PHP 7 Upgrade Guide

By Colin O'Dell

Your guide to new features, breaking changes, and more.

PHP 7 Upgrade Guide

Your guide to new features, breaking changes, and more.

Colin O'Dell

This book is for sale at http://leanpub.com/php7

This version was published on 2017-11-17

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Colin O'Dell by spreading the word about this book on Twitter!

The suggested tweet for this book is:

I just bought the PHP 7 Upgrade Guide! #php7book

The suggested hashtag for this book is #php7book.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

#php7book

This book is dedicated to all those who hunger to learn.

Contents

Sample

Thanks for downloading this sample of *What's New in PHP 7*[1]!

[1] https://leanpub.com/php7

Preface

Who is this book for

This is book is for PHP developers looking to jump into PHP 7. You'll need previous experience with PHP in order to understand the topics and examples we'll cover. The more you know about PHP 5 and OOP, the more you'll understand why some of the changes are important and how they'll impact your development.

If you're a manager or leader of a team looking to work in PHP 7, this book will help your developers quickly catch up on all the changes they need to know about, and even expose some new features they can take advantage of for faster, better development.

Contents

Chapter 1 summarizes the addition of scalar type hints. This allows your function parameters to explicitly require scalar types like `string`, `int`, `float`, or `bool`.

Chapter 2 covers the declaration of function return types, including the scalar types mentioned in chapter 1.

In Chapter 3 we explore the brand new "Spaceship Operator" which drastically simplifies 3-way comparison of two expressions.

Chapter 4 introduces another new operator, the "Null Coalesce Operators". It's a lot like `?:` but with a built-in `isset()` check.

Chapter 5 demonstrates how to easily add Unicode characters to strings using a new `\u{...}` escape sequence.

Chapter 6 reviews how anonymous classes can be used to create classes on-the-fly, which is particularly useful for mocking and implementing simple interfaces (like loggers and observers).

Chapter 7 highlights the new ability to include multiple classes from a namespace with a single `use` statement.

Chapter 8 introduces the new `Closure::call` method which drastically simplifies binding your closures to objects at call-time.

Chapter 9 shows how generator can now `return` final values.

Chapter 10 explores how generators can `yield from` arrays, iterators, and other generators (insert Xzibit "yo dawg" joke here).

In Chapter 11 we see how PHP 7 drastically simplifies the process of generating strong random numbers for security-critical applications.

Chapter 12 demonstrates how to safely and easily perform integer division.

Chapter 13 unveils the new `preg_replace_callback_array` function for executing different callbacks per regular expression.

Chapter 14 covers the new `IntlChar` class, which helps you work with Unicode characters.

Chapter 15 shows how PHP 7 is dramatically faster than previous versions.

Chapter 16 describes how syntax is more consistent and flexible.

Chapter 17 explains how previously-reserved keywords can be used for property, constant, and method names. It contains a full list of such words.

Chapter 18 walks through the new changes to engine errors and exceptions.

Other resources

These official PHP resources were a huge help during the creation of this book, and you may find them useful:

- PHP Manual: Migrating from PHP 5.6.x to PHP 7.0.x[2]
- PHP 7 UPGRADING doc[3]
- PHP 7 Requests for Comments[4]
- PHPNG (next generation)[5]

There are plenty of other great resources out there too, including some paid and some free ones:

- Getting Ready for PHP 7[6]
- What to Expect When You're Expecting: PHP 7, Part 1[7]
- What to Expect When You're Expecting: PHP 7, Part 2[8]
- Zend: 5 Things You Must Know About PHP 7[9]
- The PHP 7 Revolution: Return Types and Removed Artifacts[10]
- PHP 7: 10 Things You Need to Know[11]
- #php7 on Twitter[12]
- GoPHP7 Extensions Project[13]
- Laracasts - PHP 7 Up and Running[14]

(Any commercial products or services listed here have not been tested or endorsed by the author - they are simply provided as a jumping-off point for your continued PHP 7 education.)

[2] https://secure.php.net/manual/en/migration70.php
[3] https://github.com/php/php-src/blob/PHP-7.0.0/UPGRADING
[4] https://wiki.php.net/rfc#php_70
[5] https://wiki.php.net/phpng
[6] https://www.digitalocean.com/company/blog/getting-ready-for-php-7/
[7] https://blog.engineyard.com/2015/what-to-expect-php-7
[8] https://blog.engineyard.com/2015/what-to-expect-php-7-2
[9] http://www.zend.com/en/resources/php-7
[10] http://www.sitepoint.com/php-7-revolution-return-types-removed-artifacts/
[11] http://www.hongkiat.com/blog/php7/
[12] https://twitter.com/hashtag/php7
[13] http://gophp7.org/gophp7-ext/
[14] https://laracasts.com/series/php7-up-and-running

Acknowledgements

I'd like to thank the following people for making this book possible:

- All the programmers, testers, documentation writers, RFC authors, and everyone else who has contributed to the success of PHP. I'd have nothing to write about if it wasn't for you.
- The PHP community for sharing knowledge and expertise, thereby creating this amazing ecosystem for all developers.
- Mike Spinosa, Scott Greenwell, Ben Thomas, and the whole team at Unleashed Technologies for encouraging and promoting my continuous growth.
- Phil Sturgeon for being the catalyst behind my increased community involvement.

PHP Wiki and Documentation Content

[15] https://wiki.php.net/
[16] https://wiki.php.net/rfc
[17] https://php.net/docs.php
[18] https://creativecommons.org/licenses/by/3.0/us/

Getting Started with PHP 7

Builds for PHP 7.0 and 7.1 are now available from both official and community repositories. In most cases, these new versions can be easily installed using your system's package manager.

