


Zend Framework Page Caching: Part 1: Building A Better Page Cache

There is so much knowledge about the Zend Framework, but so few outlets for it, that the most obvious golden nuggets sit around in peoples' minds forever untapped. In writing the Performance Optimisation chapter draft for Zend Framework: Surviving The Deep End, I recognised that it was a huge appendix - and nowhere near complete. At around 6500 words, I had to reign myself in before it doubled in size. Well, actually it DID double in size - but I kept cutting out the less immediately relevant bits for another day. Here's one of those bits. Why I keep calling it a "bit", I don't know...

In this article I explore one particular topic to be added to that appendix (a mini-book by itself if this rate keeps up) - the concept of Page Caching. Briefly mentioned in the book's appendix this is one of the most powerful and overlooked optimisations utilising caching. There are still blogs and forums that seem immune to it.

Before I'm busted, these are not new ideas. It's likely they are familiar to many programmers, so the article is more of an effort to demonstrate how they can be implemented within the confines of the Zend Framework in a more conveniently managed way, along with some facets pulled from other frameworks I've always liked.

What Is Page Caching?

Page Caching is the process of caching entire generated HTML documents for a period of time so that the expensive task of dynamically generating them is avoided for that period. Done correctly, it should completely bypass the Zend Framework MVC stack. This can net you incredible results! When developing the mini-application (it's really tiny) for presenting [Zend Framework: Surviving The Deep End](#) online, implementing Page Caching via Zend_Cache resulted in a 10 fold increase in requests per second from my Slicehost VPS. This is the kind of optimisation you should be fantasising about 

Page Caching is great, but it's only applicable if the output from the application URL the cache is generated from experiences detectable or predictable periodic updates. What this means, is that once we cache a page we only want to invalidate and clear that cache when the data driving the dynamic portions of the cached page change. If change is readily detectable or follows a regular periodic update pattern, this can be accomplished quite handily. However, if changes have a high frequency, or depend on unpredictable factors, or requires authentication, then page caching often loses its benefits.

One example of a page you can't cache to a single location, is one where user specific details are displayed. Since this invariable changes depending on who's requesting the same URL, caching it as a single page is unrealistic - unless you are absolutely obsessed and cache on a per user basis! Another example is authentication where page access needs to be restricted - then the cache can only be used after authentication which is less effective since you still need to tap into the application.

These examples often prevent whole page caching, however this does not mean you can't use partial caching - creating a system where the page is aggregated from both dynamically rendered HTML, and cached HTML. However, partial caching does necessitate hitting the Zend Framework MVC stack which is perceptibly slower than page caching which should bypass the MVC stack completely.

Simple Page Caching Example

The simplest form of page caching would utilise Zend_Cache and the Zend_Cache_Frontend_Page frontend. In this example, I've elected to cache pages into memory using APC. If you can spare the RAM, caching in memory is a lot better than caching to files using this method. In fact any caching to memory is likely better than using files given the speed of memory access compared to file operations. You can switch to file or other caching if you prefer.

Since the goal is to bypass the Zend Framework completely to make the page request as fast as possible, the page cache is implemented in a Bootstrap class in a run() method just before the Bootstrap commences an MVC request cycle. When a cache "hit" is detected, this will serve the cached HTML from the selected cache backend and prevent the Front Controller from being run. If no "hit" is detected, the application is called upon as usual and its output recorded and cached for future requests. To control the cache's lifetime, you can assign a Time To Live (TTL) value or you can manually purge the cache when any Model dependency is altered.

In tests on a VPS, I managed to get from 35 requests per second to over 330 requests per second for pages which were cached to APC using this method. The main bottleneck on my VPS is the CPU so everything is cached to memory since there's nearly always spare RAM sitting around idle.

