks the current line of execution, by design), but it means `throw` can now be used in expression places in code.

The best example of where that's useful are ternaries, null coalesce, and other shortened expression symbols. Consider:

```php
// $value is non-nullable.
$value = $nullableValue ?? throw new BadData();

// $value is truthy.
$value = $falsableValue ?: throw new BadData();
```

In PHP 7, those are syntax errors. In PHP 8.0, they do what you'd expect: assign the value if possible or throw if the condition fails.

An even more fun combination is mixing it with the new match() expression.

```php
$value = match($size) {
  0 => throw new ThatsNotASize(),
  1 => throw new ThatsTooSmall(),
  3, 4 => 'small',
  5, 6 => 'medium',
  default => 'large',
};
```

`throw` expressions come to us[50] courtesy of Ilija Tovilo.

[50]https://wiki.php.net/rfc/throw_expression

Non-capturing `catch`

Speaking of exceptions, there are cases where you want to catch an exception, but what you'll do in response doesn't depend on the exception data itself. You can now catch an exception without assigning it to a variable, like so:

```
try {
    changeImportantData();
} catch (PermissionException) {
    echo "You don't have permission to do this";
}
```

That `catch` will trigger for `PermissionException` exceptions, but the exception itself won't be assigned to a value.

Max Semenik was responsible for the contents of this RFC[51].

`get_debug_type()`

For the debuggers only, PHP 8.0 includes a new function `get_debug_type()`. It's similar to `gettype()`, in that it returns the type of a variable, but it's far more useful. Specifically:

- On integers and booleans, `gettype()` returns "integer" and "boolean". `get_debug_type()` returns `int` and `bool`, which are the actual types used in in the language.
- `gettype()` would call `3.14` a "double", which is the C name for it. `get_debug_type()` calls it `float`, which is what PHP syntax calls it.
- `gettype()` calls all objects `object`. `get_debug_type()` will return the class name of the object or `class@anonymous` for an anonymous class.
- `gettype()` calls all resources `resource`. `get_debug_type()` differentiates between open and closed resources. (You shouldn't be using resources most of the time, so you likely won't run into this.)
- `gettype()` has stupid casing. `get_debug_type()` uses snake-case, like most newer parts of the PHP API, making it much easier to read.

In short, `get_debug_type()` is what `gettype()` should have been. The only reason it has a new name is for backward compatibility to avoid breaking people's expectations around `gettype()`.

The `get_debug_type()` RFC[52] comes to us courtesy of Mark Randall.

[51]https://wiki.php.net/rfc/non-capturing_catches
[52]https://wiki.php.net/rfc/get_debug_type

PDO error handling

PDO is PHP's standard cross-database abstraction layer library. It dates from the early PHP 5 days, before exceptions were fully adopted by the PHP community as a valid form of error handling. For that reason, it has a configurable error response, allowing the user to select silent failure with a method to check afterward, an `E_WARNING` trigger, or an exception as the way to handle errors. The default is silent, which … kind of made sense 15 years ago, in context, if you squint. It really makes no sense at all in 2020, however, and nearly all projects have been switching the error handling to throw exceptions for years.

In PHP 8.0, the default has changed to throw exceptions. The other error modes are still present, but really, don't use them.

AllenJB was responsible for this RFC change[53].

Regex errors

The preg (Perl-compatible REGular expressions) suite of functions is a staple of string handling in PHP, like in most languages. It's error handling, though, is rather arcane (and like most systems of its age derived from C conventions from the 80s). For instance, if something goes wrong you can consult the `preg_last_error()` function, which returns the error code of the last `preg_*()` function run. That error code isn't all that helpful, though, as you have to match it up with what it actually means manually.

PHP 8.0 now includes a `preg_last_error_msg()` function that returns the human-friendly text version of that last error. There's not much else to say about it, but if you're displaying error messages from regular expressions, it's now a little bit nicer.

Thrown trace string length

This is a highly obscure one. Exceptions and Errors in PHP include a backtrace of the call stack at the time they are created. The exception then has a method on it, `getTraceAsString()`, that will show the trace as one giant string. To keep that string from being too giant, however, it caps the length of the arguments shown for each function in the call stack to 15 bytes. That runs into the opposite problem, though; sometimes the value in an argument is an object, and you need to know more than it's first 15 bytes of string-ified data.

In PHP 8.0, there is now a new `php.ini` setting named `zend.exception_string_param_max_len` that allows you to change that length cap. Setting it to 0 will supress the data in each argument but still show other data about it, such as its type.

[53]https://wiki.php.net/rfc/pdo_default_errmode

You'll probably never need this value except in detailed debugging scenarios, but if you do, you have Tyson Andre to thank for the RFC[54].

`ValueError`

A tiny one; there is now a new built-in error type, `ValueError`. `ValueError` is thrown by certain internal functions "when the type of an argument is correct, but the value of it is incorrect." Examples include negative integers passed to a function that wants a positive integer, strings that contain invalid characters like null byte, or empty arrays for functions that need an array with content.

You can also `throw new ValueError` yourself for the same purpose.

`ValueError` didn't come through an RFC, but through other assorted error handling cleanup. As such the credit/blame can be shared by the internals team collectively.

[54]https://wiki.php.net/rfc/throwable_string_param_max_len

CMS support

In this case, CMS does not stand for Content Management System. (PHP already supports those.) It refers to the Cryptographic Message Syntax standard for encrypting and decrypting messages. CMS was defined in 2009 by IETF RFC 5652[55], so it's about time PHP supported it.