PHP 5 Conflicts

You may encounter conflicts if you already have PHP 5 installed. If so, make sure to completely remove this older version from your system before installing PHP 7.

Alternatively, you could use something like phpbrew to safely install multiple versions side-by-side.

Ubuntu 14.04 and 16.04+

Ondřej Surý[19] provides a PPA for installing PHP 7.0 and 7.1[20]. The latest release can be installed using these commands:

```
sudo add-apt-repository ppa:ondrej/php
sudo apt-get update
sudo apt-get install php7.0
```

(Replace `7.0` with `7.1` if desired)

The full list of available packages can be found here: https://launchpad.net/~ondrej/+archive/ubuntu/php/+packages?field.name_filter=php7.0

[19] https://launchpad.net/~ondrej

[20] https://launchpad.net/~ondrej/+archive/ubuntu/php

Debian 8 & 9

Ondřej Surý also provides PHP 7.0 and 7.1 packages for Debian[21] which can be installed using these commands:

```
sudo apt-get install apt-transport-https lsb-release ca-certificates
sudo wget -O /etc/apt/trusted.gpg.d/php.gpg https://packages.sury.org/php/apt.gpg
sudo sh -c "echo 'deb https://packages.sury.org/php/ $(lsb_release -sc) main'" >\
 /etc/apt/sources.list.d/php.list
sudo apt-get update
sudo apt-get install php7.0
```

[21] https://packages.sury.org/php/

Debian 6 & 7

PHP 7.0 can be installed using the Dotdeb repository[22]:

Add these two lines to your `/etc/apt/sources.list` file, replacing `<distribution>` with either `squeeze`, `wheezy`, or `jessie`:

```
deb http://packages.dotdeb.org <distribution> all
deb-src http://packages.dotdeb.org <distribution> all
```

Add the GPG key:

```
wget https://www.dotdeb.org/dotdeb.gpg
sudo apt-key add dotdeb.gpg
```

Install PHP 7:

```
sudo apt-get update
sudo apt-get install php7.0
```

The full list of available packages can be found here: http://packages.dotdeb.org/pool/all/p/php7.0/

[22] https://www.dotdeb.org/

CentOS / RHEL

PHP 7 can be installed using the Webstatic Yum repository[23]:

If you're using CentOS/RHEL 7.x, run these three commands to add the repository and install PHP 7:

```
rpm -Uvh https://dl.fedoraproject.org/pub/epel/epel-release-latest-7.noarch.rpm
rpm -Uvh https://mirror.webtatic.com/yum/el7/webtatic-release.rpm
yum install php70w
```

If you're using CentOS/RHEL 6.x, run these two commands to add the repository and install PHP 7:

```
rpm -Uvh https://mirror.webtatic.com/yum/el6/latest.rpm
yum install php70w
```

The full list of available packages (including SAPIs and extensions) can be found here: https://webtatic.com/packages/php70/#sapi

[23] https://webtatic.com/projects/yum-repository/

Mac OS X

PHP 7 can be installed using homebrew[24]:

```
brew tap homebrew/dupes
brew tap homebrew/versions
brew tap homebrew/homebrew-php
brew install php70
```

Or you can install it via Liip's php-osx tool[25]:

```
curl -s http://php-osx.liip.ch/install.sh | bash -s 7.0
```

[24] https://github.com/Homebrew/homebrew-php
[25] http://php-osx.liip.ch/

Windows

PHP 7 distributions for Windows can be found on the windows.php.net website: http://windows.php.net/download#php-7.0

For local development, you could instead use third-party distributions like XAMPP[26] or EasyPHP[27]. Both come with PHP 7, MySQL, and a web server so you can get up-and-running fast.

[26] https://www.apachefriends.org/index.html
[27] http://www.easyphp.org/

phpbrew

phpbrew[28] is a wonderful utility which allows you to easily build and switch between different versions of PHP on the same machine. It supports all modern versions of PHP, including PHP 7.

First we need phpbrew to fetch information about available versions:

```
phpbrew self-update
```

We can then install PHP 7.0, 7.1, or any other version:

```
phpbrew install php-7.0.25
phpbrew use php-7.0.25
```

[28] https://github.com/phpbrew/phpbrew

Docker

Official Docker images for PHP can be found at https://hub.docker.com/_/php/.

```
docker pull php:latest
```

Vagrant Image

Rasmus Lerdorf, the creator of PHP, provides a Vagrant box image on his GitHub: https://github.com/rlerdorf/php7dev

It's based on Debian 8 and is pre-configured to help develop PHP apps and extensions.

Build from Source

PHP.net also offers source packages for the alpha and beta releases which you can download and compile yourself. The source code can be downloaded from https://downloads.php.net/~ab/

Documentation on compiling PHP 7:

- https://wiki.php.net/phpng
- http://www.zimuel.it/install-php-7/
- http://www.hashbangcode.com/blog/compiling-and-installing-php7-ubuntu

Part 1 - New Features

Perhaps the most exciting part of PHP 7 are the new features! This part of the book covers these features in detail, including examples of useful applications.

