Velocity persistent workers for PHP developers

What changes for PHP code when Exponential runs under Velocity, the recommended application server - workers that boot once and serve many requests, the warm-up, what the kernel resets between requests, per-request...

Source file: `doc/eng-us/help/php-developers/velocity-persistent-workers-for-php-developers.md` Last changed 2026-11-04
https://doc.exponential.earth/learninminutes/exphelp/viewer/help/php-developers/velocity-persistent-workers-for-php-developers.md

This page is for PHP developers whose code runs under Exponential Velocity, the application server that is part of Exponential as of 6.0.15 and the recommended engine for every stage (Velocity is part of Exponential 6.0.15). Exponential was built for one PHP process per request; Velocity boots once and keeps a pool of workers that serve many requests each. Most code runs unchanged. This page lists what does not, and why.

Contents

1. The model
2. What the kernel and the engine reset for you
3. What you must not do
4. Per-request work: eZExecution::requestKey()
5. cleanExit() ends the request with an exception
6. New classes and new settings
7. Deploying a PHP change
8. Testing your code for Velocity

The model

	PHP-FPM, mod_php	Velocity (`qbix` engine)
process per request	fresh	a worker kept between requests (`[ServerSettings] ForkPerRequest=disabled`, the default since 30 September 2026)
boot	every request	once, in the warm-up, then inherited by the workers
static properties	gone after the request	**kept**, unless reset
loaded classes	loaded per request	kept for the worker's life
settings	read per request (cached)	read once per worker
SAPI	`fpm-fcgi`, `apache2handler`	the command line SAPI; the engine makes `header()`, `setcookie()` and friends work

Two other engines exist: FrankenPHP in classic mode (a clean state per request, like PHP-FPM)

and PHP's built-in server for development (Velocity engines, FrankenPHP for PHP developers).

What the kernel and the engine reset for you

- **Class statics.** Between requests a worker puts every class's statics back to what they were after its warm-up. The engine reads the declared default values from the source.
- **Shutdown functions and handlers.** The shutdown functions a request registered are run and then forgotten; `eZExecution`'s cleanup and fatal error handlers, its clean exit flag and its own shutdown functions belong to one request.
- **Superglobals and the request.** `$_GET`, `$_POST`, `$_COOKIE`, `$_SERVER` and `$_SESSION` are per request; `REQUEST_TIME` and `REQUEST_TIME_FLOAT` are per request (engine 0.0.4.42 and later).
- **The warm-up's leftovers.** From engine 0.0.4.51 the warm-up's request is ended before the workers fork: its headers, handlers, session, ini changes and output buffers are put back.

What you must not do

Pattern	Why it breaks under workers	Instead
function-level `static $cache` keyed by nothing	survives between requests: one siteaccess's data reaches the next request	a class static (reset by the worker), keyed by what it depends on
a `true` flag to "register once"	the flag is kept from the warm-up; the second request registers nothing	key it by `eZExecution::requestKey()`
assigning a class static after the class body (`acmeFoo::$list = ...;` at file level)	the worker resets statics to the declared value, and `include_once` does not run the assignment again	declare the value in the class body
reading `$_POST`, `$_GET` before the kernel prepared them	stale or empty	`eZHTTPTool`
a long `while ( ob_get_level() > 0 ) ob_end_clean();`	the worker keeps a buffer that cannot be removed; the loop never ends	`eZExecution::discardOutputBuffers()`

The details and examples are on Static state traps.

Per-request work: eZExecution::requestKey()

Guard a registration that must happen once per request with the request's key (its `REQUEST_TIME_FLOAT`):

PHP

```php
if ( self::$registered !== eZExecution::requestKey() )
{
    self::$registered = eZExecution::requestKey();
    register_shutdown_function( array( __CLASS__, 'flush' ) );
}
```

`eZExecution::addCleanupHandler()` and `addFatalErrorHandler()` may simply be called on every request: the same handler is kept once. Event listeners belong in `site.ini [Event]`

`Listeners[]`, which the kernel attaches correctly (Events, listeners and filters).

cleanExit() ends the request with an exception

A view that sends its own output (a download, JSON) ends with `eZExecution::cleanExit()`. Under Velocity this ends the request by an exception the server handles. **Never wrap it in** `catch ( Exception $e )` (or `Throwable`): the catch swallows the end of the request, the kernel goes on, and the page is appended to your output.

PHP

```php
header( 'Content-Type: application/json; charset=utf-8' );
echo json_encode( $data );
eZExecution::cleanExit();           // not inside a try block that catches Exception
```

New classes and new settings

- A worker keeps the class map it started with: a class added after the workers started does not exist for them. Regenerate autoloads and restart the workers.
- A setting that names a class (a handler, a listener, an `Implementation[]`) breaks workers that do not know the class. Deploy the code first, restart, then the settings, and guard with `class_exists()` where a missing class must not break pages.
- Settings are read once per worker: after a change, clear the INI cache (`--clear-tag=ini`) and restart.

Deploying a PHP change

Shell

```shell
php bin/php/console exp:velocity deploy --dry-run     # what it would do
php bin/php/console exp:velocity deploy               # the usual case
php bin/php/console exp:velocity deploy --kernel      # a kernel class was added or renamed
```

One command does the steps in order and prints PASS, FAIL or SKIP for each: autoloads, the INI cache, template and override caches, the engine archive (only when `kernel/`, `lib/` or `autoload/` changed; every file parsed first), a PHP-FPM reload, the Velocity restart, then the caches that hold rendered output (Velocity engines: deploying a PHP change).

Testing your code for Velocity

1. Run the page twice in a row on the same worker, with two different users, two siteaccesses, two languages. Output from the first request must never appear in the second.
2. Compare with PHP's built-in server (`--engine=php`), which runs each request fresh: a bug that only Velocity shows is state kept between requests.
3. Look at the debug bar's **Server** tab to confirm which engine rendered the page.
4. PHPUnit runs all tests in one process: a test that changes static state and does not reset it shows the same class of bug (PHPUnit without a database).

Related pages

- Velocity: running Exponential in a persistent-worker web server
- Deploying with Velocity
- Velocity engine upgrade notes
- Free documentation: this page is free to copy, translate and improve under the GNU FDL 1.3.

> **Become an Exponential Partner**
>
> Partners share their information, provide funding, contribute back and develop the products with us, and that is how Exponential grows. A partner website and dashboard are planned for mid-2027: please become a partner today to speed up our growth rate and land new features faster.
>
> Partners: start here | How to apply

Exponential Copyright © 1998-2026 Exponential Foundation and 7x
Permission is granted to copy, distribute and/or modify this document under the terms of the GNU Free Documentation License, Version 1.3 or any later version published by the Free Software Foundation; with no Invariant Sections, no Front-Cover Texts, and no Back-Cover Texts. A copy of the license is included in the section entitled "GNU Free Documentation License".
This documentation is free documentation, licensed under the GNU Free Documentation License 1.3 or later:
https://www.gnu.org/licenses/fdl-1.3.html.