PHP 8.0 adds five new functions to handle CMS operations, which parallel the older pkcs7 functions (which serve the same purpose but use an older algorithm):

- `openssl_cms_encrypt()`
- `openssl_cms_decrypt()`
- `openssl_cms_sign()`
- `openssl_cms_verify()`
- `openssl_cms_read()`

We won't go into further detail about how they work other than that they exist. The RFC[56] has details if you need, but now you can say that PHP supports CMS without it being a pun.

CMS support comes to PHP courtesy of Eliot Lear.

[55]https://tools.ietf.org/html/rfc5652
[56]https://wiki.php.net/rfc/add-cms-support

Changes and backward compatibility challenges

There’s been a very steady trend in PHP over the last several years toward making the language tighter. That means more edge cases that are “undefined behavior that kinda silently works most of the time” turn into explicit warnings or errors, behavior that was documented but totally illogical gets adjusted to be more logical, and so on. Usually the impact is slight, and well-behaved code usually won’t notice a difference, but as we all know not all code is well-behaved.

PHP 8.0 continues the trend of tightening. In this section, we’ll cover some of the upcoming tidying up that may affect your existing code.

Stable sorting

What happens if you have an array and you sort it, but two of the elements are equal? Does their order change or not?

When the things being sorted are strings or integers, it doesn't really matter. If they're objects, however, it may not be obvious.

Suppose you're sorting an array of `Person` objects by age. Many people can have the same age, of course, so in what order do equally aged Persons end up?

In PHP 7, the resulting order was unpredictable. In PHP 8.0, the order is now "the same as it was before." That means if in the original array Jorge, age 40, appears before Melissa, age 40, and they're sorted by age, then Jorge will still appear before Melissa. Or, in code:

```php
$people[] = new Person('Jorge', 40);
$people[] = new Person('Melissa', 40);

usort($people, fn($a, $b) => $a->age <=> $b ->age);
// Jorge is guaranteed to still be before Melissa.
```

As a side effect, it used to be nominally possible to return a boolean from the comparison function rather than an integer, which is what is expected. In PHP, booleans can "weakly cast" to integers 1 and 0 in many circumstances. That's no longer supported in sort comparison functions and will now trigger a warning. (It was always a bug, now it's just explicit.)

The stable sorting RFC[57] comes to us courtesy of inconsistency slayer Nikita Popov.

[57]https://wiki.php.net/rfc/stable_sorting

Numeric string handling

PHP, like most popular interpreted languages, makes liberal use of type coercion. That is, a variable can change type depending on where it's used if it makes sense in context to do so. Most of these conversions involve playing fast and loose between strings and number types (int and float). For example, the integer 42 and the string "42" are generally "close enough" to the same thing that they can be considered equal (==), but not identical (===).

That "it's probably good enough" approach has its advantages, but also introduces a lot of bizarre edge cases. For that reason, PHP's scalar types support for function signatures (introduced in 7.0) has both a weak mode (that allows that kind of silent conversion) and a strict mode (which doesn't), with the strict mode being generally recommended for most use cases.

A number of other gotchas lurk in that silent conversion, though. In PHP 7.4, the following are, mind-bendingly enough, true:

```
0 == "wait, what?";          // true
0 == "";                     // true
99 == '99 bottles of beer';  // true
42 == '     42';             // true
42 == '42     ';             // false
in_array(0, ['a', 'b', 'c']); // true???
```

Moreover, in some contexts "42   " is considered "close enough" to int(42) and sometimes not.

The technical term for this situation is "totally bonkers." Fortunately, a pair of RFCs cleans up this silliness in PHP 8.0.

There's some subtlety to them, but the condensed version is that:

- "Numeric strings" are now any string containing all numeric values with leading or trailing whitespace, which can be safely ignored, rather than just leading whitespace. (Note that "numeric values" can include exponents and a dot, such as `"42.5e4"`, not just numerals.)
- Numeric strings are treated and defined consistently in all cases.
- When a numeric string is compared to an integer, with == or greater-than/less-than, the string is converted to an integer and then they're compared as integers.
- When a non-numeric string is compared to an integer, the integer is converted to a string and then they are compared as strings. No more `0 == "seriously, what?"`
- When comparing a string to a float, the same happens, but the float value may be mangled by floating point precision first, so it's not always guaranteed to behave as expected. (Such is life with floating point numbers and binary computers.)

- Trying to pass a non-numeric string to a function that expects a number will now throw a TypeError.
- Explicitly casting a numeric-leading string (like “99 bottles of beer”) to an integer will still result in int(99), but that’s the only circumstance in which that happens anymore.

All of these changes help make the language more predictable, logical, and consistent, but they are changes. If your code already plays fast and loose with types and string-to-number conversions, you may see some subtle changes in behavior. Fortunately, most code these days does not play that fast and loose with types (precisely to avoid this weirdness).

We have one RFC[58] from Nikita Popov and one RFC[59] from George Banyard to thank for this cleanup. Now go enable strict types in your codebase anyway.