Chapter 1: Scalar Type Hints

PHP 5 introduced the ability to require function parameters to be of a certain type. This provides a safeguard against invalid uses, like passing a UNIX timestamp to a method which expects a `DateTime` object. It also makes it clear to other developers how the function should be used. For example, compare the following method signatures with and without type hints:

```
// No type hint:
function getNextWeekday($date) { /*...*/ }

// Class type hint (PHP 5):
function getNextWeekday(DateTime $date) { /*...*/ }
```

These hints were initially limited to just classes and interfaces, but was soon expanded to allow `array` and `callable` types too. PHP 7 extends this further by allowing scalar types like `int`, `float`, `string` and `bool`:

```
// Scalar type hint (PHP 7):
function getNextWeekday(int $date) { /*...*/ }
```

Type Checking Modes

PHP's flexible type system is one of its most-useful features, allowing numeric strings to be used as integers and vice-versa. This tradition is continued in PHP 7 but now you have the option of enabling strict type enforcement like you'd see in other languages (such as Java or C#). This setting can be enabled using the `declare` construct[29].

Weak ("Coercive") Type Checking

"Coercive" mode is the default type checking mode in PHP 7. This is identical to how previous versions of PHP handled scalar type hints for built-in functions. For example, take a look at the method signature for the `floor` function:

[29] https://php.net/manual/en/control-structures.declare.php

```
float floor ( float $value )
```

When you pass in numeric strings or integers PHP auto-converts them to a float automatically. This behavior has simply been extended to userland functions as well.

Here's a table showing which scalar types are accepted in "Coercive" mode based on the declared type:

Allowed types, coercive mode

Type declaration	int	float	string	bool	object
int	yes	yes*	yes†	yes	no
float	yes	yes	yes†	yes	no
string	yes	yes	yes	yes	yes‡
bool	yes	yes	yes	yes	no

* Only non-`NaN` floats between `PHP_INT_MIN` and `PHP_INT_MAX` accepted.

† If it's a numeric string

‡ Only if object has a `__toString()` method

Strong ("Strict") Type Checking

PHP 7 introduces a new "strict" mode which is enabled by placing `declare(strict_types=1);` at the top of your script like so:

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

function welcome(string $name) {
    echo 'Hello ' . $name;
}

welcome('World'); // Prints: Hello World
```

Be careful where you place `declare`

`declare(strict_types=1);` must be on the first line or a compiler error will be raised.

This declaration enables strict mode for all uses in the entire file, including built-in PHP function calls and return values (see next chapter). It does not affect any included files nor any other files which include it.

Strict mode essentially requires you to pass in exact types declared. If you don't, a `TypeError` will be thrown. For example, in strict mode, you cannot use numeric strings when a `float` or `int` is expected (like you can in weak/coercive mode):

This throws an error in strict mode

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

function welcome(string $name) {
    echo 'Hello ' . $name;
}

welcome(3);

// Fatal error: Uncaught TypeError: Argument 1 passed to welcome() must be of th\
e type string, integer given
```

TypeError

A `TypeError` is a new type of "exception" that can be thrown. Technically it extends from the new `Throwable` interface, which `Exception` also extends from. You can learn more about this in Chapter 17 - Error Handling.

There's one exception to this rule though, which is that `float` declarations can accept `int` values:

Scope widening example

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

function add(float $a, float $b): float {
    return $a + $b;
}

add(1, 2); // float(3)
```

Allowed types, strict mode

Type declaration	int	float	string	bool	object
int	yes	no	no	no	no
float	yes*	yes	no	no	no
string	no	no	yes	no	no
bool	no	no	no	yes	no

* Allowed due to widening primitive conversion

Mixing Modes

Because the directive is set per-file, it's entirely possible to mix modes in your application. For example, your strictly-checked app could use a weakly-checked library (or vice versa) without any issues or complications.

Backwards Compatibility

You may no longer create classes named `int`, `float`, `string` or `bool` as these would conflict with the new scalar type hints.

Further Reading

- RFC: Scalar Type Declarations[30]
- PHP Documentation: Type Hinting[31]

[30] https://wiki.php.net/rfc/scalar_type_hints_v5
[31] https://php.net/manual/en/language.oop5.typehinting.php

Chapter 3: Combined Comparison (Spaceship) Operator

PHP 7 introduces a new three-way comparison operator `<=>` (`T_SPACESHIP`) which takes two expressions: `(expr) <=> (expr)`. It compares both sides of the expression and, depending on the result, returns one of three values:

0	If both expressions are equal
1	If the left is greater
-1	If the right is greater

You may be familiar with this if you've worked with existing comparison functions like `strcmp` before.

It has the same precedence as the equality operator (==, ===, !=, !==) and behaves identically to the existing comparison operators[32] (<, <=, ==, >=, >).

Comparing Values

You can compare everything from scalar values (like `ints` and `floats`) to arrays and even objects too. Here are some examples from the relevant RFC[33]:

PHP RFC examples (licensed under CC BY 3.0)

```
// Integers
echo 1 <=> 1; // 0
echo 1 <=> 2; // -1
echo 2 <=> 1; // 1

// Floats
echo 1.5 <=> 1.5; // 0
echo 1.5 <=> 2.5; // -1
echo 2.5 <=> 1.5; // 1

// Strings
echo "a" <=> "a"; // 0
```

[32] http://php.net/manual/en/language.operators.comparison.php
[33] https://wiki.php.net/rfc/combined-comparison-operator

```php
echo "a" <=> "b"; // -1
echo "b" <=> "a"; // 1

echo "a" <=> "aa"; // -1
echo "zz" <=> "aa"; // 1

// Arrays
echo [] <=> []; // 0
echo [1, 2, 3] <=> [1, 2, 3]; // 0
echo [1, 2, 3] <=> []; // 1
echo [1, 2, 3] <=> [1, 2, 1]; // 1
echo [1, 2, 3] <=> [1, 2, 4]; // -1

// Objects
$a = (object) ["a" => "b"];
$b = (object) ["a" => "b"];
echo $a <=> $b; // 0

