What’s New in PHP 8.1: Features, Changes, Improvements, and More

Released on November 25, 2021, PHP 8.1 is finally here, packed with several exciting features.

In this article, we'll cover in detail what's new in PHP 8.1. From its new features and performance improvements to significant changes and deprecations, we'll go through them all in-depth.

Sit tight!

New Features in PHP 8.1

Let's start by covering all the new features in PHP 8.1. It's quite a list.

Info

PHP 8.1 is now available at Kinsta for all environments. Please check our PHP 8.1 feature update for more information.

Pure Intersection Types

PHP 8.1 adds support for intersection types. It's similar to union types introduced in PHP 8.0, but their intended usage is the exact opposite.

To understand its usage better, let's refresh how type declarations work in PHP.

Essentially, you can add type declarations to function arguments, return values, and class properties. This assignment is called type hinting and ensures that the value is of the correct type at call time. Otherwise, it throws up a TypeError right away. In turn, this helps you debug code better.

However, declaring a single type has its limitations. Union types help you overcome that by allowing you to declare a value with multiple types, and the input has to satisfy at least one of the declared types.

On the other hand, the RFC describes intersection types as:

> *An "intersection type" requires a value to satisfy multiple type constraints instead of a single one.*
>
> *…pure intersection types are specified using the syntax T1&T2&… and can be used in all positions where types are currently accepted…*

Note the usage of & (AND) operator to declare intersection types. In contrast, we use the | (OR) operator to declare union types.

Using most standard types in an intersection type will result in a type that can never be fulfilled (e.g. integer and string). Hence, intersection types can only include class types (i.e. interfaces and class names).

Here's an example code of how you can use intersection types:

```
class A {
    private Traversable&Countable $countableIterator;

    public function setIterator(Traversable&Countable $countableIterator)
        $this->countableIterator = $countableIterator;
    }

    public function getIterator(): Traversable&Countable {
        return $this->countableIterator;
    }
}
```

In the above code, we've defined a variable **countableIterator** as an intersection of two types: **Traversable** and **Countable**. In this case, the two types declared are interfaces.

Intersection types also conform to standard PHP variance rules already used for type checking and inheritance. But there are two additional rules with how intersection types interact with subtyping. You can read more about intersection types' variance rules in its RFC.

In some programming languages, you can combine Union Types and Intersection Types in the same declaration. But PHP 8.1 forbids it. Hence, its implementation is called "pure" intersection types. However, the RFC does mention that it's "left as a future scope."

Enums

PHP 8.1 is finally adding support for enums (also called enumerations or enumerated types). They're a user-defined data type consisting of a set of possible values.

The most common enums example in programming languages is the **boolean** type, with `true` and `false` as two possible values. It's so common that it's baked into many modern programming languages.

As per the RFC, enums in PHP will be restricted to "unit enumerations" at first:

> *The scope of this RFC is limited to "unit enumerations," that is, enumerations that are themselves a value, rather than simply a fancy syntax for a primitive constant, and do not include additional associated information. This capability offers greatly expanded support for data modeling, custom type definitions, and monad-style behavior. Enums enable the modeling technique of "make invalid states unrepresentable," which leads to more robust code with less need for exhaustive testing.*

To get to this stage, the PHP team studied many languages that already support enumerations. Their survey found that you can categorize enumerations into three general groups: Fancy Constants, Fancy Objects, and full Algebraic Data Types (ADTs). It's an interesting read!

PHP implements "Fancy Objects" enums, with plans to extend it to full ADTs in the future. It's conceptually and semantically modeled after enumerated types in Swift, Rust, and Kotlin, though it's not directly modeled on any of them.

The RFC uses the famous analogy of suits in a deck of cards to explain how it'll work:

```
enum Suit {
  case Hearts;
  case Diamonds;
  case Clubs;
  case Spades;
}
```

Here, the **Suit** enum defines four possible values: **Hearts**, **Diamonds**, **Clubs**, and **Spades**. You can access these values directly using the syntax: `Suit::Hearts`, `Suit::Diamonds`, `Suit::Clubs`, and `Suit::Spades`.

This usage may seem familiar, as enums are built atop classes and objects. They behave similarly and have almost the exact requirements. Enums share the same namespaces as classes, interfaces, and traits.

The enums mentioned above are called **Pure Enums**.

You can also define **Backed Enums** if you want to give a scalar equivalent value to any cases. However, backed enums can have only one type, either `int` or `string` (never both).

```
enum Suit: string {
  case Hearts = 'H';
  case Diamonds = 'D';
  case Clubs = 'C';
  case Spades = 'S';
}
```

Furthermore, all the different cases of a backed enum must have a unique value. And you can never mix pure and backed enums.

The RFC delves further into enum methods, static methods, constants, constant expressions, and much more. Covering them all is beyond the scope of this article. You can refer to the documentation to familiarize yourself with all its goodness.

The `never` Return Type

PHP 8.1 adds a new return type hint called `never`. It's super helpful to use in functions that always `throw` or `exit`.

As per its the RFC, URL redirect functions that always `exit` (explicitly or implicitly) are a good example for its use:

```
function redirect(string $uri): never {
    header('Location: ' . $uri);
    exit();
}

function redirectToLoginPage(): never {
    redirect('/login');
}
```

A `never`-declared function should satisfy three conditions:

- It shouldn't have the `return` statement defined explicitly.
- It shouldn't have the `return` statement defined implicitly (e.g. **if-else** statements).
- It must end its execution with an `exit` statement (explicitly or implicitly).

The URL redirection example above shows both explicit and implicit usage of the `never` return type.

The `never` return type shares many similarities with the `void` return type. They both ensure that the function or method doesn't return a value. However, it differs by enforcing stricter rules. For example, a `void`-declared function can still `return` without an explicit value, but you cannot do the same with a `never`-declared function.

As a rule of thumb, go with `void` when you expect PHP to continue executing after the function call. Go with `never` when you want the opposite.

