

Zend_Cache_Backend_Static - Pádraic Brady

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

<ac:macro ac:name="unmigrated-inline-wiki-markup"><ac:plain-text-body><![CDATA[

Zend Framework: Zend_Cache_Backend_Static Component Proposal

Proposed Component Name	Zend_Cache_Backend_Static
Developer Notes	http://framework.zend.com/wiki/display/ZFDEV/Zend_Cache_Backend_Static
Proposers	Pádraic Brady
Revision	1.0 - 25 January 2009/28 July 2009 (wiki revision: 8)

Table of Contents

1. Overview
2. References
3. Component Requirements, Constraints, and Acceptance Criteria
4. Dependencies on Other Framework Components
5. Theory of Operation
6. Milestones / Tasks
7. Class Index
8. Use Cases
9. Class Skeletons

1. Overview

Zend_Cache_Backend_Static aims to provide a backend for caching full pages as static files within the public directory, or any public subdirectory. The advantage of this caching strategy is that it allows the HTTP server to directly serve static HTML (and other) files without involving PHP allowing for impressive increases in throughput and responsiveness compared to dynamic pages requiring PHP. The caching mechanism will support tagging in full ensuring that expiry and invalidation management is simplified.

This proposal obviously has one specific deployment target, that of caching on a single server, VPS or shared hosting account where the additional performance is warranted and scaling to multiple servers is a non-issue. That said, there are use cases where static files may be cached to memcache for retrieval by a memcache aware web server (such as a typical frontend nginx setup using Apache as a backend server for PHP work).

To avoid any confusion over the operation of this static cache, it does repeat the functionality of the existing Zend_Cache_Frontend_Page to some extent. The difference is that while Zend_Cache_Frontend_Page requires the invocation of PHP to access the cache, this static backend does not. All static files are stored to either the filesystem or to memory for direct access by web servers without requiring a PHP invocation.

2. References

- [Building A Better Page Cache](#)
- [Controller Based Cache Management](#)
- [Tagging For Static File Caches](#)

Source Code (In Development)

- [git repository](#)

3. Component Requirements, Constraints, and Acceptance Criteria

- **MUST** implement a fully functional and configurable static file cache
- **MUST** provide sufficient features to be easily adapted at a high level by a Cache Manager (see related proposal)
- **SHOULD** offer easy access to page level caching from Controllers via an Action Helper (see class skeletons)

The original third requirement has been deleted after discussion with the Zend_Cache lead developer, and taken forward as a potential improvement point for Zend Framework 2.0. It related to the focus on making Frontends responsible for the validation of IDs used by Backends. Since a Backend cannot define what comprises a valid ID, any Backend using a non-typical ID system must employ a set of workarounds such as hashing and base64 conversions if possible. Reversing this position is quite simple, however it risks a backwards compatibility break for any custom Backend classes written by users. Since any static cache will inevitably be tied to the URL it was generated for, such workarounds must be employed by this proposal.

4. Dependencies on Other Framework Components

Zend Framework Classes

- Zend_Cache_Core
- Zend_Exception
- Zend_Cache_Backend_Interface

Additionally Proposed Classes

- Zend_Cache_Frontend_Capture

5. Theory of Operation

The Zend Framework currently offers a frontend called Zend_Cache_Frontend_Page which caches the output from an entire document into memory or file based caches. This cache is then tested for the next request, and if a hit is detected, the page is loaded from the cache and served. However, while it is many times faster than dynamically generating pages (by a factor of 9-10 on my VPS), it suffers from one problem - it still needs Zend_Cache and PHP. This limits the speed of even the best cached page using Zend_Cache_Frontend_Page since Apache and PHP remains one of the slowest ways of delivering responses.

The fastest way of delivering pages is to save them as ordinary HTML files any HTTP server can serve. The only limit then is your HTTP server's maximum throughput for any given user concurrency level. Apache generally has a consistent performance when optimised, but some popular lightweight HTTP servers like nginx or lighttpd can outpace Apache quite easily in many circumstances, making static files a valuable option when optimising applications on minimal and non-scaled hardware. Web servers such as nginx may also directly access memcached which means entire static files can be stored to memory using a URL based ID for retrieval.