Here's an example of a page cache implemented in a Bootstrap class.

```
class ZFExt_Bootstrap
{
    // ...
    public function run()
    {
        $this->setupEnvironment();
        // Implement Page Caching at Bootstrap level before any
        // MVC operations so these operations can be completely
        // avoided when a valid cache exists
        $this->usePageCache();
        // If a valid cache exists, execution exits!
        $this->prepare();
        $response = self::$frontController->dispatch();
        $this->sendResponse($response);
    }
    public function usePageCache()
    {
        $frontendOptions = array(
            // cache for 1 hour
            'lifetime' => 3600,
            // Disable caching by default for all URLs
            'default_options' => array(
                'cache' => false
            ),
            // Only cache URLs for Index and News controllers
            // matching the following patterns
            'regexps' => array(
                '^/$' => array('cache' => true),
                '^/index/' => array('cache' => true),
            )
        );
    }
}
```

```

    '^/news/' => array('cache' => true),
    '^/blog/tags/' => array('cache' => true)
)
);
// Note: APC backend has no options!
$cache = Zend_Cache::factory(
    'Page',
    'Apc',
    $frontendOptions
);
// Serve cached page (if it exists) and exit
// otherwise cache all output after this point
// assuming caching is enabled for the current URL
$cache->start();
}
}

```

If you want to eke out a little more speed, you can skip the Bootstrap and just implement the cache in your index.php file so any overhead from using the Bootstrap class is avoided.

Why PHP Dependent Page Caching Sometimes Sucks

In testing on one of my Virtual Private Servers with Slicehost, simple page caching to memory using Zend_Cache nets me a 9-10 fold increase in requests per second. Nearly all of that benefit lies in avoiding the Zend Framework MVC stack.

If your application has scaled beyond a handful of servers, this is probably the end of the road for you, but if you are still hosted on a single server (or a small scaled solution where files are maintained on one server) you can push the boundaries of page caching even further. The rest of this article concerns this scenario.

A lot of the reason why the ZF's default page caching sucks at times is that it requires PHP. Some people might have warm and fuzzy feelings but in the game of getting the most out of any server, Apache is a loose cannon. It uses a lot of memory and CPU and that gets worse when PHP is called upon.

The next optimisation level therefore is avoiding PHP and/or Apache completely. If we can remove PHP from the equation, our Apache processes will use less memory. If we can remove even Apache from the equation we save even more. By relying on PHP to retrieve cached pages, neither of these is possible without some drastic change.

On a related note, one growing strategy to avoid Apache is to offload requests for static resources (images, css, etc.) onto a more efficient HTTP server which uses less memory and CPU than Apache. Only requests needing PHP would then go through Apache. Apache is already fast, but it's not the fastest! Two common choices for alternative HTTP servers are lighttpd and nginx.

This strategy is often called a "reverse proxy". It involves reconfiguring Apache to operate as a backend HTTP server (by making it listen to a port other than 80) which exists to service requests that need PHP. All other requests are serviced by a frontend HTTP server listening on port 80, like nginx, which can serve non-dynamic static resources like images, javascript, css and so on, at incredible rates using minimal memory and CPU. Since nginx is the frontend HTTP server, whenever it detects a dynamic request requiring

the use of PHP it will proxy that request to Apache which is listening on an alternative port. All of this keeps Apache usage to a minimum. While this may appear to be a complex idea, in practice it's ludicrously simple to implement. I highly suggest locating a tutorial for setting up an nginx reverse proxy and giving it a trial run - it's well worth the effort.

Back on track, avoiding PHP would mean that we can't access the Zend_Cache version of page caches, since any page cache retrieval would need PHP. So instead, we'll up the ante and instead cache pages as static HTML files which Apache (or nginx in a reverse proxy setup) can serve directly without invoking PHP.


Note: This is not very scalable since it requires file manipulation. If you are scaling beyond one or two servers, the current page caching may have to do. However for single servers you'll see the advantages.


How much of a difference will static page caching have compared to in-memory page caching using APC and Zend_Cache? I did a few quick benchmarks using my nginx reverse proxy and came up with the following from a moderate 10,000 request ApacheBench run with 100 concurrent users repeated three times and averaged. The tests use a simple PHP file which echo's "Hello World" and exits, and a static text file containing the same.

Apache (serving PHP file) : 711.20 requests/sec

Apache (serving static file) : 2408.55 requests/sec [+338% vs PHP]

Nginx (serving static file) : 4208.52 requests/sec [+591% vs PHP, +175% vs Apache]

In other words, screw Apache 

. Fluctuations between hardware, configurations and ApacheBench aside, nginx outperforms Apache by quite a margin in a reverse proxy setup where static files are favoured over PHP. If anyone benches differently comment your stats - I'm throwing these out from a system optimised for nginx so they aren't gospel, and the margin is likely inflated a bit as a result 

If you translate this to a Zend Framework application applying page caching as is (using PHP on every request), well, you get the picture. You CAN do better! How?!