Furthermore, `never` is defined as a "bottom" type. Hence, any class method declared `never` can "never" change its return type to something else. However, you can extend a `void`-declared method with a `never`-declared method.

Info

The original RFC lists the `never` return type as `noreturn`, which was a return type already supported by two PHP static analysis tools, namely Psalm and PHPStan. As this was proposed by the authors of Psalm and PHPStan themselves, they retained its terminology. However, owing to naming conventions, the PHP team conducted a poll for `noreturn` vs `never`, with `never` emerging as the forever winner. Hence, for PHP 8.1+ versions, always substitute `noreturn` with `never`.

Fibers

Historically, PHP code has almost always been synchronous code. The code execution halts till the result is returned, even for I/O operations. You can imagine why this process may make code execution slower.

There are multiple third-party solutions to overcome this obstacle to allow developers to write PHP code asynchronously, especially for concurrent I/O operations. Some popular examples include amphp, ReactPHP, and Guzzle.

However, there's no standard way to handle such instances in PHP. Moreover, treating synchronous and asynchronous code in the same call stack leads to other problems.

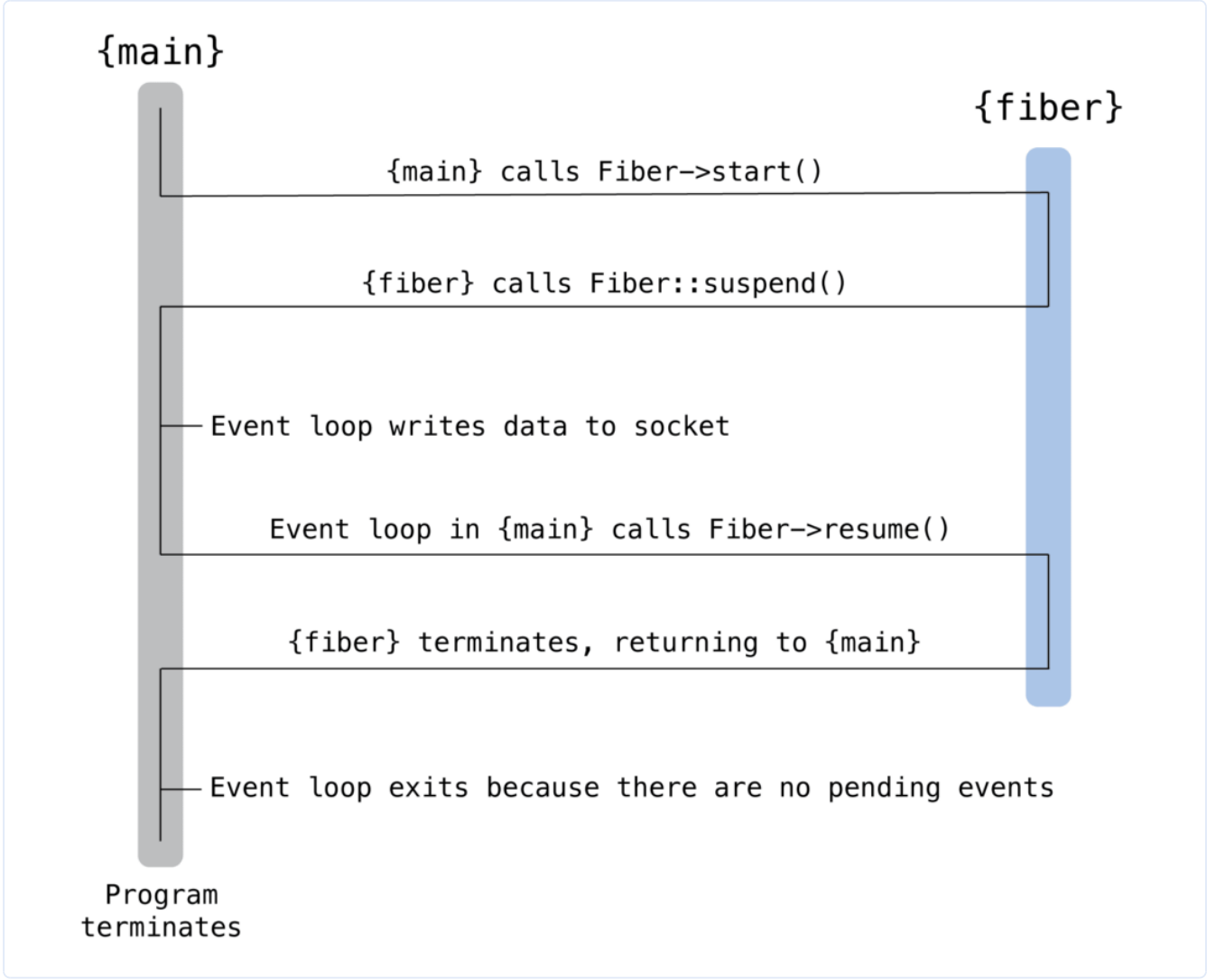
Fibers are PHP's way of handling parallelism via virtual threads (or green threads). It seeks to eliminate the difference between synchronous and asynchronous code by allowing PHP functions to interrupt without affecting the entire call stack.

Here's what the RFC *promises*:

- Adding support for Fibers to PHP.
- Introducing a new Fiber class and the corresponding reflection class ReflectionFiber.
- Adding exception classes FiberError and FiberExit to represent errors.

- Fibers allow for transparent non-blocking I/O implementations of existing interfaces (PSR-7, Doctrine ORM, etc.). That's because the placeholder (promise) object is eliminated. Instead, functions can declare the I/O result type instead of a placeholder object that cannot specify a resolution type because PHP does not support generics.

You can use Fibers to develop full-stack, interruptible PHP functions, which you can then use to implement cooperative multitasking in PHP. As Fibers pause the whole execution stack, you can rest assured knowing that it won't harm the rest of your code.



— Chart illustrating PHP code execution flow with Fibers (Source: PHP.net).