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

[58]https://wiki.php.net/rfc/string_to_number_comparison
[59]https://wiki.php.net/rfc/saner-numeric-strings

Non-numeric Arithmetic

Another small "wait, what?" fix. Historically, PHP allowed nearly all arithmetic operations to be applied to non-numeric types. Not because it makes sense, but because in ye olden days the idea was that the code should not crash and should try to do something kinda-sorta reasonable, even if there was nothing logical to do.

That leads to weirdness like `[] % [5] == 0`. That makes absolutely no logical sense, but is just what falls out of the engine by accident.

In PHP 8.0, those nonsensical combinations now throw TypeErrors rather than silently doing something that may or may not make sense. A few do make sense, such as addition on arrays being a type of merge, and those are unchanged. The behavior on primitives (strings, bool, float, etc.) is also unchanged.

This tidying RFC[60] comes once again from the king of consistency, Nikita Popov.

[60]https://wiki.php.net/rfc/arithmetic_operator_type_checks

Extensions in, extensions out

The PHP engine itself is quite small. Most of the standard library people expect from PHP comes in the form of extensions, many of which ship with PHP itself. The list of what is included has evolved over the years.

In PHP 8.0, the JSON extension is now mandatory; it cannot be disabled, so code authors and system maintainers don't need to worry about the remote possibility that it might be disabled. (It almost never was disabled, but it *could* have been.) Tyson Andre gets the thanks for this RFC[61].

On the flipside, the XML-RPC extension has long been unmaintained and depends on long-unmaintained C libraries. You've probably never used it, in favor of user-space libraries of some kind if you even wrote XML-RPC code at all. For that reason, it's now been expelled from PHP. It's still available as a PECL extension if you really need it, but it's unlikely to get any maintenance or attention in the future. If you're one of the few people still using this extension, it's best to migrate to user-space equivalents that are better maintained.

We can thank Christoph M. Becker for giving XML-RPC a proper retirement party[62].

[61]https://wiki.php.net/rfc/always_enable_json
[62]https://wiki.php.net/rfc/unbundle_xmlprc

Stricter magic

Objects in PHP support a number of "magic methods": methods with a special name that have special behavior in the engine. We already saw `__toString()` as an example earlier in this book. All magic methods begin with __ to indicate that there's something special about them. (And if you have a method of your own that begins with a double underscore but doesn't tie into special engine behavior, you are officially Doing It Wrong^tm^.) Most of those methods were added well over a decade ago, however, which means they predate PHP adding widespread type support in method signatures.

That isn't a huge problem, as long as the code is well-behaved. However, the whole point of typed function signatures is that the language itself will slap your hand if your code isn't well-behaved so you know to fix it before it causes subtle data-losing bugs. Unfortunately, the language allowed developers to add type declarations to their magic methods … even if those declarations were contrary to what the method was supposed to do. Oops.

PHP 8.0 now optionally allows you to declare the right types in your method signatures and will slap your hand (the technical phrase is "throw a Fatal error") if you specify the wrong one. For the vast majority of users nothing happens, but it allows those who prefer the language to do their work for them to do so safely.

In particular, the following magic methods now support, and enforce, the following typed signatures:

```php
Foo::__call(string $name, array $arguments): mixed;

Foo::__callStatic(string $name, array $arguments): mixed;

Foo::__clone(): void;

Foo::__debugInfo(): ?array;

Foo::__get(string $name): mixed;

Foo::__invoke(mixed $arguments): mixed;

Foo::__isset(string $name): bool;

Foo::__serialize(): array;

Foo::__set(string $name, mixed $value): void;

```

```
Foo::__set_state(array $properties): object;

Foo::__sleep(): array;

Foo::__unserialize(array $data): void;

Foo::__unset(string $name): void;

Foo::__wakeup(): void;
?>
```

Although optional, I would recommend including the types in all cases to make code more self-documenting and nitpicky about finding errors for you early on.

This added type safety[63] is thanks to Gabriel Caruso.

[63]https://wiki.php.net/rfc/magic-methods-signature

Stricter warnings and errors

PHP has a variety of error levels (Notice, Warning, Error) that it can trigger when something goes wrong, as well as the ability to throw Exceptions or engine Errors. Determining the appropriate severity of a problem is always a tricky problem, especially when some of those options didn't exist when a given error was first defined.

In PHP 8.0, several errors became stricter. The full list[64] is included in the RFC, but the most notable ones are:

- Many notices around missing values or using invalid types in weird places are now warnings. That includes reading undefined variables, properties, and array keys.
- Several warnings around misuse of arrays and traversables are now `TypeError` exceptions. They are a type problem, and anything the code tries to do after that is guaranteed to be wrong, so that makes sense.
- Division by zero (using `/` instead of `fdiv()`) now throws a `DivisionByZeroError` instead of a warning.

You get three guesses who we can thank for this RFC, and the first two guesses of Nikita Popov don't count.

Additionally, previous versions of PHP were inconsistent when enforcing Liskov Substitution rules on inheritance. A class that inherits from a parent and has an incompatible method signature on a parameter generates a warning; If it is implementing an interface, or if it's the return type that is inconsistent, it generates a fatal error. The reason class parameter inheritance wasn't as strict was mainly for BC reasons, aka, it was made too loose when that was first added in PHP 5.0 before the internals team realized it really should be fatal.

In PHP 8.0, that oversight has been corrected, and it now generates a fatal error, As Barbra Liskov Intended^{tm}.

Once again, this RFC[65] was all Nikita Popov's fault.

