

---

# **php-v8 Documentation**

**Bogdan Padalko**

**Jul 15, 2018**



---

# Contents

---

<b>1</b>	<b>About</b>	<b>1</b>
<b>2</b>	<b>Hello, world!</b>	<b>3</b>
<b>3</b>	<b>Content</b>	<b>5</b>
3.1	Getting started . . . . .	5
3.2	Performance tricks . . . . .	7
3.3	Release process . . . . .	12



# CHAPTER 1

---

## About

---

`php-v8` is a PHP 7.x extension that brings [V8 JavaScript engine API](#) to PHP with some abstraction in mind and provides an accurate native V8 C++ API implementation available from PHP.



## CHAPTER 2

---

Hello, world!

---

```
<?php
use V8\{Isolate, Context, StringValue, Script};

$script = "'Hello' + ', World!'";

$isolate = new Isolate();
$context = new Context($isolate);

echo (new Script($context, new StringValue($isolate, $script))
    ->run($context)
    ->value(), PHP_EOL;
```





## 3.1 Getting started

### 3.1.1 About

`php-v8` is a PHP 7.x extension that brings V8 JavaScript engine API to PHP with some abstraction in mind and provides an accurate native V8 C++ API implementation available from PHP.

#### Key features:

- provides up-to-date JavaScript engine with recent ECMA features supported;
- accurate native V8 C++ API implementation available from PHP;
- solid experience between native V8 C++ API and V8 API in PHP;
- no magic; no assumptions;
- does what it is asked to do;
- hides complexity with isolates and contexts scope management under the hood;
- provides a both-way interaction with PHP and V8 objects, arrays and functions;
- execution time and memory limits;
- multiple isolates and contexts at the same time;
- it works.

With this extension almost everything that the native V8 C++ API provides can be used. It provides a way to pass PHP scalars, objects and functions to the V8 runtime and specify interactions with passed values (objects and functions only, as scalars become js scalars too). While specific functionality will be done in PHP user space rather than in this C/C++ extension, it lets you get into V8 hacking faster, reduces time costs and gives you a more maintainable solution. And it doesn't make any assumptions for you, so you stay in control, it does exactly what you ask it to do.

With php-v8 you can even implement NodeJs in PHP. Not sure whether anyone should/will do this anyway, but it's doable.

### 3.1.2 Demo

Here is a [Hello World](#) from V8 *Getting Started* developers guide page implemented in raw php-v8:

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

use V8\{
    Isolate,
    Context,
    StringValue,
    Script,
};

$isolate = new Isolate();
$context = new Context($isolate);
$source  = new StringValue($isolate, "'Hello' + ', World!'");

$script = new Script($context, $source);
$result = $script->run($context);

echo $result->value(), PHP_EOL;
```

### 3.1.3 Installation

#### Requirements

##### V8

You will need a recent v8 Google JavaScript engine version installed. At this time v8  $\geq$  6.6.313 required.

##### PHP

**This extension requires PHP  $\geq$  7.2.** Last version that supports PHP 7.1 is v0.2.2 and for PHP 7.0 is v0.1.9.

##### OS

This extension works and tested on x64 Linux and macOS. As of written it is Ubuntu 16.04 LTS Xenial Xerus, amd64 and macOS 10.12.5. Windows is not supported at this time.

#### Quick guide

##### Docker

There is default phpv8/php-v8 docker image with basic dependencies to evaluate and play with php-v8:

```
docker run -it phpv8/php-v8 bash -c "php test.php"
```

## Ubuntu

There is

```
$ sudo add-apt-repository -y ppa:ondrej/php
$ sudo add-apt-repository -y ppa:pinepain/php
$ sudo apt-get update -y
$ sudo apt-get install -y php7.2 php-v8
$ php --ri v8
```

While [pinepain/php](#) PPA targets to contain all necessary extensions with dependencies, you may find following standalone PPAs useful:

- [pinepain/libv8-6.6](#)
- [pinepain/experimental](#)
- [phpv8/php-v8](#)

## OS X (homebrew)