To illustrate Fibers usage, its RFC uses this simple example:

```
$fiber = new Fiber(function (): void {
    $value = Fiber::suspend('fiber');
    echo "Value used to resume fiber: ", $value, "\n";
});

$value = $fiber->start();

echo "Value from fiber suspending: ", $value, "\n";

$fiber->resume('test');
```

You're creating a "fiber" in the above code and immediately suspending it with the string `fiber`. The `echo` statement serves as a visual cue for the fiber's resumption.

You can retrieve this string value from the call to `$fiber->start()`.

Then, you resume the fiber with the string "test," which is returned from the call to `Fiber::suspend()`. The full code execution results in an output that reads:

```
Value from fiber suspending: fiber
Value used to resume fiber: test
```

That's the barebones textbook example of PHP Fibers at work. Here's another Fibers example of performing seven asynchronous GET requests.

With all being said and done, most PHP developers will never deal with Fibers directly. And the RFC even suggests the same:

> *Fibers are an advanced feature that most users will not use directly. This feature is primarily targeted at library and framework authors to provide an event loop and an asynchronous programming API. Fibers allow integrating asynchronous code execution seamlessly into synchronous code at any point without the need to modify the application call stack or add boilerplate code.*
>
> *The Fiber API is not expected to be used directly in application-level code. Fibers provide a basic, low-level flow-control API to create higher-level abstractions that are then used in application code.*

Considering its performance benefits, you can expect PHP libraries and frameworks to take advantage of this new feature. It'll be interesting to see how they implement Fibers within their ecosystem.

New `readonly` Properties

PHP 8.1 adds support for `readonly` properties. They can only be initialized once from the scope where they're declared. Once initialized, you cannot modify their value ever. Doing so would throw up an **Error** exception.

Its RFC reads:

> *A **readonly** property can only be initialized once, and only from the scope where it has been declared. Any other assignment or modification of the property will result in an Error exception.*

Here's an example of how you can use it:

```
class Test {
    public readonly string $kinsta;
```

```
        public function __construct(string $kinsta) {
            // Legal initialization.
            $this->kinsta = $kinsta;
        }
    }
```

Once initialized, there's no going back. Having this feature baked into PHP greatly reduces boilerplate code that's often used to enable this functionality.

The `readonly` property offers a strong immutability guarantee, both inside and outside the class. It doesn't matter what code runs in between. Calling a `readonly` property will always return the same value.

However, using the `readonly` property may not be ideal in specific use cases. For example, you can only use them alongside a typed property because declarations without a type are implicitly `null` and cannot be `readonly`.

Furthermore, setting a `readonly` property does not make objects immutable. The `readonly` property will hold the same object, but that object itself can change.

Another minor issue with this property is that you cannot clone it. There's already a workaround for this particular use case. Look into it if necessary.

Define `final` Class Constants

As of PHP 8.0, you can override class constants with its child classes. It's due to the way inheritance is implemented in PHP.

Here's an example of how you can override a previously declared constant's value:

```
class Moo
{
    public const M = "moo";
}

class Meow extends Moo
{
    public const M = "meow";
}
```

Now, if the cows want to get stricter with how the cats behaved (at least with constants), they can do so with PHP 8.1's new `final` modifier.

Once you've declared a constant as `final`, it means that.

```
class Moo
{
    final public const M = "moo";
}

class Meow extends Moo
{
    public const M = "meow";
}

// Fatal error: Meow::M cannot override final constant Moo::M
```

You can read more about it in the final class constants PHP RFC.

New `fsync()` and `fdatasync()` Functions

PHP 8.1 adds two new file system functions named `fsync()` and `fdatasync()`. They'll seem familiar for those used to Linux functions of the same name. That's because they're related, just implemented for PHP.

In fact, this addition has been long overdue. PHP is one of the few major programming languages that still didn't implement fsync() and fdatasync() — that is, until PHP 8.1.

The `fsync()` function is similar to PHP's existing `fflush()` function, but it differs significantly in one way. Whereas `fflush()` flushes the application's internal buffers to the OS, `fsync()` goes one step further and ensures that the internal buffers are flushed to the physical storage. That ensures a complete and persistent write so that you can retrieve data even after an application or system crash.

Here's an example of how you can use it.

```
$doc = 'kinsta.txt';

$kin = fopen($doc, 'ki');
fwrite($kin, 'doc info');
fwrite($kin, "\r\n");
fwrite($kin, 'more info');

fsync($kin);
fclose($kin);
```

Adding the `fsync()` call at the end ensures that any data held in PHP's or the OS's internal buffer gets written to storage. All other code executions are blocked until then.

Its related function is `fdatasync()`. Use it to sync data but not necessarily metadata. For data whose metadata isn't essential, this function call makes the writing process a tad faster.

However, you should note that PHP 8.1 doesn't support `fdatasync()` fully on Windows yet. It merely acts as an alias of `fsync()`. On POSIX, `fdatasync()` is properly implemented.

New `array_is_list()` Function

PHP arrays can hold both integer and string keys. That means you can use them for several things, including lists, hash tables, dictionaries, collections, stacks, queues, and much more. You can even have arrays within arrays, creating multidimensional arrays.

You can efficiently check whether a particular entry is an array, but it's not that easy to check whether it has any missing array offsets, out-of-order keys, etc. In short, you cannot verify quickly whether an array is a list.

The array_is_list() function checks whether an array's keys are in sequential order starting from `0`, and with no gaps. If all the conditions are met, it'll return `true`. By default, it also returns `true` for empty arrays.

Here are a few examples of using it with both `true` and `false` conditions met:

```
// true array_is_list() examples
array_is_list([]); // true
array_is_list([1, 2, 3]); // true
array_is_list(['cats', 2, 3]); // true
array_is_list(['cats', 'dogs']); // true
array_is_list([0 => 'cats', 'dogs']); // true
array_is_list([0 => 'cats', 1 => 'dogs']); // true

// false array_is_list() examples
array_is_list([1 => 'cats', 'dogs']); // as first key isn't 0
array_is_list([1 => 'cats', 0 => 'dogs']); // keys are out of order
array_is_list([0 => 'cats', 'bark' => 'dogs']); // non-integer keys
array_is_list([0 => 'cats', 2 => 'dogs']); // gap in between keys
```

A PHP array list with out-of-order keys is a potential source of bugs. Using this function to enforce a strict adherence to **list** requirements before moving ahead with code execution is a great addition to PHP.