A Static File Cache is generated at the exact same level as Zend_Cache_Frontend_Page. The difference is that it caches pages into files within a public directory of the application with a valid file extension (e.g. .html) and content. On any given request, the HTTP server can skip PHP, and serve this file directly.

This is a lot faster than the current style of Page caching. A simple benchmark on my VPS using a single PHP echo statment vs a static HTML file showed a throughput increase of around 600% (711.20 vs 4,208.52 requests/sec). It also eliminates to an extent the need for using a caching proxy like Squid which has traditionally performed a similar role for applications without static file caching implemented natively, and enhances the benefit of using a low memory/CPU reverse proxy HTTP server (like nginx or lighttpd) to serve static content instead of Apache. This is particularly true on small single servers where Squid is either overkill or simply not available, and even more powerful servers where the reverse proxy HTTP server can overtake Apache with ease.

In the proposed caching system, there will also be a tagging mechanism supported using a backend cache (to maintain the tag data - a cache within a cache) which would make invalidating such statically cached files a lot easier to perform than relying on manual expire instructions (which are near impossible to track in large applications). I would hope to forward a Zend_Cache_Backend_Db proposal for a database backed cache as a separate proposal since a tagging system is more efficiently stored in a more atomic form than a file cache.

An example:

Assuming Static Caching is enabled for the URI:

```
/news/tags/zend-framework/2
```

This would be cached to file as:

```
/news/tags/zend-framework/2.html
```

To keep the public directory free of confusion, 2.html would actually saved to /public/static/2.html (a default convention). Since this obviously does not match the preserved route, the incoming REQUEST_URI is rewritten onto this file location using the following Rewrite Rule additions:

```
RewriteEngine On

RewriteRule ^/(.*)/$ /$1 [QSA]
RewriteRule ^$ static/index.html [QSA]
RewriteRule ^([\^.]+)/$ static/$1.html [QSA]
RewriteRule ^([\^.]*)$ static/$1.html [QSA]

RewriteCond %{REQUEST_FILENAME} -s [OR]
RewriteCond %{REQUEST_FILENAME} -l [OR]
RewriteCond %{REQUEST_FILENAME} -d

RewriteRule ^.*$ - [NC,L]
RewriteRule ^.*$ /index.php [NC,L]
```

A few of these rules handle cases where the URI has a trailing forward-slash, or where the request is for the root URI (i.e. the index file should be served). The rest are the usual recommended rules for Zend Framework applications preceded by rules to rewrite requests initially to the static cache.

Unfortunately, offering this backend requires either changes to Zend_Cache (backend parameter validation is performed by private static methods in Zend_Cache_Core which defeat any attempt at overloading) or workarounds to bypass the validation altogether. This is necessary because the static file cache's ID is its path in the filesystem and not an alphanumeric key. Also the backend class itself could offer additional methods which are otherwise blocked. As noted earlier - these changes will be proposed for Zend Framework 2.0 in the future.

The static cache also may require a new Frontend which can capture output without relying on a pre-set ID (see Zend_Cache_Frontend_Output), as the ID is assigned dynamically based on the value of \$_SERVER['REQUEST_URI'] but this is a relatively minor change. I've elected for the purposes of this proposal to implement this as an entirely new class however there may yet be a means of merging this with the existing Frontend for capturing output. Both methods of capturing output are very similar - but operate at slightly different levels which may make merging difficult or messy.

It is anticipated that static file caching will be of use in small to medium applications on limited hardware in a minimally scaled system. Obviously, the static cache method is not very scalable since it's normally bound to a filesystem. However using memcached is an alternative depending on whether web servers are memcache aware.

6. Milestones / Tasks

- Milestone 1: [DONE] Complete prototype/minimal iteration version
- Milestone 2: Complete final feature list incorporating feedback
- Milestone 3: Verify operation using Unit Tests
- Milestone 4: Documentation

7. Class Index

- Zend_Cache_Frontend_Capture
- Zend_Cache_Backend_Static

8. Use Cases

Note: All use cases have the same problem - the cache ID is actually the REQUEST URI of the current request and this can easily contain characters forbidden by Zend_Cache_Core's private static validation methods. I had omitted any workaround on the assumption this restriction can be lifted (which it won't - but a workaround is in development so it won't matter).