```
$ brew tap homebrew/dupes
$ brew tap homebrew/php
$ brew tap phpv8/tap
$ brew install php71 php71-v8
$ php --ri v8
```

For macOS php-v8 formulae and dependencies provided by [phpv8/tap](#) tap.

## Building php-v8 from sources

```
git clone https://github.com/phpv8/php-v8.git
cd php-v8
phpize && ./configure && make
make test
```

To install extension globally run

```
$ sudo make install
```

## 3.2 Performance tricks

If you:

- use php-v8 extension for short-lived tasks or
- you have your Context short-lived

it is likely that you won't be able benefit from V8 runtime optimizations. However, you can still improve your performance.

*Important note:* all caching techniques are V8 version-specific and platform specific, some caches won't even work on different CPU with different instructions set, you have to test following techniques for your environment and infrastructure and be ready to fallback to raw, cache-less flow.

Let's say you have an typical Hello, world! script:

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

use V8\{
    Isolate,
    Context,
    StringValue,
    Script,
};

$script_source_string = "function say() { return 'Hello' + ', World!'; say()";

$isolate = new Isolate();
$context = new Context($isolate);
$source = new StringValue($isolate, $script_source_string);

$script = new Script($context, $source);
$result = $script->run($context);

echo $result->value(), PHP_EOL;
```

Let's reshape it a bit to make it more suitable for further tweaks by introducing ScriptCompiler:

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

use V8\{
    Isolate,
    Context,
    StringValue,
    ScriptCompiler,
};

$script_source_string = "function say() { return 'Hello' + ', World!'; say()";

$isolate = new Isolate();
$context = new Context($isolate);
$source_string = new StringValue($isolate, $script_source_string);
$source = new ScriptCompiler\Source($source_string);
$script = ScriptCompiler::compile($context, $source);

$result = $script->run($context);

echo $result->value(), PHP_EOL;
```

### 3.2.1 Script code cache

Using script code cache could boost your performance from 5% to 25% or even more, it largely depends on your script and host. On slower machines it may give you better result in terms of performance gain %%, while on faster it may be not so large, according to performance benchmark.

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

use V8\{
    Isolate,
    Context,
```

(continues on next page)

(continued from previous page)

```

    StringValue,
    ScriptCompiler,
};

$script_source_string = "function say() { return 'Hello' + ', World!'; say()};";

$isolate      = new Isolate();
$context      = new Context($isolate);
$source_string = new StringValue($isolate, $script_source_string);
$source       = new ScriptCompiler\Source($source_string);

// Generating script cache. Normally you want to cache this data somewhere else
// either on filesystem, in database or in memory. Redis could be your friend
// but don't let it be your memory hog.
$unbound_script = ScriptCompiler::compileUnboundScript($context, $source);
$cached_data = ScriptCompiler::createCodeCache($unbound_script, $source_string);

// Here we utilize script cache
$source = new ScriptCompiler\Source($source_string, null, $cached_data);
$script = ScriptCompiler::compile($context, $source, ScriptCompiler::OPTION_CONSUME_
↳CODE_CACHE);

if ($cached_data->isRejected()) {
    throw new RuntimeException('Script code cache rejected!');
}

$result = $script->run($context);

echo $result->value(), PHP_EOL;

```

### 3.2.2 Isolate startup data

Startup data can speedup your context creation by populating them with script run result. It can save from 1% to 3%, so it's not so effective as script code cache, however, the benchmark was done on using quite simple example, so if you have a lot of entities that you need to bootstrap your context with, your saving may be more.

```

<?php declare(strict_types=1);

use V8\{
    Isolate,
    Context,
    StringValue,
    ScriptCompiler,
    StartupData,
};

$script_source_string = "function say() { return 'Hello' + ', World!'; say()};";

// Same here, you are likely want to store it in some quick and cheap to access_
↳storage
$startup_data = StartupData::createFromSource($script_source_string);

$isolate      = new Isolate($startup_data);
$context      = new Context($isolate);
$source_string = new StringValue($isolate, $script_source_string);

```

(continues on next page)

```
$source          = new ScriptCompiler\Source($source_string);  
$script = ScriptCompiler::compile($context, $source);  
$result = $script->run($context);  
echo $result->value(), PHP_EOL;
```

