Persistence cache

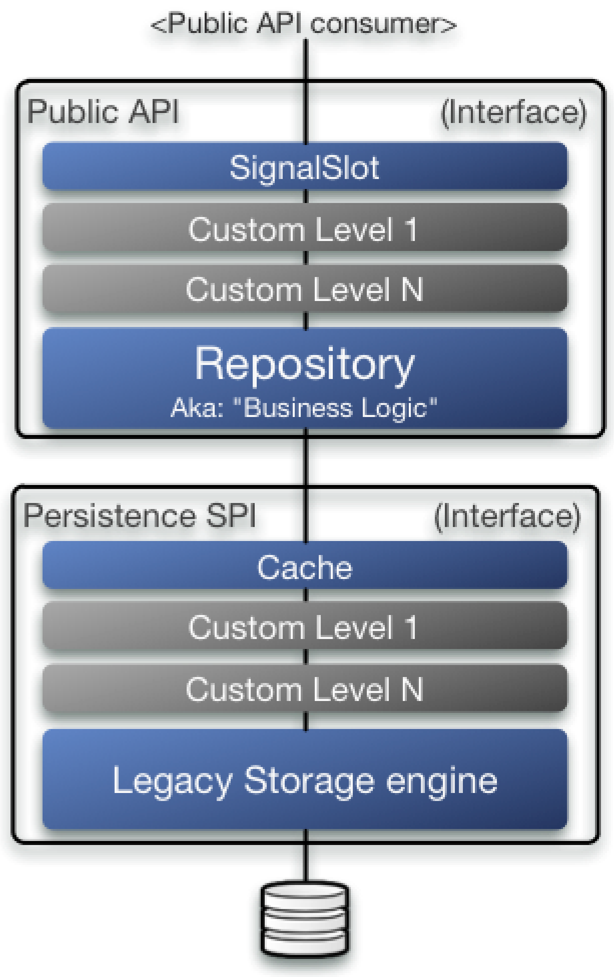
Introduction

This page describes how Persistence cache works, and how to reuse the cache service it uses.

Configuration

For configuring the cache service, look at the Persistence cache configuration page, for re-using the internal cache service see Using Cache service.

Persistent cache



Layers

Persistence cache can best be described as an implementation of `SPI\Persistence` that decorates the main backend implementation *(currently: "Legacy Storage Engine")*.

As shown in the illustration, this is done in the exact same way as the SignalSlot feature is a custom implementation of API\Repository decorating the main Repository. In the case of Persistence Cache, instead of sending events on calls passed on to the decorated implementation, most of the load calls are cached, and calls that perform changes purge the affected caches. This is done using a Cache service which is provided by StashBundle; this Service wraps around the Stash library to provide Symfony logging / debugging functionality, and allows cache handlers *(Memcached, Redis, Filesystem, etc.)* to be configured using Symfony configuration. For how to reuse this Cache service in your own custom code, see below.

Transparent cache

With the persistence cache, just like with the HTTP cache, eZ Platform tries to follow principles of "Transparent caching", this can shortly be described as a cache which is invisible to the end user and to the admin/editors of eZ Platform where content is always returned "fresh". In other words, there should be no need to manually clear the cache like it was frequently the case with eZ Publish 4.x. This is possible thanks to an interface that follows CRUD *(Create Read Update Delete)* operations per domain, and the fact that the number of other operations capable of affecting a certain domain is kept to a minimum.

Entity stored only once

To make the transparent caching principle as effective as possible, entities are, as much as possible, only stored once in cache by their primary id. Lookup by alternative identifiers (`identifier`, `remoteId`, etc.) is only cached with the identifier as cache key and primary `id` as its cache value, and compositions *(list of objects)* usually keep only the array of primary id's as their cache value.

This means a couple of things:

- Memory consumption is kept low

- Cache purging logic is kept simple (For example: `$sectionService->delete( 3 )` clears "section/3" cache entry)
- Lookup by `identifier` and list of objects needs several cache lookups to be able to assemble the result value
- Cache warmup usually takes several page loads to reach full as identifier is first cached, then the object

What is cached?

Persistence cache aims at caching most `SPI\Persistence` calls used in common page loads, including everything needed for permission checking and url alias lookups.

Notes:

- `UrlWildCardHandler` is not currently cached
- Currently in case of transactions this is handled very simply by clearing all cache on rollback, this can be improved in the future if needed.
- Some tree/batch operations will cause clearing all persistence cache, this will be improved in the future when we change to a cache service cable of cache tagging.
- Search is not defined as Persistence and the queries themselves are not planned to be cached. Use Solr which does this for you to improve scale and offload your database.

For further details on which calls are cached or not, and where/how to contribute additional caches, check out the source.