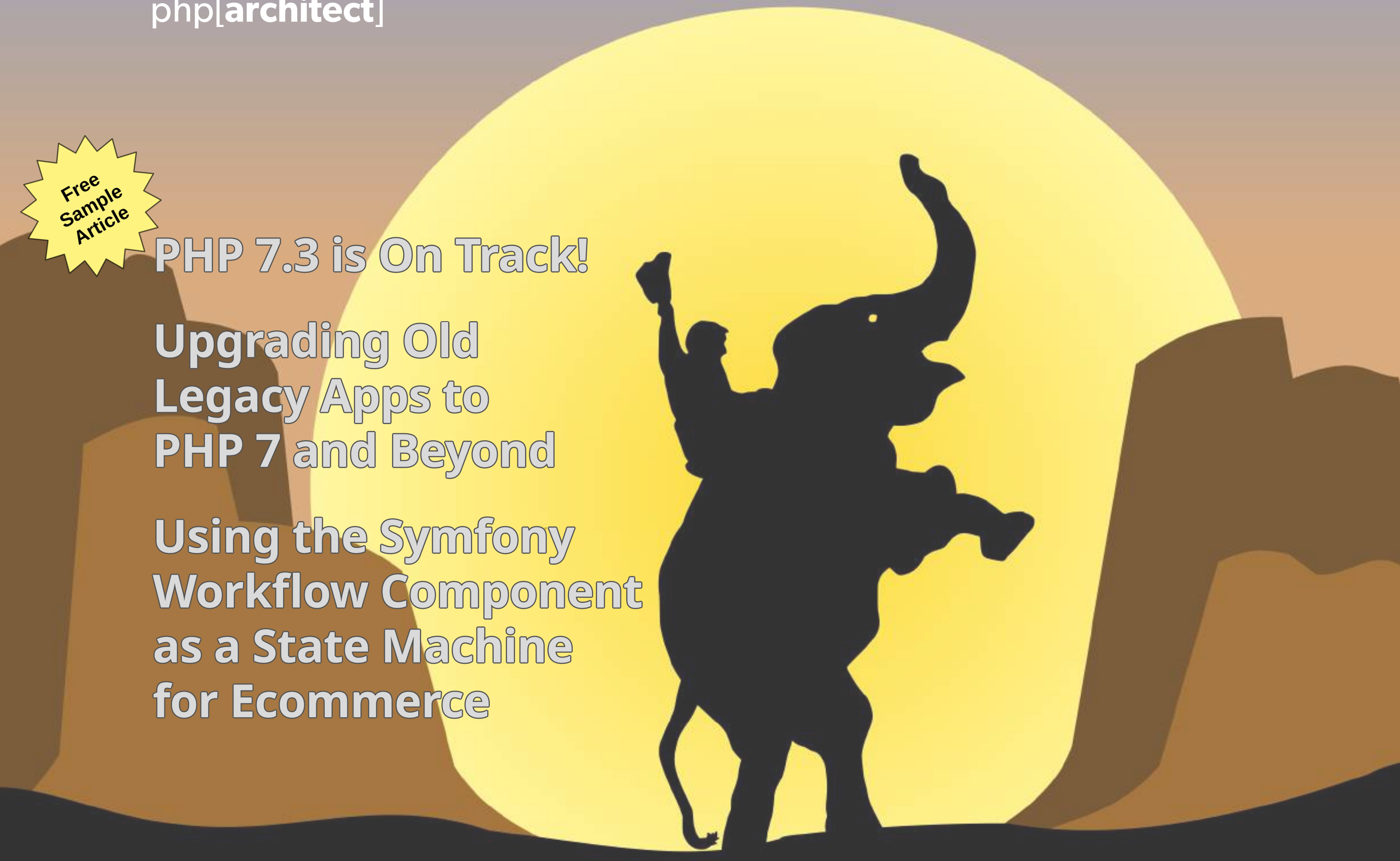
www.phparch.com

September 2018
Volume 17 - Issue 9

php[architect]

MagniPHPicent 7.3



Free Sample Article

PHP 7.3 is On Track!