Static HTML Caching

Static File Caching is a solution whereby the output from applications is cached as a static HTML/XML/RSS/JSON file. The first request hits the application, but the next will serve up the static file and never even take a peep at PHP. This does, obviously, have a downside! Yes, it's zipping along at 4000+ requests per second but if our application is completely bypassed, how are we going to manage the static file cache? Where will it be cleared, invalidated, and replaced when the data it's generated from is updated? Set that thought aside for the moment and we'll solve one problem at a time.

Static file caching is not implemented in the Zend Framework so we need to do a little legwork and create a custom backend to support it.

The goal of our custom backend is to create a static HTML file containing the application output we want to cache. It gets a little more complex though, the static file needs to be stored in such a way that it's location mirrors the URL being requested. We also need to ensure that adding static file caching (essentially .html, .xml, etc. files) does not force us to change the routing we've configured if possible. Unfortunately that's not always avoidable, and a prime problem here are pages requested with parameters (GET query strings, or

POST data). For this reason, static file caching encourages adding query parameters into the URL's path section, even for POSTs where it makes some sense.

Here we start meeting some of the limitations of static HTML caching - it works well for straight forward URLs, but throw in query strings or POST data and the static caching will need to be replaced with the current Zend_Cache page caching which relies on PHP for every request.

However, I'm going to keep this example simple and use a URL with parameters as would be typical of any Zend Framework application. Consider the following blog URL:

```
/blog/tags/zend-framework
```

We'll assume this route points to a Controller which triggers the display of some blog posts which have been tagged "zend-framework". Keeping with simplicity, we're ignoring pagination (hint: make the page number an extra URL parameter and it works). To enable static HTML caching (again, XML and others need their own magic so we'll stay with HTML for now), we'll need to do a little wizardry so Apache will map this URL to:

```
/blog/tags/html/zend-framework.html
```

At first anyway - once Apache notices there is no such HTML file it should go back to the original URL and map it onto index.php as usual. To accomplish this requires a small change to our Rewrite Rules. Based on the recommended rewrite rules for Zend Framework, here's the new version:

RewriteEngine On

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

In case you were wondering, the extra reference to a "html" directory is the location relative to /public where we will cache static HTML. Keeping it separated from /public will enable us to clear the entire cache in one go if the need arises, and generally just keeps the /public directory a bit tidier. With the HTML file extension mapping, we can now take advantage of the recommended Apache rewrite rule to serve the static content at this URL assuming the file exists, and is not a zero-size file. If the file does not exist, the URL will be rewritten onto index.php - at which point the application will be triggered to dynamically generate the page, and then cache it to the relevant static HTML location.

Let's jump the gun and take a look at an example of a simple Static HTML caching backend for Zend Framework. Fair warning in advance, since Zend_Cache_Backend method parameters are actually validated by private static methods in the frontend (confused?), manipulating the validity of a cache ID is next to impossible. We'll have to get creative for this one...so bear with me.

The answer, of sorts (for now), is an Adapter class. Using an Adapter, we can use a preferred API and let the Adapter translate our need for URL based cache ids to the underlying cache frontend's requirement for alphanumeric+underscore ids. Since it's only a few methods, the Adapter is pretty easy going and it's far less

code than attempting to extend the existing Zend_Cache_Core and copy/pasting source code everywhere. As an Adapter, we will encapsulate the entire cache object we create within it so all access passes through the Adapter.

```
<?php
class ZFExt_Cache_Backend_Static_Adapter
{
    protected $_cache = null;
    public function __construct(Zend_Cache_Core $cache)
    {
        $this->_cache = $cache;
    }
    public function load($id)
    {
        $id = $this->_encodeId($id);
        $this->__call('load', array($id));
    }
    public function test($id)
    {
        $id = $this->_encodeId($id);
        $this->__call('test', array($id));
    }
    public function save($data, $id, $tags = array(), $specificLifetime = false)
    {
        $id = $this->_encodeId($id);
        $this->__call('save', array($data, $id, $tags, $specificLifetime));
    }
    public function remove($id)
    {
        $id = $this->_encodeId($id);
        $this->__call('remove', array($id));
    }
    public function __call($method, array $args)
    {
        return call_user_func_array(array($this->_cache, $method), $args);
    }
    public function removeRecursive($id) {
        $this->_cache->getBackend()->removeRecursive($id);
    }
    protected function _encodeId($id) {
        return bin2hex($id); // encode path to alphanumeric hexadecimal
    }
}
}
```