$a = (object) ["a" => "b"];
$b = (object) ["a" => "c"];
echo $a <=> $b; // -1

$a = (object) ["a" => "c"];
$b = (object) ["a" => "b"];
echo $a <=> $b; // 1

// only values are compared
$a = (object) ["a" => "b"];
$b = (object) ["b" => "b"];
echo $a <=> $b; // 0
```

Sorting

Perhaps the best application of this operator is to simplify sorting, as functions like `usort` expect you to perform a comparison and return `-1`, `0`, or `1` accordingly:

This simplification is especially apparent when comparing objects by some property value:

```php
class Spaceship {
    public $name;
    public $maxSpeed;

    public function __construct($name, $maxSpeed) {
        $this->name = $name;
        $this->maxSpeed = $maxSpeed;
    }
}

$spaceships = [
    new Spaceship('Rebel Transport', 20),
    new Spaceship('Millenium Falcon', 80),
    new Spaceship('X-Wing Starfighter', 80),
    new Spaceship('TIE Bomber', 60),
    new Spaceship('TIE Fighter', 100),
    new Spaceship('Imperial Star Destroyer', 60),
];

// Sort the spaceships by name (in ascending order)
usort($spaceships, function ($ship1, $ship2) {
    return $ship1->name <=> $ship2->name;
});

echo $spaceships[0]->name; // "Imperial Star Destroyer"

// Sort the spaceships by speed (in descending order)
// Notice how we switch the position of $ship1 and $ship2
usort($spaceships, function ($ship1, $ship2) {
    return $ship2->maxSpeed <=> $ship1->maxSpeed;
});

echo $spaceships[0]->name; // "TIE Fighter"
```

Without the comparison operator, these functions would be much more complex:

```
usort($spaceships, function ($ship1, $ship2) {
    if ($ship1->maxSpeed == $ship2->maxSpeed) {
        return 0;
    } elseif ($ship1->maxSpeed < $ship2->maxSpeed) {
        return 1;
    } else {
        return -1;
    }
});
```

Sorting by multiple values

You can take advantage of the array comparison behavior to easily sort by multiple fields:

```
// Sort by speed (asc), then by name (asc)
usort($spaceships, function ($ship1, $ship2) {
    return [$ship1->maxSpeed, $ship1->name] <=> [$ship2->maxSpeed, $ship2->name];
});

foreach ($spaceships as $ship) {
    printf("%3d is the max speed of a(n) %s\n", $ship->maxSpeed, $ship->name);
}

// Outputs:
//  20 is the max speed of a(n) Rebel Transport
//  60 is the max speed of a(n) Imperial Star Destroyer
//  60 is the max speed of a(n) TIE Bomber
//  80 is the max speed of a(n) Millenium Falcon
//  80 is the max speed of a(n) X-Wing Starfighter
// 100 is the max speed of a(n) TIE Fighter
```

Further Reading

- RFC: Combined Comparison (Spaceship) Operator[34]
- PHP Manual: Comparison Operators[35]

[34] https://wiki.php.net/rfc/combined-comparison-operator
[35] https://secure.php.net/manual/en/language.operators.comparison.php

Chapter 5: Unicode Codepoint Escape Syntax

PHP's lack of native Unicode support can make things difficult when coding for the web. While libraries like `iconv` and `mbstring` have simplified working with strings, there were no simple mechanisms available to create Unicode characters or strings without converting them from an HTML or JSON representation:

Workaround examples

```
$char = html_entity_decode('&#x2603', 0, 'UTF-8');
$char = mb_convert_encoding('&#x2603', 'UTF-8', 'HTML-ENTITIES');
$char = json_decode('"\\u2603"');
```

PHP 7 finally introduces native support for Unicode character escape sequences within strings, just like you'd see in Ruby or ECMAScript 6:

```
$char = "\u{2603}";
```

This makes it much easier (and quicker) to embed Unicode characters, especially ones that aren't easily typed. These can be used in strings alongside other characters too. For example, here's the `U+202E RIGHT-TO-LEFT OVERRIDE` character being used to display the string in reverse:

```
echo "\u{202E}This is backwards"; // displays: sdrawkcab si sihT
```

String encoding

This construct produces the corresponding codepoint as bytes. Don't forget that PHP strings don't have specific encodings by default.

You can omit leading `0`s if you'd like:

```
echo "\u{58}"; // "X"
echo "\u{0058}"; // "X"
```

Why the {}s?

Some other languages (C/C++/Java) use a format without the {} characters: \uXXXX. Unfortunately this limits their use to the Basic Multilingual Plane (U+0000 to U+FFFF). However, Unicode supports other characters beyond 16 bits.

For example, if we wanted to represent the U+1F427 PENGUIN emoji, our escape sequence would look something like this: \u1F427. Most languages would intepret this as U+1F42 GREEK SMALL LETTER OMICRON WITH PSILI AND VARIA plus a 7, which is not what we want. In these languages, you'd have to encode it using two 16-bit sequences like this: \uD83D\uDC27. This isn't very clear though.

Wrapping with {} characters allows us to easily go beyond that 16-bit limitation without sacrficing clarity: \u{1F427}

Limitations

This feature follows the behavior of all other escape sequences in PHP - they can only be used within double-quoted strings and heredocs:

Example usage

```
$foo = "\u{2109}\u{2134}\u{2134}";
// □□□

$bar = <<<EOT
    \u{212C}\u{212B}\u{211D}
EOT;
// □Å□
```

And like other sequences such as \t, they will *not* be expanded when they occur in single-quoted strings or nowdocs:

These will not work