Meanwhile, while passing a variable to a user-defined function that was not compatible with the function's parameter type has generated a `TypeError` since PHP 7.0, that was not the case for internal functions. Sometimes those throw a `TypeError`, sometimes a warning, sometimes they just return null. Which function did which was largely random, due to variation in the internal C implementation of each function.

Fan favorite Nikita Popov was responsible for the small RFC[66] that normalized that so that internally-defined functions always behave the same as user-defined functions and throw a `TypeError` on type errors.

[64]https://wiki.php.net/rfc/engine_warnings
[65]https://wiki.php.net/rfc/lsp_errors
[66]https://wiki.php.net/rfc/consistent_type_errors

Resource evolution and devolution

Finally, although it wasn't an RFC, a lot of work has been done behind the scenes to convert resources to objects.

"Resources" in PHP speak are a special data type that dates from before PHP even had objects, so we're talking about the Bill Clinton presidency here. Resources are kind of like objects, only worse. They can only be implemented by extensions and don't support most of what objects support. It's generally acknowledged that they were a bad idea and real objects are a better solution in just about every case, but there are still many very common and very old extensions that expose resources instead of objects to user code. That's especially true of things like database connections, the file system, and other "external thingies."

There has been an ongoing effort to convert those resources to objects, a process that is 99% transparent to user code, with the end goal of removing resources from the language entirely. Much of that conversion was completed in PHP 8.0, albeit mostly for the less used extensions.

The odds of that affecting your code are very small. Likely the only reason you'd be affected is if you are checking if a variable `is_resource()` and are using the CURL, OpenSSL, Sockets, XML-RPC, ZIP, or ZLIB extensions. If so, in PHP 8.0 that function will now return `false` instead of `true`. Otherwise you likely won't notice.

If you have no idea what this is all about, congratulations, you most likely don't need to care.

`get_resource_id()`

For those few remaining resources, however, there's a new way to track them. Resources are identified by an integer ID (which can get reused if a resource is closed, making them even less useful). The typical way to access that ID has been to cast a resource to an integer, like so:

```php
$fh = fopen('somefile.txt', 'r');
print (int)$fh;
// prints "1" or similar.
```

Casting is an ugly operation, though, so there is now a `get_resource_id()` function that returns the same value.

```php
$fh = fopen('somefile.txt', 'r');
print get_resource_id($fh);
// prints "1" or similar.
```

This function also snuck in without an RFC, which is fine as you should probably never use it. Or resources, for that matter.

PHP 8 on Platform.sh

The easiest way to try out PHP 8.0 is, unsurprisingly, on Platform.sh.

Platform.sh is a managed Platform-as-a-Service (PaaS) with support for a variety of different languages, applications, frameworks, and services. It provides the same benefits as building your own CI/CD cluster with Kubernetes, Helm, Docker, and so forth … but without having to fuss around with Kubernetes, Helm, Docker, and so forth.

What makes Platform.sh even better, though, is its data cloning capability. Your Git production branch is always on, well, production. Pushing new code to that branch will cause it to be rebuilt (download Composer dependencies, compile Sass files, etc.) and then redeployed, taking your changes live.

If you `git push` to another branch, the same process happens to produce a separate environment. Platform.sh then does a copy-on-write clone of data from production to your new environment, in roughly constant time. That means any branch can, in moments, be an exact code and data duplicate of your production environment, ready to hack on. Or, say, try out PHP 8.0 without risking production.

In this section, we'll go over getting set up on Platform.sh, complete with a local development environment, and how to then test your PHP application on Platform.sh for PHP 8.0. We'll start with a template project to show how it all works, and then how to add your own application instead.

Starting with Platform.sh

Create a free trial

The best way to understand a tool is to use it. That's why Platform.sh offers a free one-month trial.

Visit the Platform.sh accounts[67] page and fill out your information to set up your trial account.

Alternatively, you can sign up using an existing GitHub, Bitbucket, or Google account. If you choose this option, you will be able to set a password for your Platform.sh account later.

Create a project

After creating an account, you'll be presented with the Platform.sh management console. From here you can create projects, adjust account settings, and a lot more.

Since you do not yet have any projects on Platform.sh, the first thing you will see when you sign in is a workflow for creating a new project.

1. Project type

You will first be given the option to `Create an empty Project` (one that you can migrate your own code base to) or to `Use a template`. Select the `Use a template` option, and then click `Next`.

2. Details

Give your project a name like `My First Project`.

In the next field, you have the option to configure the project's region, which corresponds with the data center where your project will live. Select the region that most closely matches where most of your traffic will come from, and then click `Next`.

3. Template

Now you will be able to see the large collection of Platform.sh's supported templates. There are several types of templates available, including simple language-specific examples, ready-to-use frameworks you can build upon, and set applications you can start using immediately after installation.

[67] https://accounts.platform.sh/platform/trial/general/setup

Since this is a PHP book, select the dropdown labeled "All languages" and select "PHP". That will limit the list to just PHP-based templates. Select the "Basic PHP" project for now, as that's what will be described in detail. If you'd rather jump straight to some other project, though, you are welcome to do so.

Select the template, and then click `Next`.

You can find the source code for all Platform.sh templates in the GitHub Templates Organization[68].

4. Plan & Pricing

Under your free trial, your project will be created under a "Development" plan size. The management console will tell you how many users and development environments are part of the plan, as well the price of that plan once your trial has completed. After you have read through the features, click `Continue`.