I know this Adapter is a bit unsettling and mysterious, but it's main purpose is to workaround the Zend_Cache_Core private static validators which are otherwise incapable of being overridden. The Adapter works by encoding the IDs of the hosted cache (since this is a static HTML cache, the ID is the static file's path relative to /public/html) so it doesn't trigger an exception from Zend_Cache_Core. The encoding is a simple mechanic - convert the path to a hexadecimal string. We'll decode it from our custom Static HTML backend where it's needed. It's also deliberately narrow scoped - rather than directly calling the backend from it, we're maintaining a route through the frontend since it minimises the amount of tampering we're doing and keeps a lid on any unintended behaviour changes. There is one additional method added since it's not

supported by the frontend API, and this method does directly reach the backend.

Onto the main attraction! Here's a quick stab at a ZFExt_Cache_Backend_Static class which will perform the static HTML generation when pages are sent to it for caching by Zend_Cache_Frontend_Page.

```
<?php
```

```
class ZFExt_Cache_Backend_Static extends Zend_Cache_Backend implements  
Zend_Cache_Backend_Interface
```

```
{  
    // Available options  
    protected $_options = array(  
        'public_dir' => null,  
        'file_extension' => '.html',  
        'index_filename' => 'index',  
        'file_locking' => true,  
        'cache_file_umask' => 0600,  
        'debug_header' => false  
    );  
    // Test if a cache is available for the given id and (if yes) return it  
    // (false else)  
    // $id should be the REQUEST_URI whose static file is to be deleted  
    public function load($id, $doNotTestCacheValidity = false)  
    {  
        $id = $this->_decodeId($id);  
        if ($doNotTestCacheValidity) {  
            $this->_log("ZFExt_Cache_Backend_Static::load() : $doNotTestCacheValidity=true is  
unsupported by the Static backend");  
        }  
        $fileName = basename($id);  
        if (empty($fileName)) {  
            $fileName = $this->_options['index_filename'];  
        }  
        $pathName = $this->_options['public_dir'] . dirname($id);  
        $file = $pathName . '/' . $fileName . $this->_options['file_extension'];  
        if (file_exists($file)) {  
            return file_get_contents($file);  
        }  
        return false;  
    }  
    // Test if a cache is available or not  
    // $id should be the REQUEST_URI whose static file is to be deleted  
    public function test($id)  
    {  
        $id = $this->_decodeId($id);  
        $fileName = basename($id);  
        if (empty($fileName)) {  
            $fileName = $this->_options['index_filename'];  
        }  
        $pathName = $this->_options['public_dir'] . dirname($id);  
        $file = $pathName . '/' . $fileName . $this->_options['file_extension'];  
        if (file_exists($file)) {  
            return true;  
        }  
    }  
}
```

```

    return false;
}
// Save content to a static content file in /public directory
public function save($data, $id, $tags = array(), $specificLifetime = false)
{
    clearstatcache();
    $fileName = basename($_SERVER['REQUEST_URI']);
    if (empty($fileName)) {
        $fileName = $this->_options['index_filename'];
    }
    $pathName = $this->_options['public_dir'] . dirname($_SERVER['REQUEST_URI']);
    if (!file_exists($pathName)) {
        mkdir($pathName, $this->_options['cache_file_umask'], true);
    }
    $dataUnserialized = unserialize($data);
    if ($this->_options['debug_header']) {
        $dataUnserialized['data'] =
            'DEBUG HEADER : This is a cached page !' . $dataUnserialized['data'];
    }
    $file = $pathName . '/' . $fileName . $this->_options['file_extension'];
    if ($this->_options['file_locking']) {
        $result = file_put_contents($file, $dataUnserialized['data'], LOCK_EX);
    } else {
        $result = file_put_contents($file, $dataUnserialized['data']);
    }
    @chmod($file, $this->_options['cache_file_umask']);
    if (count($tags) > ) {
        $this->_log(self::TAGS_UNSUPPORTED_BY_SAVE_OF_STATIC_BACKEND);
    }
    return (bool) $result;
}
// Remove a cache record
// $id should be the REQUEST_URI whose static file is to be deleted
public function remove($id)
{
    $id = $this->_decodeId($id);
    $fileName = basename($id);
    if (empty($fileName)) {
        $fileName = $this->_options['index_filename'];
    }
    $pathName = $this->_options['public_dir'] . dirname($id);
    $file = $pathName . '/' . $fileName . $this->_options['file_extension'];
    return unlink($file);
}
// Remove a cache record recursively (i.e. the file AND matching directory)
// it ain't perfect - there may be no file matching the directory name
// (but you get the point I'm sure!)
// $id should be the REQUEST_URI whose static file & dir tree is to be deleted
public function removeRecursively($id)
{
    $id = $this->_decodeId($id);
    $fileName = basename($id);
    if (empty($fileName)) {