New Sodium XChaCha20 Functions

Sodium is a modern, easy-to-use cryptographic library for encryption, decryption, password hashing, signatures, and more. The PECL libsodium package adds a wrapper for Sodium so that PHP developers can use it.

Even leading tech companies like Facebook, Discord, Malwarebytes, and Valve use libsodium to secure their users with fast and safe connections.

libsodium supports the XChaCha20 encryption algorithm to encrypt and decrypt data, especially for stream encryption. Likewise, the PECL libsodium extension already supports XChaCha20, but only with Poly1305 message-authentication code.

Many PHP applications use XChaCha20 directly for stream encryption. To make things easier, starting with PHP 8.1, you'll have three new functions to encrypt or decrypt data with XChaCha20 without authentication involved. This mode is called "detached mode."

The newly introduced XChaCha20 functions are:

- `sodium_crypto_stream_xchacha20_keygen`: Returns a secure random key for use with sodium_crypto_stream_xchacha20.
- `sodium_crypto_stream_xchacha20`: Expands the key and nonce into a keystream of pseudorandom bytes.
- `sodium_crypto_stream_xchacha20_xor`: Encrypts a message using a nonce and a secret key (no authentication).

Additionally, there are two new PHP constants defined in the global namespace:

- `SODIUM_CRYPTO_STREAM_XCHACHA20_KEYBYTES` (assigned **32**)
- `SODIUM_CRYPTO_STREAM_XCHACHA20_NONCEBYTES` (assigned **24**)

Use it with caution, though. Since it's without authentication, the decryption operation is vulnerable to common ciphertext attacks.

You can read more about its usage and requirements on the GitHub page.

New IntlDatePatternGenerator Class

PHP's underlying ICU library supports the creation of localized date and time formats, but it's not fully customizable.

For example, if you want to create locale-specific data and time formats until PHP 8.0, you could use the predefined IntlDateFormatter constant to do it in 6 ways:

- **`IntlDateFormatter::LONG`:** Longer, like November 10**, 2017** or **11:22:33pm**
- **`IntlDateFormatter::MEDIUM`:** A bit shorter, like November 10**, 2017**
- **`IntlDateFormatter::SHORT`:** Just numeric, like **10/11/17** or **11:22pm**

Each of these also has its own `RELATIVE_` variants, which sets the date formatting within a limited range before or after the current date. In PHP, the values are **yesterday**, **today**, and **tomorrow**.

Say you want to use the long version for the year and the short version for the month, like **10/11/2017**. As of PHP 8.0, you cannot.

In PHP 8.1+, you can specify which formats to use for the date, month, and time with the new **IntlDatePatternGenerator** class. You can leave the exact ordering of these components to the formatter.

You should note that while this class only has the word **Date** in it, it's consistent with ICU's **DateTimePatternGenerator**. That means you can also use it to create flexible time formats. To simplify naming, the PHP team has chosen to go with the shorter **IntlDatePatternGenerator** term.

Here's an example straight from its RFC:

```
$skeleton = "YYYYMMdd";

$today = \DateTimeImmutable::createFromFormat('Y-m-d', '2021-04-24');

$dtpg = new \IntlDatePatternGenerator("de_DE");
$pattern = $dtpg->getBestPattern($skeleton);
echo "de: ", \IntlDateFormatter::formatObject($today, $pattern, "de_DE"),

$dtpg = new \IntlDatePatternGenerator("en_US");
$pattern = $dtpg->getBestPattern($skeleton), "\n";
echo "en: ", \IntlDateFormatter::formatObject($today, $pattern, "en_US"),

/*
de: 24.04.2021
en: 04/24/2021
*/
```

In the above code, the **skeleton** variable defines the particular date or time formats to use. However, the formatter handles the final result's order.

Support for AVIF Image Format

AVIF, or AV1 Image File Format, is a relatively new royalty-free image format based on the AV1 video coding format. Apart from offering higher compression (and thus smaller file sizes), it also supports several features such as transparency, HDR, and more.

The AVIF format was only standardized recently (June 8, 2021). That has paved the way for browsers, such as Chrome 85+ and Firefox 86+, adding support for AVIF images.

PHP 8.1's image processing and GD extension adds support for AVIF images.

However, to include this functionality, you need to compile the GD extension with AVIF support. You can do so by running the commands below.

For Debian/Ubuntu:

```
apt install libavif-dev
```

For Fedora/RHEL:

```
dnf install libavif-devel
```

That'll install all the latest dependencies. Next, you can compile the AVIF support by running the `--with-avif` flag with the `./configure` script.

```
./buildconf --force
./configure --enable-gd --with-avif
```

If you're starting a new environment from scratch, you can also enable other PHP extensions here.

Once installed, you can test whether AVIF support is enabled by running the following command in your PHP terminal:

```
php -i | grep AVIF
```

If you've installed AVIF correctly, you'll see the following result:

```
AVIF Support => enabled
```

You can also use the `gd_info()` call to retrieve a list of GD features, including whether **AVIF Support** functionality is enabled.

This updated PHP 8.1 GD extension also adds two new functions for working with AVIF images: `imagecreatefromavif` and `imageavif`. They work similarly to their JPEG and PNG counterparts.

The `imagecreatefromavif` function returns a GdImage instance from a given AVIF image. You can then use this instance to edit or convert the image.

The other `imageavif` function outputs the AVIF image file. For instance, you can use it to convert a JPEG to AVIF:

```
$image = imagecreatefromjpeg('image.jpeg');
imageavif($image, 'image.avif');
```

You can read more about this new feature on its GitHub page.

New `$_FILES: full_path` Key for Directory Uploads

PHP maintains a large number of predefined variables to track various things. One of them is the $_FILES variable holding an associative array of items uploaded via the HTTP POST method.