UC-01: Starting/Ending Cache

```

<? $backendOptions = array(
    'public_dir' => '/var/www/example.com/public',
    'cache_dir' => 'static',
    'file_extension' => 'html',
    'index_file' => 'index.html',
    'file_locking' => true,
    'cache_file_umask' => 0600,
    'debug_header' => true
);
$frontend = new Zend_Cache_Frontend_Capture;
$backend = new Zend_Cache_Backend_Static($backendOptions);
$staticCache = Zend_Cache::factory($frontend, $backend);
$staticCache->start(); ?>
<html>
<title>Caching!</title>
<body>
<p>I am text that rarely changes...</p>
</body>
</html>
<? $staticCache->end(); ?>

```

UC-02: Deleting Caches based on \$_SERVER['REQUEST_URI']

```

$backendOptions = array(
    'public_dir' => '/var/www/example.com/public',
    'cache_dir' => 'static',
    'file_extension' => 'html',
    'index_file' => 'index.html',
    'file_locking' => true,
    'cache_file_umask' => 0600,
    'debug_header' => true
);
$frontend = new Zend_Cache_Frontend_Capture;
$backend = new Zend_Cache_Backend_Static($backendOptions);
$staticCache = Zend_Cache::factory($frontend, $backend);

// this also demonstrates why hand tracking is bloody hard!
// Using REQUEST_URI ignores that any controller/action can match many URIs
$staticCache->remove('/');
$staticCache->remove('/index');
$staticCache->remove('/index/index');
$staticCache->remove('/default/index/index');

```

UC-03: Assigning Tags

```

<? $backendOptions = array(
    'public_dir' => '/var/www/example.com/public',
    'cache_dir' => 'static',
    'file_extension' => 'html',
    'index_file' => 'index.html',
    'file_locking' => true,
    'cache_file_umask' => 0600,
    'debug_header' => true
);
$frontend = new Zend_Cache_Frontend_Capture;
$backend = new Zend_Cache_Backend_Static($backendOptions);
// add inner cache to retain Tag=>URI maps
// A database cache would be better for large sites
$innerCache = Zend_Cache::factory(
    'Core', 'File',
    array('lifetime'=>null,'automatic_serialization'=>true),
    array('cache_dir'=>'/cache')
);
$backend->setInnerCache($innerCache);
$staticCache = Zend_Cache::factory($frontend, $backend);
$staticCache->start(); ?>
<html>
<title>Caching!</title>
<body>
<p>I am text that rarely changes...</p>
</body>
</html>
<? $staticCache->end(array('tag1','tag2')); ?>

```

UC-04: Deleting static files by tag

```

$backendOptions = array(
    'public_dir' => '/var/www/example.com/public',
    'cache_dir' => 'static',
    'file_extension' => 'html',
    'index_file' => 'index.html',
    'file_locking' => true,
    'cache_file_umask' => 0600,
    'debug_header' => true
);
$frontend = new Zend_Cache_Frontend_Capture;
$backend = new Zend_Cache_Backend_Static($backendOptions);
$staticCache = Zend_Cache::factory($frontend, $backend);

// this also demonstrates why hand tracking is bloody hard!
// Using REQUEST_URI ignores that any controller/action can match many URIs
$staticCache->clean(Zend_Cache::CLEANING_MODE_MATCHING_TAG,'tag1');

```

9. Class Skeletons

```

<?php

/**
 * @see Zend_Cache_Backend_Interface
 */
require_once 'Zend/Cache/Backend/Interface.php';

/**
 * @see Zend_Cache_Backend
 */
require_once 'Zend/Cache/Backend.php';