Upgrading Old Legacy Apps to PHP 7 and Beyond

Using the Symfony Workflow Component as a State Machine for Ecommerce

AUTOMATTIC
RIPS
thermo
nexmo
twilio
PHP[WORLD]

I work on performance at Slack
PHP[WORLD]
PHP[WORLD]

CONTENTS

php[architect]

SEPTEMBER 2018
Volume 17 - Issue 9

Features

Columns

Editor-in-Chief: Oscar Merida

Editor: Kara Ferguson

Managing Partners

Oscar Merida, Sandy Smith

php[**architect**] is published twelve times a year by: musketeers.me, LLC
201 Adams Avenue
Alexandria, VA 22301, USA

Subscriptions

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

Advertising

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

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

Print ISSN 1709-7169
Digital ISSN 2375-3544

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

PHP 7.3 is On Track!

Damien Seguy

PHP 7.3 successfully passed the “feature freeze” deadline. On Aug. 1st, 2018 all features for PHP 7.3 were identified. This triggered the first PHP 7.3 beta, on the following day, and, from there, we’ll reach RC in September. It is time to review what this new PHP version has available for us, help test PHP 7.3, and get ready.

Improved Garbage Collector

One of the main improvement in the PHP 7.3 engine is the garbage collector, also called GC.

The Garbage Collector is an internal tool that frees memory. PHP accumulates new objects in memory, and when it reaches the infamous `memory_limit`, the garbage collector is fired to check if any memory may be recycled. Then, PHP resumes the normal execution of the script.

Most traditional PHP applications have no use for the Garbage Collector. First, `memory_limit` is usually far beyond what a script needs, so there is no need to collect memory; PHP frees it all at the end of the execution. Secondly, the GC only works on roots, which are large dynamical structures, such as arrays and objects. Also, the script needs to allocate ten thousand (10,000) of them, literally.

When PHP reaches the 10,000 limit, the GC is triggered. It will be triggered more frequently, as long as it stays above the limit (in particular, when leaving a context: a function, a method, a closure). Until now, any application that went beyond 10,000 would experience a lot of GC calls, and as such, a sudden degradation of its performances.

With PHP 7.3, the GC is now significantly more efficient. If it can’t reasonably free enough memory, it raises the limit. This prevents many inefficient collections and keeps script execution fast.

Applications that generate a lot of objects, like long-running CLI applications, event-driven, or framework-based applications will benefit from this improvement. Yet, most of applications will never come near the limit, and won’t feel a difference.

More on this subject:

- What About Garbage?[1]
- How to optimize the PHP garbage collector usage to improve memory and performance?[2]
- Improvements to Garbage Collection (GC) in PHP 7.3, 5x boost performance in tests[3](GC) in PHP 7.3, 5x boost performance in tests: https://phpa.me/react-etc-gc-php73]

Relaxed Heredoc/Nowdoc

Heredoc and nowdoc are a string definition syntax that is adapted to large pieces of text. Here is a heredoc:

```
$x = <<<FRENCH
Maître Corbeau, sur un arbre perché,
Tenait en son bec un fromage.
Maître Renard, par l'odeur alléché,
Lui tint à peu près ce langage:
Et bonjour, Monsieur du Corbeau,...
FRENCH;
```

The syntax starts with a triple <, followed by a token. This token is a classic PHP identifier: alphanumeric chars and underscore. It must also start with a letter or an underscore. The identifier is free. Here, we used it to comment on the language being used. Other variations include `SQL`, `HTML`, `GREMLIN`, `XML`, `PHP`, `YAML`, `DOT`, etc.

The final identifier comes with a few restrictions; it should be the first on its line, and only accepts a semi-colon with it. Not even a space should be found on that line, or you’ll end up with a `syntax error, unexpected end of file`.

Of course, the ending delimiter shouldn’t be found inside the text, at the beginning of a new line. This is quite rare, so it is really difficult to debug. When that happens, just make the identifier longer.

Nowdoc and heredoc are close cousins: heredoc behaves like a double-quote string and interpolates variables inside single-dimensional arrays and properties. NowDoc encloses the identifier in single quotes, and behaves like a single quote string: no string interpolation is available. See Listing 1.