```
$foo = '\u{2109}\u{2134}\u{2134}';
// \u{2109}\u{2134}\u{2134}

$bar = <<<'EOT'
    \u{212C}\u{212B}\u{211D}
EOT;
// \u{212C}\u{212B}\u{211D}
```

Backwards Compatibility

Double-quoted strings and heredocs containing `\u{` followed by an invalid sequence will now result in an error. This can be avoided by escaping the leading backslash with another backslash (`\\u{`).

Further Reading:

- RFC: Unicode Codepoint Escape Syntax[36]

[36] https://wiki.php.net/rfc/unicode_escape

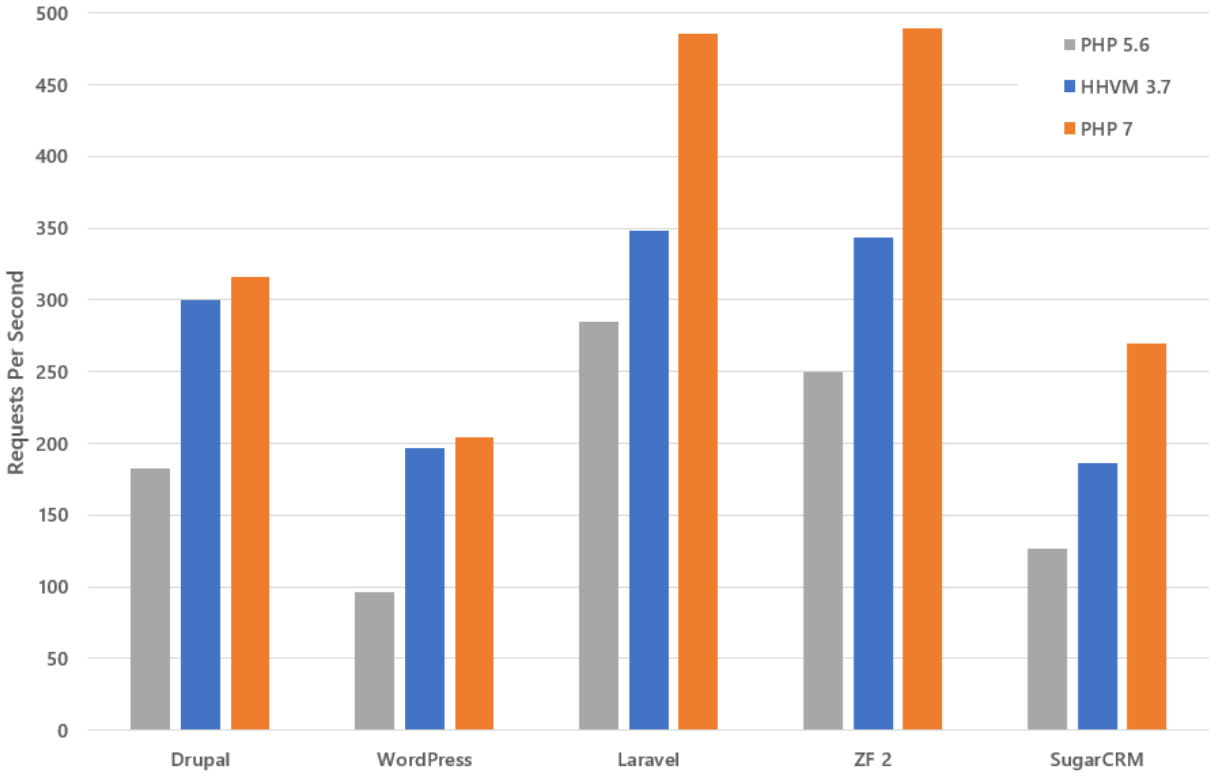
Part 2 - Language Changes & Improvements

PHP 7 includes several major improvements to the language, as well as some changes to existing features.

Learn about the features and changes which make PHP 7 faster and more flexible than ever!

Chapter 15: Performance

PHP 7 is faster than ever. Most real-world applications will see a 100%+ boost in performance simply by upgrading! Here are some actual benchmarks from the Zend Performance Team[37]:



Zend Performance Team Benchmarks

This improvement is largely due to the PHPNG (PHP Next-Gen) project which refactored the Zend Engine to be super-performant. Some of the under-the-hood changes include things like:

- Using more-compact data structures
- Reducing the number of heap allocations and deallocations
- Utilizing native thread local storage

The net result is a faster PHP which also uses less memory. These changes also pave the way for future improvements, like JIT compilation, which will make PHP even faster.

[37] https://www.zend.com/en/resources/php7_infographic

Further Reading

- Zend: Turbocharging the Web with PHP 7[38]
- Rasmus Lerdorf - PHP Australia Keynote[39]

[38] https://www.zend.com/en/resources/php7_infographic
[39] http://talks.php.net/oz15#/

Part 3 - Deprecations & Removals

PHP 7 deprecates a couple features and removes several others which worked in previous versions of PHP. You'll want to adjust your code accordingly before upgrading to PHP 7.

Appendix - Backward Compatibility Breaks

If you're coming from a modern version of PHP (like 5.5 or 5.6), most of your code will probably work fine as-is. Nevertheless, we'll cover most of the important breaking changes in this section so you'll be prepared to fix any such issues in your projects.

The majority of these BC breaks were covered in previous chapters. We'll link back to those sections as needed.

This list comes directly from the php-src `UPGRADING` document as of 7 November 2015. It may not be exhaustive. Please check https://github.com/php/php-src/blob/PHP-7.0.0/UPGRADING for the latest version.

Language Changes

Variable handling

Left-to-right parsing

Indirect variable, property and method references are now interpreted with left-to-right semantics. Some examples:

```
$$foo['bar']['baz'] // interpreted as ($$foo)['bar']['baz']
$foo->$bar['baz']   // interpreted as ($foo->$bar)['baz']
$foo->$bar['baz']() // interpreted as ($foo->$bar)['baz']()
Foo::$bar['baz']()  // interpreted as (Foo::$bar)['baz']()
```

To restore the previous behavior add explicit curly braces:

```
${$foo['bar']['baz']}
$foo->{$bar['baz']}
$foo->{$bar['baz']}()
Foo::{$bar['baz']}()
```

Refer back to Chapter 16 - Uniform Variable Syntax.

Global Keywords

The global keyword now only accepts simple variables. Instead of

```
global $$foo->bar;
```

it is now required to write the following:

```
global ${$foo->bar};
```

Refer back to Chapter 16 - Uniform Variable Syntax.

Parenthesis influencing behavior

Parentheses around variables or function calls no longer have any influence on behavior. For example the following code, where the result of a function call is passed to a by-reference function

```
function getArray() { return [1, 2, 3]; }

$last = array_pop(getArray());
// Strict Standards: Only variables should be passed by reference
$last = array_pop((getArray()));
// Strict Standards: Only variables should be passed by reference
```

will now throw a strict standards error regardless of whether parentheses are used. Previously no notice was generated in the second case.

By-reference assignment ordering

Array elements or object properties that are automatically created during by-reference assignments will now result in a different order. For example

```
$array = [];
$array["a"] =& $array["b"];
$array["b"] = 1;
var_dump($array);
```

now results in the array [“a” ⇒ 1, “b” ⇒ 1], while previously the result was [“b” ⇒ 1, “a” ⇒ 1];

Refer back to Chapter 16 - Abstract Syntax Tree.

Relevant RFCs

- https://wiki.php.net/rfc/uniform_variable_syntax
- https://wiki.php.net/rfc/abstract_syntax_tree

`list()` behavior

Variable assignment order

list() will no longer assign variables in reverse order. For example

```
list($array[], $array[], $array[]) = [1, 2, 3];
var_dump($array);
```

will now result in $array == [1, 2, 3] rather than [3, 2, 1]. Note that only the **order** of the assignments changed, but the assigned values stay the same. E.g. a normal usage like

```
list($a, $b, $c) = [1, 2, 3];
// $a = 1; $b = 2; $c = 3;
```

will retain its current behavior.

Empty list assignments

Empty list() assignments are no longer allowed. As such all of the following are invalid:

```
list() = $a;
list(,,) = $a;
list($x, list(), $y) = $a;
```

list() no longer supports unpacking strings (while previously this was only supported in some cases). The code

```
$string = "xy";
list($x, $y) = $string;
```

will now result in $x == null and $y == null (without notices) instead of $x == “x” and $y == “y”. Furthermore list() is now always guaranteed to work with objects implementing ArrayAccess, e.g.

```
list($a, $b) = (object) new ArrayObject([0, 1]);
```

will now result in $a == 0 and $b == 1. Previously both $a and $b were null.

Refer back to Chapter 28 - Behavior Changes to `list`.

Relevant RFCs

- https://wiki.php.net/rfc/abstract_syntax_tree#changes_to_list
- https://wiki.php.net/rfc/fix_list_behavior_inconsistency

`foreach` behavior

Interaction with internal array pointers

Iteration with foreach() no longer has any effect on the internal array pointer, which can be accessed through the current()/next()/etc family of functions. For example

```
$array = [0, 1, 2];
foreach ($array as &$val) {
    var_dump(current($array));
}
```

will now print the value int(0) three times. Previously the output was int(1), int(2) and bool(false).

Array iteration by-value

When iterating arrays by-value, foreach will now always operate on a copy of the array, as such changes to the array during iteration will not influence iteration behavior. For example

```
$array = [0, 1, 2];
$ref =& $array; // Necessary to trigger the old behavior
foreach ($array as $val) {
    var_dump($val);
    unset($array[1]);
}
```

will now print all three elements (0 1 2), while previously the second element 1 was skipped (0 2).

Array iteration by-reference

When iterating arrays by-reference, modifications to the array will continue to influence the iteration. However PHP will now do a better job of maintaining a correct position in a number of cases. E.g. appending to an array during by-reference iteration

```
$array = [0];
foreach ($array as &$val) {
    var_dump($val);
    $array[1] = 1;
}
```

will now iterate over the appended element as well. As such the output of this example will now be "int(0) int(1)", while previously it was only "int(0)".

Object iteration

Iteration of plain (non-Traversable) objects by-value or by-reference will behave like by-reference iteration of arrays. This matches the previous behavior apart from the more accurate position management mentioned in the previous point.

Iteration of Traversable objects remains unchanged.

Refer back to Chapter 27 - Behavior Changes to `foreach`.

Relevant RFC

- https://wiki.php.net/rfc/php7_foreach

Parameter handling

Duplicate parameter names

It is no longer possible to define two function parameters with the same name. For example, the following method will trigger a compile-time error:

```
public function foo($a, $b, $unused, $unused) {
    // ...
}
```

Code like this should be changed to use distinct parameter names, for example:

```
public function foo($a, $b, $unused1, $unused2) {
    // ...
}
```

Retrieving argument values

The func_get_arg() and func_get_args() functions will no longer return the original value that was passed to a parameter and will instead provide the current value (which might have been modified). For example

```
function foo($x) {
    $x++;
    var_dump(func_get_arg(0));
}
foo(1);
```

will now print "2" instead of "1". This code should be changed to either perform modifications only after calling func_get_arg(s)

