

Cache Project

Consistent PSR-6/PSR-16 Cache Wiring Across Derafu Packages

Anonymous · 22/09/2026 · #php

Consistent PSR-6/PSR-16 Cache Wiring Across Derafu Packages

[github](#) [derafu/cache](#) [last commit](#) [last monday](#)  [CI](#) [passing](#) [code size](#) [65.8 KiB](#) [open issues](#) [0](#)

[downloads](#) [275](#) [downloads](#) [121 this month](#)

A small, opinionated layer over [symfony/cache](#) for the *local* backends (no connection, no credentials, no network round trip) — plus a handful of things PSR-6/PSR-16 and [symfony/cache](#) genuinely don't provide on their own.

Why

[symfony/cache](#)'s own file-based adapters (`PhpFilesAdapter`, `FilesystemAdapter`) accept a `null` directory and silently fall back to one you'd have to read the source to find. A cache backed by the filesystem that nobody can point to exactly is a cache nobody can clear with confidence — [derafu/cache](#)'s adapters make `$namespace` and `$directory` required instead, so that decision is never made for you.

Past that one guardrail, this package adds a handful of things PSR-6/PSR-16 and [symfony/cache](#) genuinely don't provide on their own: a safe way to build cache keys without hand-rolling sanitization, a stampede-safe “compute and remember” helper, and a way to inspect or clear a cache directory without needing to reconstruct whatever wrote to it.

It is **not** a general abstraction over every backend [symfony/cache](#) supports. PSR-6/PSR-16 already are that abstraction — building another one on top would just be indirection. See [Scope](#) below.

Installation

```
composer require derafu/cache
```

`ext-apcu` (for `LocalCacheBackend::Apcu`) and `symfony/console` (for the bundled commands) are both entirely optional — see [composer.json's suggest](#).

Scope: Local Only

`LocalCacheBackend` and `LocalCacheFactory` cover exactly the backends that need no connection, no credentials, and no network round trip: they live in this process, in shared memory on this server, or on this filesystem.

```
enum LocalCacheBackend: string
{
    case None = 'none';           // Never caches anything (NullAdapter).
    case Memory = 'memory';       // Dies with the process (ArrayAdapter).
    case Apcu = 'apcu';           // Shared memory, survives requests (ApcuAdapter).
    case Filesystem = 'filesystem'; // Serialized files, supports objects
    (FilesystemAdapter).
    case PhpFiles = 'php_files';  // Plain PHP array files, OPcache-friendly
    (PhpFilesAdapter).
}
```

Redis, Memcached, Couchbase, PDO, and every other connection-based backend `symfony/cache` supports are deliberately out of scope — not a gap, a boundary. They have a fundamentally different construction shape (a connection or DSN, not a namespace/directory pair) and none of the “silent default” problem this package exists to close off: `RedisAdapter/MemcachedAdapter` already force you to pass a real connection, no `null` fallback to guard against. Building one directly from `symfony/cache` isn’t a workaround, it’s simply what those backends need — `$pool->clear()` (part of PSR-6 itself) already works on them the same way it works on everything else.

PhpFilesCache and FilesystemCache

```
use Derafu\Cache\Adapter\PhpFilesCache;
use Derafu\Cache\Adapter\FilesystemCache;

// OPcache-friendly: values must be var_export()-able (arrays/scalars, no objects).
$pool = new PhpFilesCache('my_package', '/var/cache/my_package');

// Serialized: supports arbitrary values, including objects. No OPcache benefit.
$pool = new FilesystemCache('my_package', '/var/cache/my_package');
```

Both extend the matching `symfony/cache` adapter directly, under the same name, so anyone already familiar with `symfony/cache` recognizes immediately what each one does. `$namespace` and `$directory` are both required — `$defaultLifetime` (3600 by default) is the only parameter allowed to have one, since it isn’t a “where does my data live” decision.