Listing 1

```
<?php
$animal = 'Corbeau';
// Heredoc
$x = <<<FRENCH
Maître $animal, sur un arbre perché,
FRENCH;

// Nowdoc
$x = <<<'FRENCH'
Maître $animal, sur un arbre perché,
FRENCH;
```

[1] What About Garbage?: https://phpa.me/ircmaxell-about-garbage

[2] How to optimize the PHP garbage collector usage to improve memory and performance?: https://phpa.me/tideways-optimize-gc

[3] Improvements to Garbage Collection (GC) in PHP 7.3, 5x boost performance in tests

In PHP 7.3, two constraints are relaxed: first, the ending delimiter may be freely followed by other operators. So, it is now possible to use heredoc in a function call.

```
$variable = 'THINGS';
print strtolower(<<<ENGLISH
ALL THOSE $variable
ENGLISH);
```

Secondly, the ending delimiter may be moved to the left. There is no need for it to be the first on its line. Better, the indentation of that ending delimiter is the indentation for the text in the Heredoc syntax. Look at that:

```
function foo() {
    return <<<MESSAGE
    Returned Message
    MESSAGE;
}

print foo();
// prints 'Returned Message'
```

The important point here is to use the same indentation for the delimiter and for every line in the text. PHP identifies the indentation on the final delimiter and uses it.

Also, to keep everyone happy, spaces and tabulations are both supported for indentation. Could it be otherwise? The rule is simple; stick to one of them. `Invalid indentation - tabs and spaces cannot be mixed` is an official PHP error message.

More on this: Flexible Heredoc and Nowdoc Syntaxes[4]

Trailing Comma for Calls

Another syntax upgrade for PHP 7.3 is the trailing comma. You may have already met this comma in array definitions or in grouped namespaces. It is now available for every function and method call. The final empty slot won't be send to the called method, it is simply ignored as shown in Listing 2.

The rationale behind this syntax is to create smaller diffs when committing the code to a VCS. With the final comma, adding a new argument to a function call (here, the sixth argument is the `secure` argument, and should always be used), will create a one-line diff.

```
>            1,
```

instead of a two-line diff:

```
>            "example.com",
>            1
```

Note that function definitions don't get that syntax.

More on this: Allow a trailing comma in function calls[5]

Deprecated Case-insensitive Constants

Constants defined with the `define` function may be case-insensitive. This behavior is now deprecated in PHP 7.3 and will be completely removed in PHP 8.0.

`null`, `true` and `false` are exempted from this deprecation. So are the magic constants `__function__` or `__TRAIT__`, although it was not explicitly mentioned in the RFC.

```
// This triggers an error in PHP 7.3
define('FOO', 1, true);

echo FOO;

// This triggers another error in PHP 7.3
echo foo;
```

Removing this support leads to cleaner code in line with PHP common usage. It also reduces complexity in the PHP engine and prevents some bugs.

In a quick survey of 700 applications, we found 2 percent of PHP applications use case-insensitive constants. This feature was rarely used, yet, for those who do, the impact shall be significant.

Listing 2

```
$value = 'something from somewhere';

setcookie("TestCookie",
           $value,
           time()+3600,
           "/~rasmus/",
           "example.com",
           1,
         );
```

More on this: Deprecate and Remove Case-Insensitive Constants[6].

PCRE 2.0

PHP's regex are based on an independent library, called PCRE[7]. PHP has been using the PCRE library for ages. It has made its way to the core of PHP, and can't be disabled anymore. Yet, PHP still uses the version 1.0 of the PCRE library. It has been abandoned, and a new version, PCRE2, was released in 2015. This new version is actively developed.

The change from PCRE 1 to 2 means the API for the library has changed. This only impacts internals, and Anatol Belski did a great job an integrating this new version in the heart of PHP.

As for userland code, the new version means regex may be impacted. There are several places where impact may be felt:

- Modifier S[8] is now on by default. PCRE does some extra optimization.
- Option X is disabled by default. It makes PCRE do more syntax validation than before.
- Unicode 10 is used, while it was Unicode 7. This means more emojis, more characters, and more sets. Unicode regex may be impacted.
- Some invalid patterns may be impacted.