With these few selections Platform.sh will create the project and build the template in less than two minutes.

A brief tour of the console

Once you have configured the template application, Platform.sh will build your project for you.

1. Explore the management console

When the build screen has cleared, Platform.sh will return you to the management console. Since you now have a project on your account, a version of this page will be what you see each time you visit the console.

You will start on the main page for your new project, `My First Project`. From here, you can control the settings of this project and monitor its status.

In the `Environments` box, click on `Master`.

2. Check the build status

Take a minute to notice some the information available on this page.

[68]https://github.com/platformsh-templates

- **Overview**

  In this box the `Master` environment, which is a live environment built from the `master` branch of your application code, will have a status of `Building`.

  Once that status has updated to `Active`, the build is complete and the application has deployed.
- **Environment Activity**

  In this block, you can see what you have done so far has two initial entries: `My First Project` was created, and the template profile you chose was initialized on the environment.

3. Done!

That's it! Once the build status has changed to `Active`, your application has been deployed on Platform.sh.

You can view the template by clicking on the link that is now visible for the `Master` environment under the Overview box. It will open another tab in your browser to your new live site.

All set

In these few steps you created a free trial account, configured a template application on a project, and deployed it using the management console entirely from your browser.

Using the Platform.sh CLI, however, you get even more control over your project configurations, including the ability to migrate your own applications to Platform.sh.

Install the CLI

Prerequisites

With the management console you can start new projects from templates just as you did in the previous steps, but deploying your own applications requires you to also use the Platform.sh CLI.

Before you install it there are a few requirements that must be met first.

Git

Git is the open source version control system used by Platform.sh. Any change you make to your Platform.sh project will need to be committed via Git. You can see all the Git commit messages of an environment in the `Environment Activity` feed of the management console for each project you create.

Before getting started, make sure you have Git installed[69] on your computer.

SSH

You will need Secure Shell (SSH) to securely connect to your Git repository and environments. SSH clients are readily available for every platform and may already be installed on your computer.

Platform.sh supports both keypair-based and certificate-based authentication. Both are secure and protect your account from snooping when you log in. For now, you can use certificate-based authentication as that is easier. You will be prompted to login via your web browser the first time you run `platform ssh`.

Installation

With all of the requirements met, install the CLI to start developing with Platform.sh.

1. Install the CLI

In your terminal run the following command:

Installing on OSX or Linux

[69] https://git-scm.com/

```
curl -sS https://platform.sh/cli/installer | php
```

Installing on Windows

```
curl https://platform.sh/cli/installer -o cli-installer.php
php cli-installer.php
```

2. Authenticate and Verify

Once the installation has completed, you can run the CLI in your terminal with the command

```
platform
```

If you opened your free trial account using another login (i.e., GitHub), you will not be able to authenticate with this command until you setup your account password with Platform.sh in the console.

You should now be able to see a list of your Platform.sh projects, including the template you made in this guide. You can copy its *project ID* hash and then download a local copy of the repository with the command

```
platform get <project ID>
```

With a local copy, you can create branches, commit to them, and push your changes to Platform.sh right away.

```
git push platform master
```

Take a minute to explore some of the commands available with the CLI by using the command `platform list`.

That’s it! Now that you have the management console set up and the CLI installed on your computer, you’re well on your way to exploring all of the ways that Platform.sh can improve your development workflow.

Configuration

A Platform.sh project is a set of deployment tools on top of a Git repository, such that that repository's branches can be built and deployed for each branch (an environment) as your projects evolve. Your PHP 8.0 application will reside within its own *Application* container, which makes up one third of your cluter. It will also contain a *Router* container and potentially one or more *Service* containers.

Each of these container types are configured from separate files committed to your repository, which means that your entire project's infrastructure is driven by just three files. These files provide all the directives needed to turn your pile-of-code in a Git repository into a live, running application.

- A `.platform/routes.yaml` file, which configures the routes used in your environments. That is, it describes how an incoming HTTP request is going to be processed by Platform.sh.
- A `.platform/services.yaml` file, which configures the services that will be used by the application, such as MariaDB or Redis. Including one of Platform.sh's maintained services in your project only requires writing this file. While this file must be present, if your application does not require services it can remain empty.
- At least one `.plaform.app.yaml` file, which configures the application itself. It provides control over the way the application will be built and deployed on Platform.sh. It is this file that controls what language runtime your application needs, and what version.

```
/
├── .platform
|   ├── routes.yaml
|   └── services.yaml
├── .platform.app.yaml
└── < application code >
```

If you are following along and created a project from a template then these files will already be available, and most are heavily commented. The pages below assume the "Plain PHP" template.

There is another directory, `.platform/local`, that is created when you check out a project using the CLI. That contains local tracking information for the CLI itself and should not be committed to Git. It includes a `.gitignore` file to exclude that directory, but it can accidentally get included fairly easily. If you find it in your Git repo, just remove it completely.

If you don't want the tour and want to skip straight to upgrading to PHP 8.0, go on to the next chapter. This one will be here waiting for you when you return.

Service configuration

The `.platform/services.yaml` file defines what backend services are available to your project and what version. Platform.sh supports many common services, some of which can be further configured.