```
function foo($x) {
    var_dump(func_get_arg(0));
    $x++;
}
```

or avoid modifying the parameters altogether:

```
function foo($x) {
    $newX = $x + 1;
    var_dump(func_get_arg(0));
}
```

Effect on backtraces

Similarly exception backtraces will no longer display the original value that was passed to a function and show the modified value instead. For example

```
function foo($x) {
    $x = 42;
    throw new Exception;
}
foo("string");
```

will now result in the stack trace

```
Stack trace:
#0 file.php(4): foo(42)
#1 {main}
```

while previously it was:

```
Stack trace:
#0 file.php(4): foo('string')
#1 {main}
```

While this should not impact runtime behavior of your code, it is worthwhile to be aware of this difference for debugging purposes.

The same limitation also applies to debug_backtrace() and other functions inspecting function arguments.

Refer back to Chapter 29 - Parameter Handling Changes.

Relevant RFC

- https://wiki.php.net/phpng

Integer handling

Invalid octal literals

Invalid octal literals (containing digits larger than 7) now produce compile errors. For example, the following is no longer valid:

```
$i = 0781; // 8 is not a valid octal digit!
```

Previously the invalid digits (and any following valid digits) were simply ignored. As such $i previously held the value 7, because the last two digits were silently discarded.

Refer back to Chapter 31 - Errors on Invalid Octal Literals.

Negative bitwise shifting

Bitwise shifts by negative numbers will now throw an ArithmeticError:

```
var_dump(1 >> -1);
// ArithmeticError: Bit shift by negative number
```

Left bitwise shifts

Left bitwise shifts by a number of bits beyond the bit width of an integer will always result in 0:

```
var_dump(1 << 64); // int(0)
```

Previously the behavior of this code was dependent on the used CPU architecture. For example on x86 (including x86-64) the result was int(1), because the shift operand was wrapped.

Right bitwise shifts

Similarly right bitwise shifts by a number of bits beyond the bit width of an integer will always result in 0 or -1 (depending on sign):

```
var_dump(1 >> 64);  // int(0)
var_dump(-1 >> 64); // int(-1)
```

Refer back to Chapter 24 - Integer Semantics.

Relevant RFC

- https://wiki.php.net/rfc/integer_semantics

String handling

Hexadecimal numeric strings

Strings that contain hexadecimal numbers are no longer considered to be numeric and don't receive special treatment anymore. Some examples of the new behavior:

```
var_dump("0x123" == "291");     // bool(false)     (previously true)
var_dump(is_numeric("0x123")); // bool(false)     (previously true)
var_dump("0xe" + "0x1");        // int(0)          (previously 16)

var_dump(substr("foo", "0x1")); // string(3) "foo" (previously "oo")
// Notice: A non well formed numeric value encountered
```

filter_var() can be used to check if a string contains a hexadecimal number or convert such a string into an integer:

```
$str = "0xffff";
$int = filter_var($str, FILTER_VALIDATE_INT, FILTER_FLAG_ALLOW_HEX);
if (false === $int) {
    throw new Exception("Invalid integer!");
}
var_dump($int); // int(65535)
```

Refer back to Chapter 31 - Errors on Invalid Octal Literals.

Unicode escape sequence

Due to the addition of the Unicode Codepoint Escape Syntax for double-quoted strings and heredocs, "\u{" followed by an invalid sequence will now result in an error:

```
$str = "\u{xyz}"; // Fatal error: Invalid UTF-8 codepoint escape sequence
```

To avoid this the leading backslash should be escaped:

```
$str = "\\u{xyz}"; // Works fine
```

However, "\u" without a following { is unaffected. As such the following code won't error and will work the same as before:

```
$str = "\u202e"; // Works fine
```

Refer back to Chapter 5 - Unicode Codepoint Escape Syntax.

Relevant RFCs

- https://wiki.php.net/rfc/remove_hex_support_in_numeric_strings
- https://wiki.php.net/rfc/unicode_escape

Error handling

Errors as throwables

There are now two exception classes: Exception and Error. Both classes implement a new interface Throwable. Type hints in exception handling code may need to be changed to account for this.

Fatal errors

Some fatal errors and recoverable fatal errors now throw an Error instead. As Error is a separate class from Exception, these exceptions will not be caught by existing try/catch blocks.

For the recoverable fatal errors which have been converted into an exception, it is no longer possible to silently ignore the error from an error handler. In particular, it is no longer possible to ignore type hint failures.

Parser errors

Parser errors now generate a ParseError that extends Error. Error handling for eval()s on potentially invalid code should be changed to catch ParseError in addition to the previous return value / error_get_last() based handling.

Internal class constructor failures

Constructors of internal classes will now always throw an exception on failure. Previously some constructors returned NULL or an unusable object.

Refer back to Chapter 18 - Error Handling and Exceptions.

Reclassification of E_STRICT

The error level of some E_STRICT notices has been changed.

Refer back to Chapter 36 - Reclassification and Removal of E_STRICT Notices.

Relevant RFCs

- https://wiki.php.net/rfc/engine_exceptions_for_php7
- https://wiki.php.net/rfc/throwable-interface
- https://wiki.php.net/rfc/internal_constructor_behaviour
- https://wiki.php.net/rfc/reclassify_e_strict

Other language changes

Static calls to non-static methods

Removed support for static calls to non-static methods from an incompatible `$this` context. In this case $this will not be defined, but the call will be allowed with a deprecation notice. An example:

```php
class A {
    public function test() { var_dump($this); }
}

// Note: Does NOT extend A
class B {
    public function callNonStaticMethodOfA() { A::test(); }
}