```

```

class Zend_Cache_Backend_Static extends Zend_Cache_Backend implements Zend_Cache_Backend_Interface
{
    const INNER_CACHE_NAME = 'zend_cache_backend_static_tagcache';

    protected $_options = array(
        'public_dir' => null,
        'sub_dir' => 'html',
        'file_extension' => '.html',
        'index_filename' => 'index',
        'file_locking' => true,
        'cache_file_umask' => 0600,
        'debug_header' => false,
        'tag_cache' => null
    );

    protected $_tagCache = null;

    protected $_tagged = null;

    /**
     * Interceptor child method to handle the case where an Inner
     * Cache object is being set since it's not supported by the
     * standard backend interface
     *
     * @param string $name
     * @param mixed $value
     * @return void
     */
    public function setOption($name, $value);

    /**
     * Test if a cache is available for the given id and (if yes) return it (false else)
     *
     * Note : return value is always "string" (unserialization is done by the core not by the backend)
     *
     * @param string $id Cache id
     * @param boolean $doNotTestCacheValidity If set to true, the cache validity won't be tested
     * @return string|false cached datas
     */
    public function load($id, $doNotTestCacheValidity = false);

    /**
     * Test if a cache is available or not (for the given id)
     *
     * @param string $id cache id
     * @return mixed|false (a cache is not available) or "last modified" timestamp (int) of the
     available cache record
     */
    public function test($id);

    /**
     * Save some string datas into a cache record
     *
     * Note : $data is always "string" (serialization is done by the
     * core not by the backend)
     *
     * @param string $data Datas to cache
     * @param string $id Cache id
     * @param array $tags Array of strings, the cache record will be tagged by each string entry
     * @param int $specificLifetime If != false, set a specific lifetime for this cache record (null =>
     infinite lifetime)
     * @return boolean true if no problem
     */
    public function save($data, $id, $tags = array(), $specificLifetime = false);

    /**
     * Remove a cache record
     *

```

```

* @param string $id Cache id
* @return boolean True if no problem
*/
public function remove($id);

/**
 * Remove a cache record recursively for the given directory matching a
 * REQUEST_URI based relative path (deletes the actual file matching this
 * in addition to the matching directory)
 *
 * @param string $id Cache id
 * @return boolean True if no problem
 */
public function removeRecursively($id);

/**
 * Clean some cache records
 *
 * Available modes are :
 * Zend_Cache::CLEANING_MODE_ALL (default) => remove all cache entries ($tags is not used)
 * Zend_Cache::CLEANING_MODE_OLD => remove too old cache entries ($tags is not used)
 * Zend_Cache::CLEANING_MODE_MATCHING_TAG => remove cache entries matching all given tags
 * ($tags can be an array of strings or a single string)
 * Zend_Cache::CLEANING_MODE_NOT_MATCHING_TAG => remove cache entries not {matching one of the
given tags}
 * ($tags can be an array of strings or a single string)
 * Zend_Cache::CLEANING_MODE_MATCHING_ANY_TAG => remove cache entries matching any given tags
 * ($tags can be an array of strings or a single string)
 *
 * @param string $mode Clean mode
 * @param array $tags Array of tags
 * @return boolean true if no problem
 */
public function clean($mode = Zend_Cache::CLEANING_MODE_ALL, $tags = array());

/**
 * Set an Inner Cache, used here primarily to store Tags associated
 * with caches created by this backend. Note: If Tags are lost, the cache
 * should be completely cleaned as the mapping of tags to caches will
 * have been irrevocably lost.
 *
 * @param Zend_Cache_Core
 * @return void
 */
public function setInnerCache(Zend_Cache_Core $cache);

/**
 * Get the Inner Cache if set
 *
 * @return Zend_Cache_Core
 */
public function getInnerCache();

protected function _verifyPath($path);

protected function _detectId();

/**
 * Validate a cache id or a tag (security, reliable filenames, reserved prefixes...)
 *
 * Throw an exception if a problem is found
 *
 * @param string $string Cache id or tag
 * @throws Zend_Cache_Exception
 * @return void
 */