The "Plain PHP" template does not ship with any services by default, so the file will be empty. An example `.platform/services.yaml` for a more built-out application may look something like this:

```yaml
# The services of the project.
#
# Each service listed will be deployed
# to power your Platform.sh project.

db:
  type: mariadb:10.4
  disk: 2048

cache:
  type: redis:6.0
```

If your application does not use any services at this point then you can leave it blank, but it must exist in your repository to run on Platform.sh. If your application does use a database or other services, you can configure them with the following attributes:

- The key for each service is its name. It can be any unique value, so long as it is alphanumeric. If your application requires multiple services of the same type, make sure to give them different names so that your data from one service is never overwritten by another. (Note, however, that most services that you might need multiple instances of support multiple databases, indexes, or cores, so you will rarely need to create a second service.)
- `type`: This specifies the service type and its version using the format

  ```
  type:version
  ```

  The table below lists all Platform.sh maintained services, along with their `type` and supported `versions`.

Service	`type`	Supported `version`
Headless Chrome	`chrome-headless`	73
Elasticsearch	`elasticsearch`	6.5, 7.2, 7.5, 7.7
InfluxDB	`influxdb`	1.2, 1.3, 1.7, 1.8
Kafka	`kafka`	2.1, 2.2, 2.3, 2.4, 2.5
MariaDB	`mariadb`	10.0, 10.1, 10.2, 10.3, 10.4
Memcached	`memcached`	1.4, 1.5, 1.6
MongoDB	`mongodb`	3.0, 3.2, 3.4, 3.6
Network Storage	`network-storage`	1.0
Oracle MySQL	`oracle-mysql`	5.7, 8.0
PostgreSQL	`postgresql`	9.6, 10, 11, 12
RabbitMQ	`rabbitmq`	3.5, 3.6, 3.7, 3.8
Redis	`redis`	3.2, 4.0, 5.0, 6.0
Solr	`solr`	7.7, 8.0, 8.4, 8.6
Varnish	`varnish`	5.6, 6.0

- `disk`: The `disk` attribute configures the amount of persistent disk that will be allocated between all of your services. Projects by default are allocated 5 GB (5120 MB), and that space can be distributed across all of your services. Note that not all services require disk space. If you specify a `disk` attribute for a service that doesn't use it, like Redis, you will receive an error when trying to push your changes.

Platform.sh provides *managed services*, and each service comes with considerable default configuration that you will not have to include yourself in `services.yaml`.

Each service may have additional attributes to configure in `services.yaml`. Check out the full Platform.sh documentation[70] or the template for your application for more details.

Application configuration

The `.platform.app.yaml` file is the most flexible of the three configuration files. It defines how the application container that runs your code should behave. That includes both its *build time* and its *runtime* configuration.

Most applications today are more complex than "check code out of Git, run." At minimum, most PHP applications have a `composer install` step. Many also require compiling Sass files, compressing CSS and JS files, generating compiled dependency injection or event dispatcher code, and so forth. Platform.sh considers all of that the "build step," and it can be arbitrarily complex.

The *runtime* configuration includes what runtime and version to use, what services to connect to, how the web server itself should be configured, and what commands should run once after each code update.

A basic PHP application's `.platform.app.yaml` can look like this:

[70]https://docs.platform.sh/configuration/services.html

A basic PHP .platform.app.yaml file

```yaml
# The name of this app. Must be unique within a project.
name: app

# The type of the application to build.
type: php:7.4
build:
  flavor: composer

# The hooks that will be performed when the package is deployed.
hooks:
  build: |
    set -e

  deploy: |
    set -e

# The relationships of the application with services or other applications.
# The left-hand side is the name of the relationship as it will be exposed
# to the application in the PLATFORM_RELATIONSHIPS variable. The right-hand
# side is in the form `<service name>:<endpoint name>`.
relationships:
    database: "db:mysql"

# The size of the persistent disk of the application (in MB).
disk: 2048

# The mounts that will be performed when the package is deployed.
mounts:
  # Because this directory is in the webroot, files here will be web-accessible.
  'web/uploads':
    source: local
    source_path: uploads
  # Files in this directory will not be web-accessible.
  'private':
    source: local
    source_path: private


# The configuration of app when it is exposed to the web.
web:
  locations:
    "/":
      # The public directory of the app, relative to its root.
      root: "web"
      # The front-controller script to send non-static requests to.
      passthru: "/index.php"
```

The `.platform.app.yaml` file is extremely flexible and can contain many lines with very fine-grained control over your application. At the very least, Platform.sh requires three principle attributes in this file to control your builds:

- `name`: The name of your application container does not have to be the same as your project name, and in most single application cases you can simply name it `app`. You should notice in the next section, when you configure how requests are handled in `.platform/routes.yaml` that the `name` is reused there, so it is important that those are the same.

  If you are trying to to deploy multiple applications in one project, the only constraint is that each of these application names must be unique.

- `type`: The `type` attribute in `.platform.app.yaml` sets the container base image for the application and sets the primary language. In general, `type` should have the form

```
type: <runtime>:<version>
```

  For now the only `runtime` we care about is `php`, and the supported `versions` are `7.2`, `7.3`, `7.4`, and now `8.0`. Older versions going as far back as PHP 5.4 are also available but are not maintained and should be avoided. Current versions of Node.js, Python, Ruby, Go, Java, and Elixir are also available.

There are a few additional keys in `.platform.app.yaml` you will likely need to fully configure your application but are not required:

* `relationships`: Relationships define how services are mapped within your application. Without this block, an application cannot communicate with a service container. Provide a unique name for each relationship and associate it with a service. For example, if in the previous step you defined a MariaDB container in your `.platform/services.yaml` with

or the template for your application for more details.