Most modern browsers support uploading an entire directory with HTML file upload fields. Even PHP <8.1 supported this functionality, but with a big caveat. You couldn't upload a folder with its exact directory structure or relative paths because PHP didn't pass this information to the `$_FILES` array.

That changes in PHP 8.1 with the addition of a new key named `full_path` to the `$_FILES` array. Using this new data, you can store relative paths or duplicate the exact directory structure on the server.

You can test this information by outputting the `$FILES` array using the `var_dump($_FILES);` command.

However, proceed with caution if you're using this feature. Make sure that you safeguard against standard file upload attacks.

Array Unpacking Support for String-Keyed Arrays

PHP 7.4 added support for array unpacking with the array spread operator (**...**). It acts as a quicker alternative to using the `array_merge()` function. However, this feature was limited to numeric-keyed arrays as unpacking stringed-key arrays caused conflicts while merging arrays with duplicate keys.

However, PHP 8 added support for named arguments, removing this limitation. Hence, array unpacking will now also support string-keyed arrays using the same syntax:

```
$array = [...$array1, ...$array2];
```

This RFC example illustrates how merging arrays with duplicate string keys is handled in PHP 8.1:

```
$array1 = ["a" => 1];
$array2 = ["a" => 2];
$array = ["a" => 0, ...$array1, ...$array2];
var_dump($array); // ["a" => 2]
```

Here, the string key “a” appears thrice before merging via array unpacking. But only its last value belonging to `$array2` wins.

Explicit Octal Numeral Notation

PHP supports various numeral systems, including decimal (base-10), binary (base-2), octal (base-8), and hex (base-16). The decimal numeral system is the default.

If you want to use any other numeral system, then you’ll have to prefix each number with a standard prefix:

- **Hex:** `0x` prefix. (e.g. 17 = `0x11`)
- **Binary:** `0b` prefix. (e.g. 3 = `0b11`)
- **Octal:** `0` prefix. (e.g. 9 = `011`)

You can see how the octal numeral system’s prefix varies from the rest. To standardize this concern, many programming languages are adding support for an explicit octal numeral notation: `0o` or `0O`.

Starting with PHP 8.1, you can write the example shown above (i.e. number 9 in base-10) in the octal numerical system as `0o11` or `0O11`.

```
0o16 === 14; // true
0o123 === 83; // true

0O16 === 14; // true
0O123 === 83; // true

016 === 0o16; // true
016 === 0O16; // true
```

Furthermore, this new feature also works with the underscore numeric literal separator introduced in PHP 7.4.

Read more about this new PHP 8.1 feature in its RFC.

MurmurHash3 and xxHash Hash Algorithms Support

PHP 8.1 adds support for MurmurHash3 and xxHash hashing algorithms. They're not designed for cryptographic use, but they still provide impressive output randomness, dispersion, and uniqueness.

These new hashing algorithms are faster than most of PHP's existing hashing algorithms. In fact, some of these hashing algorithms' variants are faster than the RAM throughput.

As PHP 8.1 also adds support for declaring algorithm-specific `$options` parameters, you can do the same with these new algorithms. The default value of this new argument is `[]`. So, it won't affect any of our existing hash functions.

You can read more about these new PHP 8.1 features on their GitHub pages: MurmurHash3, xxHash, Algorithm-specific $options.

DNS-over-HTTPS (DoH) Support

DNS-over-HTTPS (DoH) is a protocol for DNS resolution via the HTTPS protocol. Using HTTPS to encrypt data between the client and DNS resolver, DoH increases user privacy and security by preventing MitM attacks.

Starting with PHP 8.1, you can use the Curl extension to specify a DoH server. It requires PHP to be compiled with **libcurl** 7.62+ versions. That's not an issue for most popular operating systems, including Linux distros, as they often include Curl 7.68+.

You can configure the DoH server URL by specifying the `CURLOPT_DOH_URL` option.

```
$doh = curl_init('https://kinsta.com');
curl_setopt($doh, CURLOPT_DOH_URL, 'https://dns.google/dns-query');
curl_exec($doh);
```

In the above example, we've used Google's public DNS server. Also, note the use of `https://` in all the URLs used. Make sure to configure this perfectly as there's no default DNS server to fall back to in Curl.

You can also choose from a list of public DoH servers included in the Curl documentation.

Furthermore, the Curl documentation's CURLOPT_DOH_URL reference explains how to use its various arguments thoroughly.

File Uploads from Strings with CURLStringFile

The PHP Curl extension supports HTTP(S) requests with file uploads. It uses the **CURLFile** class to achieve this, which accepts a URI or a file path, a mime type, and the final file name.

However, with the **CURLFile** class, you can only accept the file path or URI, but not the contents of the file itself. In cases where you already had the file being uploaded in the memory (e.g. processed images, XML documents, PDFs), you had to use `data://` URIs

with Base64 encoding.

But **libcurl** already supports an easier way to accept the file's contents. The new **CURLStringFile** class adds support for precisely that.

You can read its GitHub page to learn more about how it's implemented in PHP 8.1.

New `MYSQLI_REFRESH_REPLICA` Constant

PHP 8.1's **mysqli** extension adds a new constant called `MYSQLI_REFRESH_REPLICA`. It's equivalent to the existing `MYSQLI_REFRESH_SLAVE` constant.

This change was welcome in MySQL 8.0.23 to address racial insensitivity in tech vocabulary (the most common examples include "slave" and "master").

You should note that the older constant isn't being removed or deprecated. Developers and applications can continue using it. This new constant is but an option for developers and companies who wish to leave behind such terminology.

Performance Improvements with Inheritance Cache

Inheritance Cache is a new addition to opcache that eliminates PHP class inheritance overhead.

PHP classes are compiled and cached by opcache separately. However, they're already linked at run-time on each request. This process may involve several compatibility checks and borrowing methods/properties/constants from parent classes and traits.

As a result, this takes considerable time to execute, even though the result is the same for each request.