```

```

        protected static function _validateIdOrTag($string);
    }

```

Output capturing frontend:

```

<?php
/**
 * Zend Framework
 *
 * LICENSE
 *
 * This source file is subject to the new BSD license that is bundled
 * with this package in the file LICENSE.txt.
 * It is also available through the world-wide-web at this URL:
 * http://framework.zend.com/license/new-bsd
 * If you did not receive a copy of the license and are unable to
 * obtain it through the world-wide-web, please send an email
 * to license@zend.com so we can send you a copy immediately.
 *
 * @category Zend
 * @package Zend_Cache
 * @subpackage Zend_Cache_Frontend
 * @license http://framework.zend.com/license/new-bsd New BSD License
 */

/**
 * @see Zend_Cache_Core
 */
require_once 'Zend/Cache/Core.php';

/**
 * @package Zend_Cache
 * @subpackage Zend_Cache_Frontend
 * @license http://framework.zend.com/license/new-bsd New BSD License
 */
class Zend_Cache_Frontend_Capture extends Zend_Cache_Core
{
    protected $_idStack = array();

    protected $_tags = array();

    /**
     * Start the cache
     *
     * @param string $id Cache id
     * @return mixed True if the cache is hit (false else) with $echoData=true (default) ; string else
     (datas)
     */
    public function start($id, $tags);

    /**
     * callback for output buffering
     * (shouldn't really be called manually)
     *
     * @param string $data Buffered output
     * @return string Data to send to browser
     */
    public function _flush($data);
}

```

With `Zend_Cache_Manager`, implementation of an Action Helper to integrate page level caching.

```

<?php

/**
 * @see Zend_Controller_Action_Helper_Abstract
 */
require_once 'Zend/Controller/Action/Helper/Abstract.php';

/**
 * @see Zend_Controller_Action_Exception
 */
require_once 'Zend/Controller/Action/Exception.php';

/**
 * @see Zend_Cache_Manager
 */
require_once 'Zend/Cache/Manager.php';

class Zend_Controller_Action_Helper_Cache extends Zend_Controller_Action_Helper_Abstract
{

    /**
     * Local Cache Manager object used by Helper
     *
     * @var Zend_Cache_Manager
     */
    protected $_manager = null;

    /**
     * Indexed map of Actions to attempt Page caching on by Controller
     *
     * @var array
     */
    protected $_caching = array();

    /**
     * Indexed map of Tags by Controller and Action
     *
     * @var array
     */
    protected $_tags = array();

    /**
     * Track output buffering condition
     */
    protected $_obStarted = false;

    /**
     * Tell the helper which actions are cacheable and under which
     * tags (if applicable) they should be recorded with
     *
     * @param array $actions
     * @param array $tags
     * @return void
     */
    public function direct(array $actions, array $tags = array());

    /**
     * Remove a specific page cache static file based on its
     * relative URL from the application's public directory.
     * The file extension is not required here; usually matches
     * the original REQUEST_URI that was cached.
     *
     * @param string $relativeUrl
     * @param bool $recursive
     * @return mixed
     */
    public function removePage($relativeUrl, $recursive = false);

    /**
     * Remove a specific page cache static file based on its

```

```

* relative URL from the application's public directory.
* The file extension is not required here; usually matches
* the original REQUEST_URI that was cached.
*
* @param array $tags
* @return mixed
*/
public function removePagesTagged(array $tags);

/**
* Commence page caching for any cacheable actions
*
* @return void
*/
public function preDispatch();

/**
* Set an instance of the Cache Manager for this helper
*
* @param Zend_Cache_Manager $manager
* @return void
*/
public function setManager(Zend_Cache_Manager $manager);

/**
* Get the Cache Manager instance or instantiate the object if not exists
*
* @return Zend_Cache_Manager
*/
public function getManager();

/**
* Return a list of actions for the current Controller marked for caching
*
* @return array
*/
public function getCacheableActions();

/**
* Return a list of tags set for all cacheable actions
*
* @return array
*/
public function getCacheableTags();

/**
* Proxy non-matched methods back to Zend_Cache_Manager where appropriate
*
* @param string $method
* @param array $args
* @return mixed
*/

```

```
public function __call($method, $args);  
}
```

```
]]</ac:plain-text-body></ac:macro>  
]]</ac:plain-text-body></ac:macro>
```