

Memoizer and Keys

Memoizer and Keys

Anonymous · 10/10/2026

Memoizer and Keys

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.

Last updated on 10/10/2026