Inheritance Cache links all unique dependent classes (parent, interfaces, traits, property types, methods) and stores the results in opcache shared memory. As this happens only once now, inheritance requires lesser instructions.

Furthermore, it removes limitations for immutable classes, such as unresolved constants, typed properties, and covariant type checks. Thus, all the classes stored in opcache are

immutable, further reducing the number of instructions required.

All in all, it promises significant performance benefits. Dimitry Stogov, the author of this patch, found that it showed 8% improvement on the base Symfony “Hello, World!” program. We can’t wait to test it out in our following PHP benchmarks.

First-Class Callable Syntax

PHP 8.1 adds a first-class callable syntax to supersede existing encodings using strings and arrays. Besides creating a cleaner **Closure**, this new syntax is also accessible by static analysis tools and respects the declared scope.

Here are a few examples taken from the RFC:

```
$fn = Closure::fromCallable('strlen');
$fn = strlen(...);

$fn = Closure::fromCallable([$this, 'method']);
$fn = $this->method(...)

$fn = Closure::fromCallable([Foo::class, 'method']);
$fn = Foo::method(...);
```

Here, all the expression pairs are equivalent. The triple-dot (**...**) syntax is similar to the argument unpacking syntax (`...$args`). Except here, the arguments are not yet filled in.

Changes in PHP 8.1

PHP 8.1 also includes changes to its existing syntax and features. Let’s discuss them:

PHP Interactive Shell Requires readline Extension

PHP's **readline** extension enables interactive shell features such as navigation, autocompletion, editing, and more. While it's bundled with PHP, it's not enabled by default.

You can access the PHP interactive shell using PHP CLI's `-a` command-line option:

```
php -a

Interactive shell

php >
php > echo "Hello";
Hello
php > function test() {
php { echo "Hello";
php { }
php > test();
Hello
```

Before PHP 8.1, you could open the interactive shell using PHP CLI even without the **readline** extension enabled. As expected, the shell's interactive features didn't work, rendering the `-a` option meaningless.

In PHP 8.1 CLI, the interactive shell exits with an error message if you've not enabled the readline extension.

```
php -a
Interactive shell (-a) requires the readline extension.
```

MySQLi Default Error Mode Set to Exceptions

Before PHP 8.1, MySQLi defaulted to silent the errors. This behavior often led to code that didn't follow strict Error/Exception handling. Developers had to implement their own explicit error handling functions.

PHP 8.1 changes this behavior by setting MySQLi's default error reporting mode to throw an exception.

```
Fatal error: Uncaught mysqli_sql_exception: Connection refused in ...:...
```

As this is a breaking change, for PHP <8.1 versions, you should explicitly set the error handling mode using the `mysqli_report` function before making the first MySQLi connection. Alternatively, you can do the same by selecting the error reporting value by instantiating a `mysqli_driver` instance.

The RFC follows a similar change introduced in PHP 8.0.

Customizable Line Endings for CSV Writing Functions

Before PHP 8.1, PHP's built-in CSV writing functions, `fputcsv` and `SplFileObject::fputcsv`, were hard-coded to add `\n` (or the Line-Feed character) at the end of every line.

PHP 8.1 adds support for a new parameter named `eol` to these functions. You can use it to pass a configurable end-of-line character. By default, it still uses the `\n` character. So, you can continue using it in your existing code.

Standard character escaping rules apply for using end-of-line characters. If you want to use `\r`, `\n`, or `\r\n` as EOL characters, you must enclose them in double quotes.

Here's the GitHub page tracking this new change.

New `version_compare` Operator Restrictions

PHP's `version_compare()` function compares two version number strings. This function accepts an optional third argument called `operator` to test for a particular relationship.

Though not covered explicitly in the documentation, before PHP 8.1, you could set this parameter to a partial value (e.g. `g`, `l`, `n`) without facing an error.

PHP 8.1 adds stricter restrictions to the `version_compare()` function's `operator` argument to overcome this situation. The only operators you can now use are:

- **==**, **=**, and **eq**
- **!=**, **<>**, and **ne**
- **>** and **gt**
- **>=** and **ge**
- **<** and **lt**
- **<=** and **le**

No more partial operator values.

HTML Encoding and Decoding Functions Now Use `ENT_QUOTES | ENT_SUBSTITUTE`

HTML entities are textual representations of characters that would otherwise be interpreted as HTML. Think of characters such as < and > used to define HTML tags (e.g. <a>, <h3>, `<script>`).

The HTML entity for < is `&lt;` (lesser than symbol) and > is `&gt;` (greater than symbol).

Note: Remove the space between "&" and "amp."

You can use these HTML entities safely in an HTML document without triggering the browser's rendering engine.

For instance, `&lt;script&gt;` will show as `<script>` in the browser, rather than being interpreted as an HTML tag.

Prior to PHP 8.1, the htmlspecialchars() and htmlentities() functions converted symbols like ", <, >, and & to their respective HTML entities. But they didn't convert the single quote character (') to its HTML entity by default. Moreover, they returned an empty string if there was a malformed UTF-8 in the text.

In PHP 8.1., these HTML encoding and decoding functions (and their related functions) will also convert single quote characters to their HTML entity by default.

And if the given text has invalid characters, the functions will substitute them with a Unicode substitution character (•) instead of returning an empty string. PHP 8.1 accomplishes this by changing the signatures of these functions to `ENT_QUOTES | ENT_SUBSTITUTE` rather than `ENT_COMPAT` by default.

Most frameworks already use `ENT_QUOTES` as the default flag value. So, you'll not see much difference due to this change. However, the new `ENT_SUBSTITUTE` flag is not that widely used. PHP 8.1 will cause invalid UTF-8 characters to be substituted with the • character instead of returning an empty string.

Warning on Illegal compact Function Calls

PHP's `compact()` function is super handy. You can use it to create an array with variables using their names and values.

For example, consider the following code:

```
$animal = 'Cat';
$sound = 'Meow';
$region = 'Istanbul';
compact('animal', 'sound', 'region');
// ['animal' => "Cat", 'sound' => "Meow", 'region' => "Istanbul"]
```

The compact function's documentation states that it'll only accept string parameters or array values with string values. However, before PHP 7.3, any strings that aren't set would be

silently skipped.