Route configuration

The final configuration file you will need to modify in your repository is the `.platform/routes.yaml` file, which describes how an incoming HTTP request is going to be processed by Platform.sh.

A minimal configuration `.platform/routes.yaml` for all languages will look very similar:

```yaml
# The routes of the project.
#
# Each route describes how an incoming URL is going
# to be processed by Platform.sh.

"https://{default}/":
  type: upstream
  upstream: "app:http"

"https://www.{default}/":
  type: redirect
  to: "https://{default}/"
```

Configuring the routes can be done using either an absolute URL or a URL template as shown in the examples above. With the form `http://www.{default}`, `{default}` will be substituted by either your configured domain or those automatically generated by Platform.sh.

If you set up a domain of `example.com`, the route configuration `http://www.{default}` will be resolved to `http://www.example.com/`. Your production (`master`) environment's routes will be configured according to these rules, but so will each development environment that you activate.

Each route can then be configured with the following properties:

- `type`:
  * `upstream`: serves the application. Takes the form `upstream: <application name>:http`, using the application `name` set in your `.platform.app.yaml`.
  * `redirect`: configures redirects from `http://{default}` to your application.
- `cache`: controls caching behavior of the route.
- `redirects`: controls redirect rules associated with the route.

Each application may have additional attributes to configure in `routes.yaml`. Check out the full Platform.sh documentation[71] or the template for your application for more details.

[71]https://docs.platform.sh/configuration/routes.html

Upgrading PHP

Now that you have a basic PHP project, what are you going to do with it? Upgrade it to PHP 8.0, of course. This will be a trivial change, but demonstrates the power and ease of making changes with Platform.sh.

Check out the project

If you haven't already, use the Platform.sh CLI to download a local copy of your site. In the management console, open the "CLI" drop-down in the top right. It will give you a command to copy and paste that will look something like

```
platform get abc123 -e master
```

`abc123` will be replaced with your project ID. Paste that into a terminal window on your computer in whatever directory you prefer. That will check out the project on the `master` branch (aka "environment," or `-e` for short) into a new subdirectory. Change into that directory.

Make a branch

Git drives almost everything on Platform.sh. Every branch in your repository can be spun up as an "environment" with its own complete set of services. That means the first step is making a branch.

You can make a branch the normal Git way:

```
git checkout -b upgrade
```

or you can use the Platform.sh CLI To create it. The latter will create a branch on Platform.sh's repository and "activate" it, creating a completely new copy of the entire site off of the same code base, and branch locally.

```
platform branch upgrade
```

Upgrade PHP

Now open the `.platform.app.yaml` file in the text editor of your choice. Locate the line that reads `type: php:7.4`. Change the "7.4" to "8.0" and save the changes, then exit.

That's it. You've now changed your language version, on that one branch, to PHP 8.0. (Platform.sh manages the patch release and upgrades that periodically for you. Every deploy will use the most recent PHP 8.0.x version we have packaged.)

Commit that change, push the code, and if necessary activate the environment. If you used a raw Git command before, you would run:

```
git commit -am "Try out PHP 8.0"
git push -u platform upgrade
platform activate -y
```

If you used the Platform.sh CLI before, you can run:

```
git commit -am "Try out PHP 8.0"
git push
```

In either case you'll now have an environment on Platform.sh named `upgrade` that is running the same code as `master`, but on PHP 8.0.

Review and Merge

From the management console, navigate to the `upgrade` environment and click the link on the left to go to the home page of that environment. Since this is a Basic PHP project, there should be no difference. With a real project there likely won't be any difference either, but it's possible you may run into some language tightening as discussed in earlier chapters. If so, correct those, commit, and push again until you're happy with the result.

When you're ready to switch your production environment (the `master` branch) to PHP 8.0, merge the `upgrade` branch to `master`. There's three ways you can do that:

- Click the "Merge" button on the "Upgrade" branch in the management console. That performs a `git merge` on the server and causes the `master` environment to redeploy with the new code.
- Run `platform merge` in your local Git checkout while on the `upgrade` branch. That will do the same thing as the "Merge" button in the browser.
- Merge your code from `upgrade` to `master` locally using normal Git commands, then push the `master` branch.

In any case, your `master` branch is now updated with the PHP 8.0 setting and any code changes you made. That will cause the environment to be rebuilt (rerun Composer and anything else in your build hook) and redeployed.

You can also verify by logging into your environment over SSH. Run

```
platform ssh -e master
```

That will open a bash shell on your application container on Platform.sh. Now run

```
php -v
```

You should get output similar to the following:

```
PHP 8.0.0 (cli) ( NTS )
Copyright (c) The PHP Group
Zend Engine v4.0.0, Copyright (c) Zend Technologies
    with Zend OPcache v8.0.0, Copyright (c), by Zend Technologies
```

There you have it. PHP 8.0.

You can now delete the `upgrade` branch to clean up after yourself.

Local development with Lando

Platform.sh solves the production and staging environments problem. However, you still need to be able to edit and test code locally, as running a Git push every time you want to see if a code changed worked is … inefficient.

Platform.sh is agnostic about what your local development environment is. You're welcome to use whatever works for your workflow. However, the best tool that Platform.sh has found is Lando[72], and Lando has now added specific integration for Platform.sh.