(new B)->callNonStaticMethodOfA();

// Deprecated: Non-static method A::test() should not be called statically
// Notice: Undefined variable $this
NULL
```

Note that this only applies to calls from an incompatible context. If class B extended from A the call would be allowed without any notices.

Refer back to Chapter 36 - Reclassification and Removal of E_STRICT Notices.

Reserved words

It is no longer possible to use the following class, interface and trait names (case-insensitive):

```
bool
int
float
string
null
false
true
```

This applies to class/interface/trait declarations, class_alias() and use statements.

Furthermore the following class, interface and trait names are now reserved for future use, but do not yet throw an error when used:

```
resource
object
mixed
numeric
```

Refer back to Chapter 1 - Scalar Type Hints.

Parenthesis requirement for `yield`

The yield language construct no longer requires parentheses when used in an expression context. It is now a right-associative operator with precedence between the "print" and "⇒" operators. This can result in different behavior in some cases, for example:

```
echo yield -1;
// Was previously interpreted as
echo (yield) - 1;
// And is now interpreted as
echo yield (-1);

yield $foo or die;
// Was previously interpreted as
yield ($foo or die);
// And is now interpreted as
(yield $foo) or die;
```

Such cases can always be resolved by adding additional parentheses.

Refer back to Chapter 16 - Abstract Syntax Tree.

Other removals

- Removed ASP (`<%`) and script (`<script language=php>`) tags. (RFC: https://wiki.php.net/rfc/remove_alternative_php_tags)
- Removed support for assigning the result of new by reference.
- Removed support for scoped calls to non-static methods from an incompatible `$this` context. See details in https://wiki.php.net/rfc/incompat_ctx.
- Removed support for `#`-style comments in ini files. Use `;`-style comments instead.
- `$HTTP_RAW_POST_DATA` is no longer available. Use the `php://input` stream instead.

Standard Library Changes

- `substr()` now returns an empty string instead of `FALSE` when the truncation happens on boundaries.
- `call_user_method()` and `call_user_method_array()` no longer exists.
- `ob_start()` no longer issues an `E_ERROR`, but instead an `E_RECOVERABLE_ERROR` in case an output buffer is created in an output buffer handler.
- The internal sorting algorithm has been improved, what may result in different sort order of elements that compare as equal.
- Removed `dl()` function on fpm-fcgi.
- `setcookie()` with an empty cookie name now issues an `E_WARNING` and doesn't send an empty set-cookie header line anymore.

Other Changes

Curl

- Removed support for disabling the CURLOPT_SAFE_UPLOAD option. All curl file uploads must use the curl_file / CURLFile APIs.

Date

- Removed $is_dst parameter from mktime() and gmmktime().

DBA

- dba_delete() now returns false if the key was not found for the inifile handler, too.

GMP

- Requires libgmp version 4.2 or newer now.
- gmp_setbit() and gmp_clrbit() now return FALSE for negative indices, making them consistent with other GMP functions.

Intl

- Removed deprecated aliases datefmt_set_timezone_id() and IntlDateFormatter::setTimeZoneID(). Use datefmt_set_timezone() and IntlDateFormatter::setTimeZone() instead.

libxml

- Added LIBXML_BIGLINES parser option. It's available starting with libxml 2.9.0 and adds suppport for line numbers >16-bit in the error reporting.

Mcrypt

- Removed deprecated mcrypt_generic_end() alias in favor of mcrypt_generic_deinit().
- Removed deprecated mcrypt_ecb(), mcrypt_cbc(), mcrypt_cfb() and mcrypt_ofb() functions in favor of mcrypt_encrypt() and mcrypt_decrypt() with an MCRYPT_MODE_* flag.

Session

- session_start() accepts all INI settings as array. e.g. [‘cache_limiter’⇒‘private’] sets session.cache_limiter=private. It also supports ‘read_and_close’ which closes session data immediately after read data.
- Save handler accepts validate_sid(), update_timestamp() which validates session ID existence, updates timestamp of session data. Compatibility of old user defined save handler is retained.
- SessionUpdateTimestampHandlerInterface is added. validateSid(), updateTimestamp() is defined in the interface.
- session.lazy_write(default=On) INI setting enables only write session data when session data is updated.

Opcache

- Removed opcache.load_comments configuration directive. Now doc comments loading costs nothing and always enabled.

OpenSSL

- Removed the “rsa_key_size” SSL context option in favor of automatically setting the appropriate size given the negotiated crypto algorithm.
- Removed “CN_match” and “SNI_server_name” SSL context options. Use automatic detection or the “peer_name” option instead.

PCRE:

- Removed support for /e (PREG_REPLACE_EVAL) modifier. Use preg_replace_callback() instead.

PDO_pgsql:

- Removed PGSQL_ATTR_DISABLE_NATIVE_PREPARED_STATEMENT attribute in favor of ATTR_EMULATE_PREPARES.

Standard:

- Removed string category support in setlocale(). Use the LC_* constants instead.
- Removed set_magic_quotes_runtime() and its alias magic_quotes_runtime().

JSON:

- Rejected RFC 7159 incompatible number formats in json_decode string - top level (07, 0xff, .1, -.1) and all levels ([1.], [1.e1])
- Calling json_decode with 1st argument equal to empty PHP string or value that after casting to string is empty string (NULL, FALSE) results in JSON syntax error.

Stream:

- Removed set_socket_blocking() in favor of its alias stream_set_blocking().

XSL:

- Removed xsl.security_prefs ini option. Use XsltProcessor::setSecurityPrefs() instead.