### 3.2.3 Combining both approaches

Combining both techniques is your friend in boosting performance:

```
<?php declare(strict_types=1);  
  
use V8\  
    Isolate,  
    Context,  
    StringValue,  
    ScriptCompiler,  
    StartupData,  
};  
  
$script_source_string = "function say() { return 'Hello' + ', World!'}; say()";  
  
$startup_data = StartupData::createFromSource($script_source_string);  
  
$isolate      = new Isolate($startup_data);  
$context      = new Context($isolate);  
$source_string = new StringValue($isolate, $script_source_string);  
$source       = new ScriptCompiler\Source($source_string);  
  
$unbound_script = ScriptCompiler::compileUnboundScript($context, $source);  
$cached_data    = ScriptCompiler::createCodeCache($unbound_script, $source_string);  
  
$source = new ScriptCompiler\Source($source_string, null, $cached_data);  
$script = ScriptCompiler::compile($context, $source, ScriptCompiler::OPTION_CONSUME_  
    ↳CODE_CACHE);  
  
if ($cached_data->isRejected()) {  
    throw new RuntimeException('Script code cache rejected!');  
}  
  
$script = ScriptCompiler::compile($context, $source);  
  
$result = $script->run($context);  
  
echo $result->value(), PHP_EOL;
```

### 3.2.4 Benchmarks

Note, that your mileage may vary so you are highly encouraged to run benchmarks located under project's root /pref folder by yourself on your hardware, in your infra and even with your js script.

## From Ubuntu in Docker on macOS

4 cores, 16Gb memory

```
# php -v
PHP 7.2.2-3+ubuntu16.04.1+deb.sury.org+1 (cli) (built: Feb 6 2018 16:11:23) ( NTS )
Copyright (c) 1997-2018 The PHP Group
Zend Engine v3.2.0, Copyright (c) 1998-2018 Zend Technologies
    with Zend OPcache v7.2.2-3+ubuntu16.04.1+deb.sury.org+1, Copyright (c) 1999-2018,
↳by Zend Technologies

# php --ri v8
V8 support => enabled
Version => v0.2.1-master-dev
Revision => 5d7c3e4
Compiled => Feb 25 2018 @ 11:29:00

V8 Engine Compiled Version => 6.6.313
V8 Engine Linked Version => 6.6.313
```

*Less is better*

subject	mode	stdev	rstdev	diff ( <i>less is better</i> )
Cold Isolate, no code cache	3,602.599us	49.778us	1.38%	+26.98%
Cold Isolate, with code cache	2,885.638us	39.775us	1.36%	+2.86%
Warm Isolate, no code cache	3,489.959us	44.036us	1.27%	+22.46%
Warm Isolate, with code cache	2,813.156us	43.351us	1.53%	0.00%

## From macOS host

4 cores, 16Gb memory

```
$ php -v
PHP 7.2.2 (cli) (built: Feb 1 2018 11:50:40) ( NTS )
Copyright (c) 1997-2018 The PHP Group
Zend Engine v3.2.0, Copyright (c) 1998-2018 Zend Technologies
    with Zend OPcache v7.2.2, Copyright (c) 1999-2018, by Zend Technologies

$ php --ri v8
V8 support => enabled
Version => v0.2.1-master-dev
Revision => 5d7c3e4
Compiled => Feb 25 2018 @ 11:42:00

V8 Engine Compiled Version => 6.6.313
V8 Engine Linked Version => 6.6.313
```

subject	mode	stdev	rstdev	diff ( <i>less is better</i> )
Cold Isolate, no code cache	8,732.585us	97.889us	1.11%	+6.90%
Cold Isolate, with code cache	8,290.880us	141.583us	1.69%	+1.78%
Warm Isolate, no code cache	8,722.684us	104.547us	1.19%	+6.68%
Warm Isolate, with code cache	8,194.924us	70.345us	0.85%	0.00%