Lando works as a layer on top of Docker Compose. In the case of Platform.sh, it can run locally to read your Platform.sh configuration files and translate them on the fly into Docker Compose instructions. Although Platform.sh doesn't use Docker internally, it does provide Docker versions of all of its application and service containers that Lando can then use.

The net result is that with Lando, you can run locally the exact same container images that will be used on Platform.sh. If you want to upgrade your language or service version, or add or remove services, you can just edit the Platform.sh configuration files and that will take effect locally as well on Platform.sh, using the exact same code. It really is the fastest and simplest way to develop a website.

Installation

Lando's documentation includes detailed installation instructions[73], but we'll cover the quickstart here.

First make sure you have the latest version of Docker installed and a recent version of Node.js.

For Mac

- Download the latest .dmg package from GitHub[74]
- Mount the DMG by double-clicking it
- Double-click on the LandoInstaller.pkg
- Go through the setup workflow
- Enter your username and password when prompted

For Debian/Ubuntu

Run the following to install the `.deb` package:

[72]https://lando.dev/
[73]https://docs.lando.dev/basics/installation.html
[74]https://github.com/lando/lando/releases

```
wget https://files.devwithlando.io/lando-stable.deb
sudo dpkg -i lando-stable.deb
```

For Fedora and derivatives

Run the following to install the `.rpm` package:

```
wget https://files.devwithlando.io/lando-stable.rpm
sudo dnf install lando-stable.rpm
```

For Arch Linux

Run the following to install the `.packman` package:

```
wget https://files.devwithlando.io/lando-stable.pacman
sudo pacman -U lando-stable.pacman
```

For Windows 10 Professional

- Make sure you are using at least Windows 10 Professional with the latest updates installed and with Hyper-V enabled.
- Download the latest Windows .exe installer from GitHub[75].
- Double-click on lando.exe
- Go through the setup workflow
- Approve various UAC prompts during install

Using Lando

Once Lando is installed, you technically don't need anything further. Run

```
lando init
lando start
```

in your local Git repository. The first command will walk you through connecting Lando to your local repository and to the Platform.sh project it corresponds to. Then download the appropriate container images. (Depending on how many services you have and the speed of your connection, this may take a while.) When it's done, `lando start` will load the containers and start the application.

Now that it's running, the command `lando pull` will download files and databases from the corresponding Platform.sh environment (based on the branch currently checked out) to your local

[75]https://github.com/lando/lando/releases

Lando instance, allowing you to have a complete local copy of your entire site to work from. (It won't be a constant time copy-on-write operation the way it is between Platform.sh environments, of course, but it's still just one command.) The `lando pull` help text has more information on how to specify what to download.

When you're done, `lando stop` will shut down the containers, and `lando destroy` will wipe them from existence to free up disk space.

Some Platform.sh templates include a `.lando.upstream.yml` file with additional configuration for Lando specific to that template. You can also add a `.lando.yml` file with your own configuration as well. Consult the Lando documentation for more details.

Connecting to services

Most applications require more than just PHP code, of course. They need to talk to some backend service like a database.

To add a MariaDB database to your application, include the following in the `.platform/services.yaml` file:

```
db:
    type: mariadb:10.5
    disk: 1024
```

That declares that Platform.sh should create a MariaDB service, specifically version 10.5, when the application is deployed. It will also have 1 GB of disk storage set aside for it.

Then, in your `.platform.app.yaml` file, you need to tell the application about the service by defining a `relationship`:

```
relationships:
    database: "mysqldb:mysql"
```

In order to connect to this service and use it in your application, Platform.sh exposes its credentials in the application container within a base64-encoded JSON `PLATFORM_RELATIONSHIPS` environment variable.

To access this variable you can install the Platform.sh configuration reader library

```
composer install platformsh/config-reader
```

and access the credentials of `database`

```
use Platformsh\ConfigReader\Config;

$config = new Config();
$credentials = $config->credentials('database');
```

or read and decode the environment variable directly.

```
$relationships = json_decode(base64_decode(getenv('PLATFORM_RELATIONSHIPS')), TRUE);
$credentials = $relationships['database'];
```

In either case, `credentials` can now be used to connect to `database`:

```
{
  "username": "user",
  "scheme": "mysql",
  "service": "mysql",
  "fragment": null,
  "ip": "169.254.197.253",
  "hostname": "czwb2d7zzunu67lh77infwkm6i.mysql.service._.eu-3.platformsh.site",
  "public": false,
  "cluster": "rjify4yjcwxaa-master-7rqtwti",
  "host": "mysql.internal",
  "rel": "mysql",
  "query": {
    "is_master": true
  },
  "path": "main",
  "password": "",
  "type": "mysql:10.2",
  "port": 3306
}
```

You can find out more information about Platform.sh Config Reader library[76] on GitHub.

You can now use those credentials to connect to the database with PDO or `mysqli_connect()` or whatever your database layer of choice is. Once that's done, your application can do whatever it is your application wants to do.

Every branch environment has its own entirely independent instance of every service. When an environment is created it will start with a copy of its parent branch's data, but after that you can modify it and use it for testing however you like without impacting the production site at all.

That separation gives you the confidence to experiment freely with your application, develop new features, and then deploy new versions to production safely and easily.

Yes, even on Friday.

[76]https://github.com/platformsh/config-reader-php