PHP 7.3 modified the `compact()` function to throw up a notice if you use undefined variables. PHP 8.1 takes it a step further and throws up a warning.

You can read its GitHub page to get an understanding of how this change came to be.

New Migrations from Resources to Class Objects

One of PHP's long-term goals is to move away from resources towards standard class objects.

Due to historical reasons, resource objects are used extensively in PHP applications. Hence, the migration of resources to class objects needs to be as less disruptive as possible. PHP 8.1 migrates five such resources:

The `file_info` Resource Migrated to `finfo` Objects

PHP's finfo class offers an object-oriented interface for the `fileinfo` functions. However, using `finfo` functions returns `resource` objects with the `file_info` type rather than an instance of the `finfo` class itself.

PHP 8.1 fixes this anomaly.

IMAP Resources Migrated to `IMAP\Connection` Class Objects

In line with the resource-to-object migration goal, the new `IMAP\Connection` class minimizes potential breaking changes when PHP eventually modifies the class's implementation details.

This new class is also declared `final`, so you're not allowed to `extend` it.

Read more about its implementation on its GitHub page.

FTP Connection Resources Are Now `FTP\Connection` Class Objects

In PHP <8.1, if you create an FTP connection with `ftp_connect()` or `ftp_ssl_connect()` functions, you'd get back a **resource** object of type **ftp**.

PHP 8.1 adds the new `FTP\Connection` class to rectify that. And like with the `IMAP\Connection` class, it's also declared `final` to prevent it from being extended.

Read more about its implementation on its GitHub page.

Font Identifiers Migrated to `GdFont` Class Objects

PHP's GD extension provides the imageloadfont() function to load a user-defined bitmap and return its font-identifier resource ID (an integer).

In PHP 8.1, this function will instead return a **GdFont** class instance. Furthermore, to make the migration hassle-free, all the functions that previously accepted a resource ID from `imageloadfont()` will now take the new **GdFont** class objects.

Read more about this migration on its GitHub page.

LDAP Resources Migrated to Objects

LDAP, or Lightweight Directory Access Protocol, is used to access "Directory Servers." Like a hard disk directory structure, it's a unique database that holds data in a tree structure.

PHP includes an LDAP extension that accepted or returned **resource** objects before PHP 8.1. However, they've all migrated seamlessly to new class instances now. The **resource** types that have been transitioned are:

- `ldap link` resource to `\LDAP\Connection` class object
- `ldap result` resource to `\LDAP\Result` class object
- `ldap result entry` resource to `\LDAP\ResultEntry` class object

Go through its GitHub page to understand this migration better.

Pspell Resources Are Now Class Objects

PHP's Pspell extension allows you to check spellings and word suggestions.

PHP <8.1 used `pspell` and `pspell config` resource object types with an integer identifier. These two resource objects are now replaced with `PSpell\Dictionary` and `PSpell\Config` class objects.

Like previous migrations, all Pspell functions that previously accepted or returned resource object identifiers will take the new class object instances.

Refer to its GitHub page for more information.

Deprecations in PHP 8.1

PHP 8.1 deprecates many of its previous features. The following list provides a brief overview of the functionalities PHP 8.1 deprecates:

Can't Pass `null` to Non-Nullable Internal Function Parameters

As of PHP 8.0, its internal functions silently accept `null` values even for non-nullable arguments. The same doesn't hold for user-defined functions — they only accept `null` for nullable arguments.

For example, consider this usage:

```
var_dump(str_contains("foobar", null));
// bool(true)
```

Here, the `null` value is silently converted to an empty string. Thus, the result returns `true`.

This RFC aims to synchronize the behavior of internal functions by throwing a deprecation warning in PHP 8.1.

```
var_dump(str_contains("foobar", null));
// Deprecated: Passing null to argument of type string is deprecated
```

The deprecation will become a TypeError in the next major PHP version (i.e. PHP >=9.0), making the behavior of internal functions consistent with user-defined functions.

Restricted $GLOBALS Usage

PHP's `$GLOBALS` variable provides a direct reference to its internal symbol table. Supporting this functionality is complex and affects array operations performance. Plus, it was rarely used.

As per the RFC, indirectly modifying `$GLOBALS` is no longer allowed. This change is backward incompatible.

The impact of this change is relatively low:

> *In the top 2k composer packages I found 23 cases that use $GLOBALS without directly dereferecing it. Based on a cursory inspection, there are only two instances where **$GLOBALS** is not used in a read-only way.*

However, read-only usage of `$GLOBALS` continues to work as usual. What's no longer supported is writing to `$GLOBALS` as a whole. As a result, you can expect a slight performance bump, especially when working with ordinary PHP arrays.

Return Type Declarations for Internal Functions

PHP 8.0 allowed developers to declare parameters and return types for most internal functions and methods. It was possible thanks to various RFCs such as Consistent type errors for internal functions, Union Types 2.0, and Mixed Type v2.

However, there are many cases where type information can be missing. Some of them include a type with resources, **out** pass-by-ref parameters, return type of non-final methods, and functions or methods that don't parse parameters according to general rules. You can read the exact details in its RFC.

This RFC only addresses the issue with non-final methods' return type. However, rather than phasing it out altogether immediately, the PHP team provides a gradual migration path to update your codebases with the relevant method return types.

Non-final internal method return types – when possible – are declared tentatively in PHP 8.1, and they will become enforced in PHP 9.0. It means that in PHP 8.x versions, a "deprecated" notice is raised during inheritance checks when an internal method is overridden in a way that the return types are incompatible, and PHP 9.0 will make these a fatal error.

If you see this deprecation notice after updating to PHP 8.1, make sure to update your methods' return types.

Serializable Interface Deprecated

PHP 7.4 introduced the custom object serialization mechanism with two new magic methods: `__serialize()` and `__unserialize()`. These new methods aim to replace the broken **Serializable** interface eventually.