Since PHP doesn't lint regex at linting time, the best is to do an inventory of all the regex in use in the application and test them with PHP 7.3 (aka, PCRE 2.0). You can see the regex inventory of CodeIgniter[9] here.

Then, using `null` as haystack, PHP compiles the regex and reports any error with a `false` and a warning. We are running an arbitrary regex into

[4] *Flexible Heredoc and Nowdoc Syntaxes:* *https://phpa.me/php73-rc-flexdoc*

[5] *Allow a trailing comma in function calls:* *https://phpa.me/php73-trailing-comma*

[6] *Deprecate and Remove Case-Insensitive Constants: https://phpa.me/php73-ci-const*

[7] *PCRE: https://www.pcre.org*

[8] *Modifier S:* *http://php.net/reference.pcre.pattern.modifiers*

[9] *CodeIgniter:* *https://phpa.me/exakat-regex-ci*

`preg_match`, and begging it to raise an error, which we can suppress with an `@`.

```
// since we hunt for regex error, let's not log them
if (false === @preg_match($regex, null)) {
  print "$regex is an invalid regex in pcre\n. Error : "
        . error_get_last() . PHP_EOL;
}
```

More on this: PCRE2 migration[10]

SQLite 3.24

SQLite is another independent library embedded in PHP. It now supports version 3.24[11], which was released in June 2018.

This new version adds support for `UPSERT`. `UPSERT` is inspired by the same command in PostgreSQL. An `UPSERT` is an ordinary `INSERT` statement that is followed by the special ON CONFLICT clause: if the row being inserted is not available in the table, it is inserted. If it is already there, the insert is turned into an UPDATE command.

This is close to the `REPLACE` command, which is already available in SQLite. The main difference is the UPDATE doesn't have to change every column and may decide to preserve some of them.

Here is an example, taken from the UPSERT[12] documentation:

```
CREATE TABLE vocabulary(
   word TEXT PRIMARY KEY, count INT DEFAULT 1
);
INSERT INTO vocabulary(word) VALUES('jovial')
  ON CONFLICT(word) DO UPDATE SET count=count+1;
```

A word is inserted in the `vocabulary` table, and the column `count` keeps track of the number of its insertions.

More on this: version 3.24[13].

Json_encode May Throw Exceptions

PHP has two functions for dealing with JSON: `json_decode()` and `json_encode()`. Unfortunately, they both return `null` when an error happens while processing the data. Yet, `null` is a possible valid result when decoding a JSON. For example, the string `"null"` will be decoded into `null` (the constant).

When decoding `null` is part of the business logic, there is no other way than checking `json_last_error()` for errors before continuing.

PHP 7.3 introduces a new option for the two functions: `JSON_THROW_ON_ERROR`. This makes json_* throw an exception when an error happens, and it may be cleanly caught with a `try/catch` clause.

```
try {
   $read = json_decode(
       $data, false, 512, JSON_THROW_ON_ERROR
   );
} catch (JsonException $e) {
   echo "the incoming data are not valid json code\n";
}
```

The default behavior is to keep the old behavior, for backward compatibility. It is recommended to make a call to `json_last_error()` after calls to `json_encode()` and `json_decode()`.

More on this: JSON_THROW_ON_ERROR[14].

array_first_key(), array_last_key()

Finding the first key of an array required some workaround in PHP 7.2 and older. While it is easy to know what the first element of an auto-generated list is, (hint: it is 0) it requires some work to find the first key of an array.

Various workarounds are shown in Listing 3.

Starting in PHP 7.3, it is possible to use `array_first_key()` to reach that value, without resetting the internal pointer, nor starting a loop on the array.

The same applies to `array_last_key()`, which targets the last element of an array. Solutions to that thorny problem were even more creative than the previous (Listing 4).