```

```

    $fileName = $this->_options['index_filename'];
}
$pathName = $this->_options['public_dir'] . dirname($id);
$file = $pathName . '/' . $fileName . $this->_options['file_extension'];
$directory = $pathName . '/' . $fileName;
if (file_exists($directory)) {
    if (!is_writable($directory)) {
        return false;
    }
    foreach (new DirectoryIterator($directory) as $file) {
        if (true === $file->isFile()) {
            if (false === unlink($file->getPathName())) {
                return false;
            }
        }
    }
    rmdir($directory);
}
if (file_exists($file)) {
    if (!is_writable($file)) {
        return false;
    }
    return unlink($file);
}
}
}
// Clean some cache records
// Not implemented here since we would need a backend tagging system given
// that static files themselves cannot be tagged in the filename. The noon-tag
// related functionality could be implemented in the future if required.
public function clean($mode = Zend_Cache::CLEANING_MODE_ALL, $tags = array())
{
    switch ($mode) {
        case Zend_Cache::CLEANING_MODE_ALL:
        case Zend_Cache::CLEANING_MODE_OLD:
        case Zend_Cache::CLEANING_MODE_MATCHING_TAG:
        case Zend_Cache::CLEANING_MODE_NOT_MATCHING_TAG:
        case Zend_Cache::CLEANING_MODE_MATCHING_ANY_TAG:
            $this->_log("ZFExt_Cache_Backend_Static : Cleaning Modes Currently Unsupported By
This Backend");
            break;
        default:
            Zend_Cache::throwException("Invalid mode for clean() method");
            break;
    }
}
}
// "Danger, Will Robinson!"
// Ensure path is not below the configured public_dir
// Encoded by ZFExt_Cache_Backend_Static_Adapter
protected function _decodeId($id)
{
    $path = pack("H*", $id);
    if (!$this->_verifyPath($path)) {
        Zend_Cache::throwException("Invalid cache id: does not match expected public_dir path");
    }
}

```

```

    }
    return $path;
}
protected function _verifyPath($path)
{
    $path = realpath($path);
    $base = realpath($this->_options['public_dir']);
    return strcmp($path, $base, strlen($base)) !== 0;
}
}
}

```

To make the most of things, ZFExt_Cache_Backend_Static should be used with the Zend_Cache_Frontend_Page frontend since it works well when capturing full pages to cache. It does extra stuff, and maybe the Output frontend would work too, but this mix works fine for our example.

Let's replace the original page caching in our earlier example with this static file caching in the Bootstrap.

```

<?php
class ZFExt_Bootstrap
{
    // ...
    public function run()
    {
        $this->setupEnvironment();
        // Implement Page Caching at Bootstrap level before any
        // MVC operations so these operations can be completely
        // avoided when a valid cache exists
        $this->usePageCache();
        // If a valid cache exists, execution exits!
        $this->prepare();
        $response = self::$frontController->dispatch();
        $this->sendResponse($response);
    }
    public function usePageCache()
    {
        $frontendOptions = array(
            'default_options' => array(
                'cache' => false
            ),
            // Only cache URLs for Index and News controllers
            // matching the following patterns
            'regexprs' => array(
                '^/$' => array('cache' => true),
                '^/index/' => array('cache' => true),
                '^/news/' => array('cache' => true),
                '^/blog/tags/' => array('cache' => true)
            )
        );
        $backendOptions = array(
            'debug_header' => true,
            // cache to a sub-directory of /public for separation
            'public_dir' => self::$root . '/public/html'
        );
    }
}