This RFC proposes to finalize that decision by laying out a plan for the eventual removal of **Serializable**.

In PHP 8.1, if you implement the **Serializable** interface without implementing `__serialize()` and `__unserialize()` methods, PHP will throw a "Deprecated" warning.

```
Deprecated: The Serializable interface is deprecated. Implement __seriali
```

If you're supporting **PHP <7.4** and **PHP >=7.4**, you should implement both the **Serializable** interface and the new magic methods. On **PHP >=7.4** versions, the magic methods will take precedence.

Non-Compatible `float` to `int` Conversions Deprecated

PHP is a dynamically typed language. As such, there are many cases where type coercion naturally occurs. Most of these coercions are harmless and super convenient.

However, when a **float** number is converted to an **integer**, it often involves data loss. For example, when the float **3.14** is converted to an integer **3**, it loses its fractional value.

The same happens when the float is outside the platform integer range, or when a float string is converted to an integer.

PHP 8.1 rectifies this behavior and brings its dynamic type coercing more in line with most modern programming languages. The goal is to make such coercions predictable and intuitive.

In PHP 8.1, you'll see a deprecation notice when a non-compatible **float** is implicitly coerced to an **int**. But what constitutes an integer-compatible float? The RFC answers this:

A float is said to be integer-compatible if posses the following characteristics:

- *Is a number (i.e. not NaN or Infinity)*
- *Is in range of a PHP integer (platform dependent)*
- *Has no fractional part*

This deprecation notice will upgrade to a **TypeError** in the next major PHP version (i.e. PHP 9.0).

The `mysqli::get_client_info` Method and `mysqli_get_client_info($param)` Deprecated

The MySQL client API defines two constants: `client_info` (a string) and `client_version` (an int). MySQL Native Driver (MySQLnd) is part of the official PHP source and pegs these constants to the PHP version. In libmysql, they represent the client library version at the time of compilation.

Before PHP 8.1, mysqli was exposing these constants in 4 ways: `mysqli_driver` properties, `mysqli` **`properties`**, `mysqli_get_client_info()` function, and `mysqli::get_client_info` method. Though, there's no method for `client_version`.

MySQLnd exposes these constants in 2 ways to PHP: a constant and a function call. To unify mysqli access methods with these same two options, PHP 8.1 is deprecating these other two options:

- `get_client_info` method in the **mysqli** class. Instead, you can just use the `mysqli_get_client_info()` function.
- `mysqli_get_client_info()` function with parameters. Call the function without any parameters to avoid the deprecation notice.

Read more about this deprecation on its GitHub page.

All `mhash*()` Functions (hash Extension) Are Deprecated

PHP 5.3 integrated `mhash*()` functions into `ext/hash` as a compatibility layer for `ext/mhash`. Later, PHP 7.0 removed `ext/mhash`.

Unlike the `hash_*()` functions, the `mhash*()` functions aren't always available. You have to enable them separately while configuring PHP.

In PHP 7.4, the hash extension was bundled along with PHP, making it a default extension for PHP. However, it still supported enabling the `--enable-mhash` option for compatibility reasons.

The PHP team has decided to deprecate mhash*() functions in PHP 8.1, and remove them altogether in PHP 9.0. The functions deprecated are `mhash()`, `mhash_keygen_s2k()`, `mhash_count()`, `mhash_get_block_size()` and `mhash_get_hash_name()`. You can use the standard `ext/hash` functionality in place of them.

Both `filter.default` and `filter.default_options` INI Settings Deprecated

PHP's `filter.default` INI settings allows you to apply a filter to all PHP super-globals — i.e. GPCRS data (`$_GET`, `$_POST`, `$_COOKIE`, `$_REQUEST`, and `$_SERVER`).

For example, you can set `filter.default=magic_quotes` or `filter.default=add_slashes` (based on PHP version) to resurrect PHP's controversial and insecure magic quotes feature (removed in PHP 5.4).

The `filter.default` INI setting provides additional functionality by allowing many more filters, making it even worse. For example, its another option — `filter.default=special_chars` — enables magic quotes only for HTML. There's much less awareness of these settings.

PHP 8.1 will throw a deprecation warning if `filter.default` is set to any value other than `unsafe_raw` (the default). You'll see no separate deprecation notice for `filter.default_options`, but PHP 9.0 will remove both these INI settings.

As an alternative, you can start using the filter_var() function. It filters variables with the specified filter.

Deprecate `autovivification` on false

PHP allows for autovivification (auto-creation of arrays from false values). This feature is super helpful if the variable is undefined.

Nonetheless, it's not ideal to auto-create an array when the value is false or null.

This RFC disallows autovivification from false values. However, note that autovivification from undefined variables and null is still permitted.

In PHP 8.1, appending to a variable of type false will emit a deprecation notice:

```
Deprecated: Automatic conversion of false to array is deprecated in
```

PHP 9.0 will throw a fatal error for the same, which is identical to other scalar types.

The `mysqli_driver->driver_version` Property Is Deprecated

MySQLi extension's **mysqli_driver->driver_version** property hasn't been updated for 13 years. Despite many changes to the driver since then, it still returns the old driver version value, making this property meaningless.

In PHP 8.1, the mysqli_driver->driver_version property is deprecated.

Other Minor Changes

There are many more deprecations in PHP 8.1. Listing them all here will be an exhausting exercise. We recommend you directly check the RFC for these minor deprecations.4

PHP's GitHub page also includes a PHP 8.1 UPGRADE NOTES guide. It lists all the breaking changes you should consider before upgrading to PHP 8.1.

Summary

PHP 8.1 is better than its predecessor, which is no small feat. We think the most exciting PHP 8.1 features are Enums, Fibers, Pure Intersection Types, and its many performance improvements. Also, we can't wait to put PHP 8.1 through its paces and benchmark various PHP frameworks and CMSs.

Make sure to bookmark this blog post for your future reference.

Which PHP 8.1 feature is your favorite? Share your thoughts with the community in the comments section below.