Listing 3

```
$array = ['a' = > 1, 'b' => 2, 'c' => 3];

// solution 1 :
reset($array);
$key = key($array);

// solution 2 :
$key = array_keys($array)[0];

// solution 3 :
foreach($array as $key => $value) {
    break 1;
}
```

Listing 4

```
$array = ['a' = > 1, 'b' => 2, 'c' => 3];

// solution 1 :
reset($array);
end($array);
$key = key($array);

// solution 2 :
$key = array_keys($array)[count($array) - 1];

// solution 3 :
foreach ($array as $key => $value) {
    // Actually found in code...
}
```

[10] *PCRE2 migration: https://wiki.php.net/rfc/pcre2-migration*

[11] *version 3.24: https://www.sqlite.org/releaselog/3_24_0.html*

[12] *UPSERT: https://www.sqlite.org/lang_UPSERT.html*

[13] *version 3.24: https://www.sqlite.org/releaselog/3_24_0.html*

[14] *JSON_THROW_ON_ERROR: https://wiki.php.net/rfc/json_throw_on_error*

`array_value_first()` and `array_value_last()` were also part of the RFC, but once the first and final key is found, its value is one dereferencing away.

More on this: int the array_key_first, array_key_last, etc RFC[15]

list() with References

PHP has had the `list()` function, and its short syntax version `[]`. Until PHP 7.3, it was not possible to assign references with `list()`. This is now possible.

```
$array = [1, 2];
list($a, &$b) = $array;
```

This is the same as:

```
$array = [1, 2];
$a = $array[0];
$b =& $array[1];

// and also

[$a, &$b] = $array;
```

As usual, `list()` may be combined with `foreach()`. For example, this will only set ‘c’ index to 7.

```
$array = [['c' => 1, 2], ['c' => 3, 4], ['c' => 5, 6]];
foreach ($array as list('c' => &$a, 1 => $b)) {
   $a = 7;
}
print_r($array);
```

More on this: list Reference Assignment[16]

is_countable()

This new function aims at identifying easily data that may passed to `count()`. Since `count()` yields a warning if fed with wrong data:

```
Parameter must be an array or an object
that implements Countable
```

`is_countable()` is here to provide protection. The function is syntactic sugar: it replaces the following logical construct:

```
if (is_array($foo) || $foo instanceof Countable) {
    // $foo is countable
}
```

More on this: is_countable[17]

net_get_interfaces()

`net_get_interfaces()` is a new function which lists all network interfaces. Until now, it was necessary to rely on `php_exec()` to poke the system, then parse its results. Now, `net_get_interfaces()` provides the same information in an array, directly inside PHP. It has been ported to all available OS, including Windows.

The result of a call to this function may look like Output 1.

More on this: getting ip for eth0[18]

Removing image2wbmp()

`image2wbmp()` was removed from the PHP API. It is used to produce WBMP pictures, and a twin function called `imagewbmp()`. The latter is still available in PHP 7.3.

This function was identified as a duplicate feature of

Output 1

```
Array
(
  [lo0] => Array
      (
          [unicast] => Array
              (
                  [0] => Array
                      (
                          [flags] => 32841
                          [family] => 18
                      )

                  [1] => Array
                      (
                          [flags] => 32841
                          [family] => 30
                          [address] => ::1
                          [netmask] => ffff:ffff:ffff:ffff:
                      )

                  [2] => Array
                      (
                          [flags] => 32841
                          [family] => 2
                          [address] => 127.0.0.1
                          [netmask] => 255.0.0.0
                      )

                  [3] => Array
                        (
                          [flags] => 32841
                          [family] => 30
                          [address] => fe80::1
                          [netmask] => ffff:ffff:ffff:ffff::
                      )


              )

```

See code archive for complete listing

[15] *array_key_first, array_key_last, etc RFC:* *https://wiki.php.net/rfc/array_key_first_last*

[16] *list Reference Assignment:* *https://wiki.php.net/rfc/list_reference_assignment*

[17] *is_countable: https://wiki.php.net/rfc/is-countable*

[18] *getting ip for eth0: https://bugs.php.net/bug.php?id=17400*

`imagewbmp()`, and removed. The later function is still available.

assert() is Now a Reserved Function

`assert`[19] is already a PHP function. It belongs to the global namespace and checks if an assertion is true (obvious, isn't it?). It is currently not possible to define an `assert` function in the global namespace, since the native one is already there.

