Content Cache

What the built-in content cache does and does not speed up, and how to swap its backend.

Anonymous · 21/08/2026

Content cache

Every content plugin that supports nesting (Academy, Blog, Docs, FAQ, Pages) builds its item tree by scanning the filesystem and parsing the YAML frontmatter of every single file — on every request, since this package has no build step and no long-running process. The content cache exists to avoid redoing that scan-and-parse work on every request; it is not a general-purpose page cache.

What it caches

Exactly one thing: the **already-built item tree** of a content registry (the result of `AbstractContentRegistry::all()`) — titles, tags, dates, hierarchy, everything derived from frontmatter. It is keyed by the plugin's `path` + `include` + `exclude` options, so two registries pointed at the same content share one cache entry regardless of which plugin instantiated them first.

This benefits every operation that needs to look at more than one item — listings, tag pages, the sidebar, Sitemap, and API's `allContent()` export — since those are exactly the operations that today pay the "read and parse everything" cost on every request, whether or not the specific page requested needed most of that data.

What it does not cache

To avoid false expectations:

- **Rendered output.** HTML, Markdown, PDF and JSON responses are all still generated fresh on every request — Twig compilation, the `markdown`/`twig` filters, PDF generation. Caching the index does not cache a page.
- **HTTP-level caching.** No `Cache-Control` or `ETag` header is set by any content controller. A reverse proxy or CDN in front of the site would still treat every request as uncacheable unless configured to do otherwise, independently of this feature.
- **Search's external calls.** Queries to the search engine and the LLM backend are never cached — they hit the configured `url`/`llm_url` on every call.
- **MCP tool responses.** Each tool call resolves fresh against the (possibly cached) registries; the MCP layer itself adds no caching of its own.
- **Storage attachment downloads.** Files are streamed from disk on every request.

In short: this makes it cheaper to *know what content exists and what it's about*. It does not make any specific page's response faster to render or serve.

Default: a real filesystem cache, no configuration needed

Caching is on by default, with no service to run. The package's own `content-services.yaml` (imported by every site that enables this package) wires a `Symfony\Component\Cache\Adapter\FilesystemAdapter` under namespace `derafu_content`, 60-second TTL, rooted at `%kernel.cache_dir%` — the exact same `var/cache/<env>/` directory the rest of a `derafu/kernel`-based app already uses (compiled container, etc.). Concretely, that means `var/cache/dev/derafu_content/` in a `dev` environment, `var/cache/prod/derafu_content/` in `prod`, and so on — clearing it is `rm -rf var/cache/<env>/derafu_content`, the same mental model as clearing any other cache in the app, nothing extra to remember.

`Derafu\Content\ContentContext` itself defaults to a `FilesystemAdapter` under the system temp directory if no cache pool is injected at all — that fallback only matters if this package is used standalone, outside the shipped `services.yaml` (e.g. directly in PHP, or in this package's own test suite). Any real site importing `content-services.yaml` gets the `var/cache/<env>/derafu_content` location described above, not the temp directory.

Freshness

Each cache entry expires after 60 seconds. There is no manual invalidation and no filesystem-change detection: editing a Markdown file is not reflected instantly, only once its registry's entry expires (up to 60 seconds later) or the process restarts. This is a deliberate trade-off — a short TTL removes the "rescan everything on every request" cost without needing any invalidation logic — but it does mean content changes are not instantaneous the way they were without caching.

On a multi-worker or multi-server deployment, each worker/server builds and caches its own copy independently unless they share the cache backend (e.g. all pointed at the same Redis instance) — see below.

Disabling the cache: `cache.enabled`

Enabled under `derafu.content.config`:

```
parameters:
    derafu.content.config:
        cache:
            enabled: true
```

Option	Type	Default	Description
`enabled`	bool	`true`	Whether the content cache is used at all. When `false`, `ContentContext::cache()` returns `null` and every registry falls back to `AbstractContentRegistry`'s own `null`-cache path: it rescans and reparses on every request, with no pool involved.

This package reads that value as a plain boolean — it does not know about environments or kernels. Sourcing it from an environment variable, with whatever per-environment default a site wants, is entirely the site's own `services.yaml` concern:

```
parameters:
    derafu.content.config:
        cache:
            enabled: '%env(bool:DERAFU_CONTENT_CACHE_ENABLED)%'
```

`DERAFU_CONTENT_CACHE_ENABLED` then comes from wherever the site already resolves per-environment values (a `.env`/`.env.local` cascade, a `when@dev:` config block using `%kernel.environment%`, etc.) — this package never reads `$_SERVER['APP_ENV']` or the kernel for this decision, only the boolean it's handed.

Using a different cache backend

The cache pool is the `derafu_content.cache` service, typed against `Psr\Cache\CacheItemPoolInterface` (PSR-6) where it's injected into `ContentContext`. A site's own `services.yaml` is loaded after the package's, so redefining that same service ID replaces it outright — no need to touch `ContentContextInterface`'s own definition:

```
services:
    derafu_content.cache:
        class: Symfony\Component\Cache\Adapter\RedisAdapter
        arguments:
            $redis: '@Redis'
            $namespace: 'derafu_content'
            $defaultLifetime: 60
```

This is the right move on a multi-worker or multi-server deployment: with the default `FilesystemAdapter`, each worker/server builds and caches its own copy independently; pointing `derafu_content.cache` at a shared backend (Redis, Memcached) instead means they all share one cache instead of duplicating the work. This is a different concern than `cache.enabled` above: swapping the backend still caches, just somewhere shared; `cache.enabled: false` skips caching altogether regardless of which backend is wired.

Content frontmatter

Not applicable — caching operates on already-loaded items, it adds no frontmatter fields of its own.

Last updated on 21/08/2026