LocalCacheFactory

For the case where the backend itself is a runtime choice — from configuration, the way a per-feature

cache: `memory|filesystem` setting would — rather than something you already know at the call site:

```
use Derafu\Cache\LocalCacheFactory;
use Derafu\Cache\Enum\LocalCacheBackend;
use Symfony\Component\Cache\Adapter\ArrayAdapter;

$pool = LocalCacheFactory::pool(LocalCacheBackend::Filesystem, 'my_package',
    '/var/cache/my_package');

// PSR-16 instead of PSR-6 — the same pool, wrapped in Symfony's Psr16Cache bridge.
$cache = LocalCacheFactory::simple(LocalCacheBackend::Memory, 'my_package');

// Tag-aware — $item->tag([...]) on save, invalidateTags([...]) to drop a whole group
at once.
$cache = LocalCacheFactory::taggedPool(LocalCacheBackend::Apcu, 'my_package');

// Several pools, fastest first, with automatic backfill on a slower-tier hit.
$pool = LocalCacheFactory::layered([new ArrayAdapter(), $filesystemPool]);
```

`$directory` is required for `Filesystem/PhpFiles`, ignored for `None/Memory/Apcu` — passing it for those three is harmless, omitting it for the other two throws.

`LocalCacheBackend::None` (backed by Symfony's own `NullAdapter`) is how “no caching” is expressed — not a runtime flag a consumer has to check, but a pool like any other. A decorator that always receives a `CacheItemPoolInterface` needs no special case for “caching is off”: it just does nothing useful when this happens to be the pool it got.

Memoizer: Compute-and-Cache Without Reinventing Stampede Protection

```
use Derafu\Cache\Memoizer;

$memoizer = new Memoizer();

$value = $memoizer->remember($pool, 'some_key', function () {
    return expensive_computation();
}, ttl: 3600);
```

Every hand-written PSR-6 caching decorator ends up rewriting the same “is it there? no? compute it, store it, return it” dance — `getItem()` → `isHit()` → `set()` → `save()`. That naive version has no protection against a cache stampede: under concurrent access, every process that hits a cold cache computes and saves independently.

Every `symfony/cache` adapter also implements Symfony's own `CacheInterface::get()` contract, which does the same thing with real locking — only one process computes, the rest wait for it and

read the result. `Memoizer::remember()` uses that path automatically whenever the given pool supports it, and falls back to the naive dance only for a PSR-6 implementation that isn't a Symfony adapter (no locking possible without Symfony's own contract). `$ttl` means exactly what it means for `expiresAfter()` — `null` leaves the pool's default lifetime in effect, `0` means the entry never expires, a positive integer is seconds until expiration. Nothing here redefines that; a decorator that also needs a "don't even try to cache" switch should express it as its own separately-named parameter, or — more simply — accept a `LocalCacheBackend::None` pool.

CacheKey: One Safe Way to Build a Key

```
use Derafu\Cache\CacheKey;

$cacheKey = new CacheKey();

$key = $cacheKey->build('my_package', 'ExampleWorker', ['api_resource' => true]);
// "my_package.ExampleWorker.e381541fbf7da0e4036f1cc881592fce"
```

PSR-6 forbids `{ } ( ) / \ @ :` in keys but sanitizes nothing for you — building a safe, collision-resistant key from real context (a class name with backslashes, a filter array) is left entirely to each caller.

`CacheKey::build()` normalizes each part: a string gets its forbidden characters *replaced* (never dropped — dropping them could collide `App\FooBar` with `App\Foo\Bar`, both becoming `AppFooBar`), any other scalar is used as-is, and anything else (an array, an object) is hashed. If the assembled key would exceed 250 bytes (Memcached's real limit, a safe ceiling for any backend), the whole thing collapses to the prefix plus a hash of the full key — long context never silently breaks a backend with an actual key-length limit.