A problem arises when an `assert` function is created in a namespace and is called as an unqualified name. Basically, just like `assert( )`.

When disabling the assertions with `zend.assertions=0` or assert_options[20], PHP prevents any call to `assert`. This includes namespaced `assert`, as the fully qualified name is reduced to the bare minimum. `assert` is not a function anymore, but a language construct, so reserving it is probably a good idea.

More on this: Deprecations for PHP 7.3[21]

Continue for Loops, Break for Switch

One may have noticed PHP has two very similar keywords: `continue` and `break`. They both break a control flow structure, and they actually may be used interchangeably. They should have distinct usage, as explained by Nikita Popov in the RFC related to this.

```
while ($foo) {
   switch ($bar) {
      case "baz":
         continue; // In PHP: Behaves like "break;"
                   // In C:   Behaves like "continue 2;"
   }
}
```

`continue` and `break` behave the same, while other languages, such as C, make a distinction between the two. For example, Drupal[22] or TCPDF[23] emits notice at linting time.

So, when a continue may be mistaken for a break, PHP 7.3 now emits this error: `"continue" targeting switch is equivalent to "break." Did you mean to use "continue 2"?`.

More on this: Deprecate and remove continue targeting switch[24]

Monotonic Timer: hrtime()

We all have relied on `date()` and `time()` to tell us the date and time of the present moment. When measuring elapsed time, that is, the amount of time between two moments, `microtime(true)` is often used. `microtime()` returns the time of the day, with the microseconds. As soon as a difference is made between two `microtime()` calls, a bug is waiting to happen.

`microtime()` is based on the internal system clock, and the assumption is the clock will only go on, at least until 2037. Many events may impact the internal clock—daylight saving time changing (twice a year), leap seconds (27 times since 1970), and manual reconfiguration of the clock.

PHP 7.3 introduces `hrtime()`. It is a monotonic timer. It's a timer, as it behaves like `microtime()` and provides an ultra-precise representation of the time: either an array with seconds and microseconds, or a large integer.

```
print_r(hrtime(true));
print PHP_EOL;
print_r(hrtime());
```

This displays :

```
828536158380710
Array
(
    [0] => 828536
    [1] => 158403415
)
```

You may consider `hrtime()` as a modern version of `microtime()`. Any time difference should be done with `hrtime()`, while `microtime()` may be reserved for displaying the actual time. In fact, `hrtime()` `starts counting at some uncertain point in the past`. On the other hand, `hrtime()` is not affected by any variation of the internal clock.

More on this: High resolution monotonic timer #2976[25] and Monotonic Clocks—the Right Way to Determine Elapsed Time[26]

compact() Reports Undefined Variables

`compact()` is a convenient function which converts a list of variable names into an array.

```
$foo = 'bar';

$array = compact('foo', 'foz');
// ['foo' => 'bar'];
```

Until now, `compact()` would silently ignore undefined variables. It was up to the developer to check if the array was fully built, or send it to the next method without checking. Those days are gone.

`compact()` is heavily used with templating engines. This new feature may raise a large number of notices. It is recommended to check the logs in production and adapt the code. At

