

Persistence cache



Persistence cache was added in 5.1 and enhanced with multi repository (database) support in 5.2.

- [Introduction](#)
- [Persistent cache](#)
 - [Layers](#)
 - [Transparent cache](#)
 - [Entity stored only once](#)
 - [What is cached?](#)
 - [Legacy kernel cache purging](#)
 - [PersistenceCachePurger](#)
- [Reusing Cache service](#)
 - [Get Cache service](#)
 - [Via Dependency injection](#)
 - [Via Symfony2 Container](#)
 - [In legacy via Symfony2 Container](#)
 - [Using the cache service](#)

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.

Persistent cache

Layers

Persistence cache can best be described as a implementation of `SPI\Persistence` that wraps around the main 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 `APIRepository` wraps around the main `Repository`. In the case of Persistence Cache, instead of sending events on calls passed on to the wrapped implementation, most of the load calls are cached, and calls that perform changes purges 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 configuration on cache handlers (`Memcached`, `Apc`, `Filesystem`, ..) to be configured using `Symfony` configuration. For how to reuse this Cache service in your own custom code, see below.

Transparent cache

The persistence cache, just like the `HTTP` cache, 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 Publish where content is always returned "fresh". In other words there should not be a need to manually clear the cache like was frequently the case with eZ Publish 4.x (aka "legacy"). This is possible thanks to an interface that follows `CRUD` (Create Read Update Delete) operations per domain, and 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`, ..) is only cached with the identifier as cache key and primary `id` as it's 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 (Example: `$sectionService->delete(3)` clears "section/3" cache entry)



- Lookup by identifier and list of objects needs several cache lookups to be
- Cache warmup usually takes several page loads to reach full as identifier is first cached, then the object

What is cached?

Persistence cache aimed in its first iteration for caching all `SPI\Persistence` calls used in a normal page load, including everything needed for permission checking and url alias lookups. The following SPI calls are not currently cached:

- `ObjectStateHandler`
- `UrlWildcardHandler`
- `TrashHandler`
- `SearchHandler`

Search queries are currently not cached as it is more difficult to make sure they stay fresh unless all search cache is purged on every modification, or a complicated search cache purge system is implemented that is able to detect which search result to clear. Also, if the Solr implementation is able to perform well, there might not be a need to implement a complex search cache in the first place.

Another thing to mention is transactions, currently this is handled very simple by clearing all cache on rollback, this can be improved in the future if a need for it arises.

For more details on which calls are cached or not, and where to contribute additional caches, check out the [source](#).

Legacy kernel cache purging

Currently with the Dual-kernel eZ Publish has in version 5.x, the "Transparent caching principle" referred to above has one major obstacle. eZ Publish 4.x ("legacy") kernel was not made for such a thing and has a lot of API's that can make changes to the data in the database and hence make the persistence cache "stale" (aka out of date).

A couple of things are in place to try to avoid this from happening / making it less of a problem:

- The Persistence cache has a expiry time configurable in `ezpublish.yml`
- `LegacyBundle` (the bundle that exposes legacy to Symfony and integrates Symfony with legacy) has a `PersistenceCachePurger`

PersistenceCachePurger

`PersistenceCachePurger` is setup by `LegacyBundle` to receive all relevant [cache events](#) triggered by legacy kernel, and clear relevant Persistence cache based on the incoming data.

This means a "Clear all cache" operation done in legacy will also clear all persistence cache, it also means relevant content cache is cleared on publishing, so all code using the api's covered by these events should in effect be cache safe in regards to persistence cache.

So, in case of stale persistence cache, a clear all cache in legacy admin interface is thus still possible, however a manual cache clear is also possible but how to do it depends on which cache handler is currently used.

Reusing Cache service

Using the cache service allows you to use a interface and not have to care about if the system has been configured to place the cache in Memcached, Apc or on File system. And as eZ Publish requires that instances uses a cluster aware cache, you can safely assume your cache is shared across all eZ Publish web servers.



Interface warning

Current implementation uses a caching library called [Stash](#), via [Stash-bundle](#). If this changes, then the Interface of the cache service will most likely change as well.



Cache key warning

When reusing the cache service within your own code, it is very important to not conflict with the cache keys used by others, hence why example of usage starts with a unique "myApp" key for the namespace of your own cache, you must do the same! So never clear cache using the cache service without your key specified, otherwise you'll clear all cache.

**Multi repository info**

New in 5.2 is support for multi repository setups (several databases on same eZ Publish install), see [how to configure this using several Stash cache pools](#) and site access/group setting "cache_pool_name" for selecting the one to use.

Get Cache service

Via Dependency injection

In eZ Publish 5.x Symfony2 stack you can simply define that you require the "cache" service in your configuration like so:

yml configuration

```
myApp.myService:
    class: %myApp.myService.class%
    arguments:
        - @ezpublish.cache_pool
```

The "cache" service is an instance of the following class: `Tedivm\StashBundle\Service\CacheService`

Via Symfony2 Container

Like any other service, it is possible to get the "cache" service via container as well like so:

getting the cache service in php

```
/** @var $cacheService \Tedivm\StashBundle\Service\CacheService */
$cacheService = $container->get( 'ezpublish.cache_pool' );
```

In legacy via Symfony2 Container

When eZ Publish legacy runs via eZ Publish 5.x Symfony2 stack, you will be able to get the service container in the following way:

Getting cache service in legacy

```
// From a legacy module or any PHP code running in legacy context.
$container = ezpKernel::instance()->getServiceContainer();

/** @var $cacheService \Tedivm\StashBundle\Service\CacheService */
$cacheService = $container->get( 'ezpublish.cache_pool' );
```

Using the cache service

Example usage of the cache service:

Actual example from cache use in ezpublish-kernel

```
$cacheItem = $cacheService->getItem( 'myApp', 'object', $id );
if ( $cacheItem->isMiss() )
{
    $myObject = $container->get('my_app.backend_service')->loadObject( $id )
    $cacheItem->set( $myObject );
}
else
{
    $myObject = $cacheItem->get();
}
return $myObject;
```

For more info on usage, take a look at [Stash's documentation](#).