`CacheKey` implements `Derafu\Cache\Contract\CacheKeyInterface` so a consumer that needs key logic driven by what's actually being cached — not just its class name — can inject its own implementation of that one-method interface instead of subclassing `CacheKey` and reaching into its private sanitizing helpers.

CacheDirectory: Inspecting and Clearing by Path Alone

```
use Derafu\Cache\CacheDirectory;

$directory = new CacheDirectory('/var/cache/my_package');

$directory->stats(); // ['count' => 42, 'totalSize' => 1048576, 'oldest' =>
1700000000, 'newest' => 1700003600]
$directory->clear(); // int — how many files were deleted.
```

`stats()` is a real, new capability — neither PSR-6 nor `symfony/cache` expose a way to ask any pool how many entries it holds or how much space they use, for any backend. `clear()` is not:

`CacheItemPoolInterface::clear()` is already part of PSR-6 and works fine when you have a live pool

instance in hand. `CacheDirectory::clear()` exists for exactly the case where you don't — an admin command that only knows a path, not which adapter type (namespace, TTL, backend) originally wrote there.

CacheWarmer and WarmableInterface: Warming Without a Kernel

```
use Derafu\Cache\CacheWarmer;
use Derafu\Cache\Contract\WarmableInterface;

final class ExampleWarmable implements WarmableInterface
{
    public function warmup(): void { /* ... */ }
}

$failures = (new CacheWarmer())->warmup([$warmableA, $warmableB]);
// list<array{warmable: WarmableInterface, exception: Throwable}> — empty if
// everything succeeded.
```

`symfony/cache` has no warmup concept at all — only `PruneableInterface` (the opposite: removing *expired* entries). Symfony's real `CacheWarmerInterface` lives in `symfony/http-kernel`, tied to the full framework Kernel's boot lifecycle: it warms the *framework's own* caches (compiled container, routing) into a temporary location and atomically swaps them in, specifically because a half-written container cache would take the whole application down.

Warming an application-level PSR-6 cache doesn't carry that risk — if a warmable fails partway through, the next real read just recomputes that one entry. So `CacheWarmer` doesn't try to replicate the atomic swap or the Kernel's boot-order resolution: it runs each warmable in order, keeps going if one throws, and reports every failure at the end instead of stopping at the first one.

There's deliberately no bundled warmup *command*: a generic one would need to discover which `WarmableInterface` instances exist in a given application and how to construct each one — almost none will have a no-argument constructor, since a real warmable needs whatever it's warming. That discovery is exactly what a DI container is for, which this package chooses not to depend on (see [Console Commands](#) below). The intended shape is a small, application-specific command that constructs the warmables it actually has and calls `(new CacheWarmer())->warmup($warmables)`.

Console Commands (Optional)

```
use Derafu\Cache\Console\ClearLocalCacheCommand;
use Derafu\Cache\Console\LocalCacheStatsCommand;

$app->add(new ClearLocalCacheCommand());
$app->add(new LocalCacheStatsCommand());
```

```
php bin/console derafu:local-cache:clear /var/cache/my_package
php bin/console derafu:local-cache:stats /var/cache/my_package
```

Both wrap `CacheDirectory` and require only a path. Named `derafu:local-cache:*`, not the more obvious `cache:clear/cache:stats` — that name is already taken in any real Symfony full-stack app (`symfony/framework-bundle` ships its own `cache:clear`, for the framework's own internal cache), and a generic name from a library meant to be added into someone else's `Application` is a collision waiting to happen. Still want a shorter name? Rename after construction: `(new ClearLocalCacheCommand())->setName('cache:clear-local')`.

Neither command is registered automatically — this package has no DI container of its own to register commands into. `symfony/console` is a `suggest`, not a hard requirement: Composer's autoloading is lazy, so a consumer that never references `Console\*` pays nothing for it even without `symfony/console` installed.

Requirements

PHP 8.5+. Depends on `symfony/cache`, `psr/cache`, and `psr/simple-cache`.

Last updated on 22/09/2026