[19] `assert`: http://php.net/assert

[20] assert_options: http://php.net/function.assert-options

[21] Deprecations for PHP 7.3: https://wiki.php.net/rfc/deprecations_php_7_3

[22] Drupal: https://phpa.me/drupal-86-break

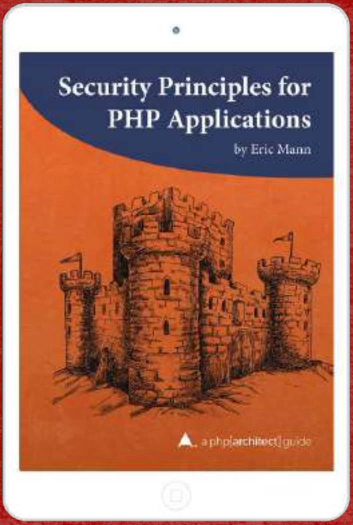
[23] TCPDF: https://phpa.me/tcpdf-break

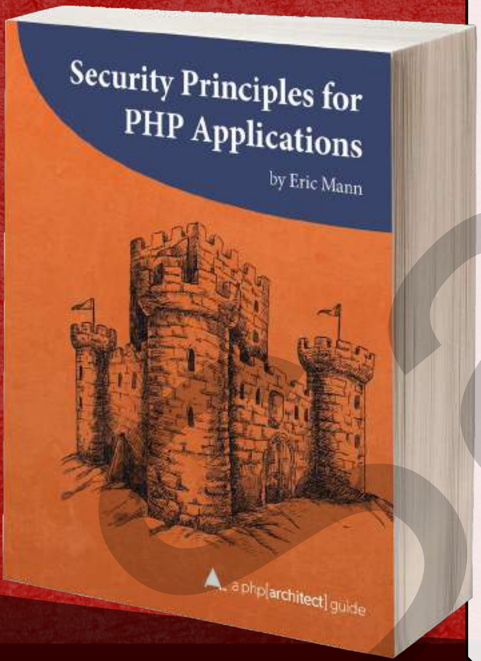
[24] Deprecate and remove continue targeting switch: https://phpa.me/php73-switch-depr

[25] High resolution monotonic timer #2976: https://github.com/php/php-src/pull/2976

[26] Monotonic Clocks—the Right Way to Determine Elapsed Time: https://phpa.me/softwariness-monotonic-clocks

a php[architect] guide





Discover how to secure your applications against many of the vulnerabilities exploited by attackers.

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

Security Principles for PHP Applications is a comprehensive guide. This book contains examples of vulnerable code side-by-side with solutions to harden it. Organized around the 2017 OWASP Top Ten list, topics cover include:

- Injection Attacks
- Authentication and Session Management
- Sensitive Data Exposure
- Access Control and Password Handling
- PHP Security Settings
- Cross-Site Scripting
- Logging and Monitoring
- API Protection
- Cross-Site Request Forgery
- ...and more.

Read a Sample Online

Written by PHP professional Eric Mann, this book builds on his experience in building secure, web applications with PHP.

Order Your Copy

http://phpa.me/security-principles

worse, you may revert to the old behavior by using the `@` operator.

More on this: Make compact function reports undefined passed variables[27]

Migration to PHP 7.3

With all those new features and incompatibilities, how do you prepare for the newest PHP version?

Figure 1 is a summary of the features, and their impact on your code.

There are three ways to get your code ready: prepare your code, review situations that may benefit from upgrades, or just wait for PHP 7.3.

Getting Ready for PHP 7.3

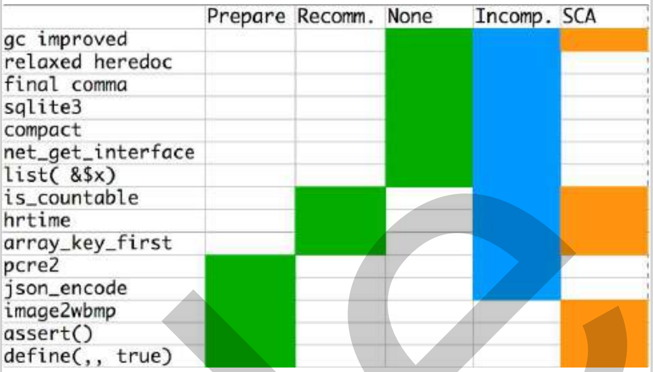
Getting ready means removing every incompatibility between the current code and the new version. For example, you can prepare for PCRE2 by collecting all your regex and linting them with PHP 7.3's regex engine. That will tell you if they are compatible or not.

On the other hand, there is not much to prepare for SQLite 3.24, since the main evolution is a new feature (UPSERT). The "recommendations" column indicates we can search now for portions of code that will benefit from the new version, such as `is_countable()`. You'll have to wait after the new version to actually start using them, or rely on a compatibility library that will emulate those functions until then. Symfony provides a polyfill for PHP 7.3[28], which at the moment only has an `is_countable` implementation but may have more to come now that the feature freeze is done. In any way, this will be a job for after the big migration.