## 3.3 Release process

Here you can find basics to the release process

### 3.3.1 Release php-v8

#### GitHub release

1. Make sure current state is ready for release:
  - All relevant PR merged and issues closed.
  - Build passed.
2. Prepare release notes by creating release draft on github.
3. Update `PHP_V8_VERSION` to have desired version and set `PHP_V8_REVISION` to release in `php_v8.h`.
4. Run `./scripts/refresh-package-xml.php -f` to update `package.xml` with proper php-v8 version and update directories and files tree.
5. Update `package.xml` `<notes>` with release notes. Keep eye on special characters to properly escape them, e.g. `>` should be written as `&gt;`; instead.
6. Commit all changes with `Prepare X.Y.Z release` commit message.
7. Push this commit and make sure it will pass the build.
8. Tag it with `vX.Y.Z` tag and push. Create github release from a draft prepared in step above.
9. Close relevant milestone, if any.
10. **Run `./scripts/subsplit.sh` to update `php-v8-stubs` which are available in a separate read-only repository to match packagist and composer expectations.**

#### PECL release

1. Run `pecl package` in your build machine (it's normally vagrant box used for php-v8 development). It should create `v8-X.Y.Z.tgz` file.
2. Log in to PECL and upload file from previous step at <https://pecl.php.net/release-upload.php>. Verify that release info is accurate and confirm release.

#### Docker image release

1. Go into `pinepain/dockerfiles php-v8` folder.
2. Make sure you have valid stable and latest versions in `Makefile`.
3. To avoid caching, run `make clean-stable` to remove any image for the current stable version and `make clean-latest` to do the same for the current latest version.
4. Run `make stable` to build and upload current stable version and `make latest` to build and upload the latest version.



### After all

1. Update `js-sandbox` dependencies, if required, to use latest `php-v8` and other dependencies, if any.
2. Update `PHP_V8_VERSION` to the next version and set `PHP_V8_REVISION` to `dev` in `php_v8.h`.
3. Commit changes with `Back to dev [skip ci]` message and push them to master.

### 3.3.2 Release libv8

Currently Docker is the recommended way to distribute and use both `php-v8` and `libv8` itself. We also support building `libv8` in macOS with Homebrew via `phpv8/tap` tap.

To track v8 changes you can use these links:

- <https://github.com/v8/v8/commits/master/include/v8.h> - to keep track on v8 upstream changes
- <https://omahaproxy.appspot.com/> - to keep track v8 channel(version) heads and what version is used in chrome

### Building docker image

1. Build `phpv8/libv8` docker image, tag it with the relevant v8 full version and push to Docker Hub. Hint: use `Makefile`.
2. You may want to set proper V8 version in `php-v8` by updating it in `.travis.yml`.

### After docker images rebuilt/published

1. Update min required `libv8` version in `php-v8 config.m4`, `V8_MIN_API_VERSION_STR=X.Y.Z`.
2. If there was new docker images published, update reference to them in `php-v8 .travis.yml` and in `php-v8 Dockerfile`, and set proper V8 and TAG value there.
3. Update reference to `v8@X.Y` in `php-v8 CMakeLists.txt` on minor version bump.
4. Also, update references to v8 version in `php-v8/scripts/provision/provision.sh`, it's normally could be done by replacing old version with new, e.g. `6.3 => 6.4`.
5. On every version bump update `php-v8 README.md` file with proper min v8 version required/tested.
6. **Make sure** you tested `php-v8` locally first before pushing to remote, upgrading v8 could be tricky as it may break BC even in patch releases.
7. Note, that doing all this in a separate branch and merging that later into master is a nice and safe idea (note, you may skip PR overhead and do fast-forward merge locally to master).
8. Commit message should state that it is v8 version bump, e.g. `Require libv8 >= X.Y.Z`
9. Push changes and make sure build is green. If not, fix code/update tests and repeat.

### Building packages for macOS Homebrew

1. **Skip this step if you are updating v8 patch release version.** If it is a minor version bump, create new `v8@X.Y` formula.
2. **Skip this step if you are updating v8 patch release version.** Create new `v8:X.Y` Package on bintray for it.
3. Remove/reset formula `revision` if it is version bump and not rebuild.

4. Build `v8@X.Y` (locally or with TravisCI if it provides relevant macOS version) and publish.