Migrating Then Downgrading

Finally, the table in Figure 1 also reports backward incompatibilities introduced by the new features of PHP 7.3. After the code has been migrated, you'll be able to adopt new features, such as the trailing comma, or even, totally automatically, the improved Garbage Collector. All of them will make any fallback to PHP 7.2 hard or impossible, as those new features won't compile with older versions. So, once you have migrated to PHP 7.3, think twice before adopting a new feature, which may break your application's backward compatibility.

Figure 1



	Prepare	Recomm.	None	Incomp.	SCA
gc improved			🟩	🟦	🟧
relaxed heredoc			🟩	🟦	
final comma			🟩	🟦	
sqlite3			🟩	🟦	
compact			🟩	🟦	
net_get_interface			🟩	🟦	
list(&$x)			🟩	🟦	
is_countable		🟩		🟦	🟧
hrtime		🟩		🟦	🟧
array_key_first		🟩		🟦	🟧
pcre2	🟩			🟦	
json_encode	🟩			🟦	
image2wbmp	🟩				🟧
assert()	🟩				🟧
define(,, true)	🟩				🟧

Static Analysis for the Review

The final column indicates if static analysis is able to review, report, or recommend a feature. Tools like `Exakat` are already working on PHP 7.3-dev and review efficiently code for migration. It proofreads the code well beyond what the lint is capable of, and can direct your attention to the most interesting parts of the code. Migrating to a new PHP version is a good moment to add them to your tool belt.

While We're Waiting for December

PHP 7.3 is forecast for December 13th. Until then, it is important to keep checking that your code meets this new version. There are several steps you can take which will help you and the PHP community:

- download PHP 7.3 from GitHub[29] and build it
- lint your code with `php -l`, just to check the syntax
- run a static analysis tool, such as Exakat[30], to review all issues
- run your test suite
- report bugs to PHP Bugs[31]

Damien Seguy is CTO at Exakat Ltd., a company specialized in PHP code quality solutions for the industry. He leads development of the exakat PHP static analysis engine that automatically audit code for version compatibility, security and auto-documentation. Since last millenium, Damien has contributed to PHP, as documentation author, elephpant breeder, conference speaker on every continents. He also enjoys machine learning, gremlin, 狮子头 and camembert. @exakat

Related Reading

- *finally{}: Innovation in PHP* by Eli White. May 2018.
  https://phparch.com/magazine/2018-2/may/
- *Community Corner: The Imminent Release of PHP 7.2* by James Titcumb. October 2017.
  https://phparch.com/magazine/2017-2/october/

[27] *Make compact function reports undefined passed variables:* *https://wiki.php.net/rfc/compact*

[28] *polyfill for PHP 7.3:* *https://phpa.me/symfony-php73-polyfill*

[29] *GitHub: https://github.com/php/php-src/*

[30] *Exakat: https://www.exakat.io*

[31] *PHP Bugs: https://bugs.php.net*

php[architect]
CONTENTS
Mutant Testing
MARCH
24
Education Station: Hunting Mutants with Humbug
29
Leveling Up: Ensuring Your Tests are Valuable
Columns
22 Community Corner: Modern PHP
12 An Introduction to Compilers, Interpreters, and JITs
17 Getting Started with Zend Expressive
33 Security Corner: Securing Legacy Applications—Part 1
35 February Happenings
36 finally(): Growing Programmer Population
Object-Relational Mapping with Laravel's Eloquent
Integrating Heterogeneous Web Applications with Laravel and Guzzle
Mutant Testing
Education Station: Hunting Mutants with Humbug
Leveling Up: Ensuring Your Tests are Valuable
ALSO INSIDE
Object-Relational Mapping with Laravel's Eloquent
Integrating Heterogeneous Web Applications with Laravel and Guzzle
An Introduction to Compilers, Interpreters, and JITs
Getting Started with Zend Expressive
Community Corner: Modern PHP
finally(): Growing Programmer
Security Corner: Securing Legacy
St Louis

Borrowed this magazine?

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

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

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

php[**architect**]

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

http://phpa.me/mag_subscribe