```


```

);
$backend = new ZFExt_Cache_Backend_Static($backendOptions);
// use our Adapter to deal with the Core private validation
$cache = new ZFExt_Cache_Backend_Static_Adapter(
    Zend_Cache::factory('Page', $backend, $frontendOptions)
);
// cache all output after this point to static HTML
// assuming caching is enabled for the current URL
$cache->start();
}
}

```

If you have an application handy, throw this in and see what happens (edit the regexps to match a route or two). If it breaks, let me know, but in most cases the above code will spawn static HTML files for URLs matching the regular expressions you set for the Zend_Cache_Frontend_Page frontend. Subsequent hits should serve the static HTML file which, per the options we're using, will contain a debug message noting it's a cached file.

Now that we've seen the caching work, how do we get rid of the damn HTML files when they are out of date?

Tune in tomorrow to find out! 

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 22:30

Hmm, this beats my fetch-page-with-curl-and-serve-from-lighttpd way of caching and proxying dynamic webpages, maybe it's me just being lazy.

But my ZF-veins have again filled with adrenaline so I'm going to check this out and see with my own eyes what an improvement it can be.

Great job ! Anonymous on Jan 18 2009, 11:09

Haha stumbled on your website by supreme accident, remembered the name from Solar Empire days gone by. If you google "eve online review" your website comes up like 5th or 6th.

Seems a bit funny that a few of us ol' SE dorks play eve online, huh! Drop me a line in game sometime if you still play, we can catch up or whatever the equivalent is for online acquaintances. Toon: Weazy Z Anonymous on Jan 18 2009, 17:37

Not worth the time, such a cache for better performance. Even if you see it as kind of security layer.


We have a Zend Framework app running. It is a high load scenario and we have that kind of cache. But as a security fallback.

2 Things come in mind when i read that posting.

1. Use a reverse proxy cache. Configure Squid or another product that does NOT use php for any request. So you reduce (for example) the number of filehandles used to serve a page in a cached scenario dramatically. Besides you are not running a request that needs a apache worker. The number of pages/sec served for quasi-static content raises dramatically and you can sleep well, because its much harder for naturally generated traffic to bring your webserver down.


2. Async mode

If you really strive for performance the caches answer to a request has always to be yes. Simple done: Have a worker thread that gets a list of pages to put to cache and have the cache in case of a non hit saying yes i have something and let that be a empty string or

pregenerated result. Pretty Complicated, but caches in general suck when it comes to a high load of users wanting the same thing at once. Let that be your super overview page, that one day is broken by some superduper dev (let that be me) by using one query that takes 1 sec instead of 0.1 sec. If you run a synced cache you're doomed with your solution anyways. Better to have an empty page for 5 seconds, instead of having a big traffic jam in your mysql replication 

all wanting the same resultset. Anonymous on Jan 18 2009,

20:12

I never once referred to this as a "security layer" since it most definitely offers no security benefit whatsoever. 


Reverse proxy caches work great when an application offers no caching of its own and you need to put up a caching frontend like Squid. If an application runs the cache itself, as I'm doing here, then there is no need for one more memory hog on the server and nginx/lighty can steamroll out thousands of requests per second without breaking a sweat to a lot of concurrent users. I think you might be surprised at its performance against Squid. I personally think Nginx, coupled with a self caching application, is a pretty solid solution for high load in a 1-3 server setup.

I think you also underestimate the value of alternatives - how many hosts offer PHP again? 

How many people have more than one server? Not everyone on the planet can install Squid, let alone configure PHP even.

Static caching mitigates this since 1) you don't need to use memory, 2) it's dumb and simple, 3) configuration is an INI file away.


As for whether static caching is worth pursuing - well, go complain to Wordpress users how wrong they are. Or pick some other commonly cached PHP application. Static file caching is a basic optimisation and you'll find it commonly used and supported by many other frameworks.

You should also bear in mind page caching is not the One True Solution - one should also cache expensive SQL queries, large resultsets, partial output on dynamic pages, expensive processing tasks, etc. It's the totality of all caching measures that count - not one in isolation. So absolutely, my solution as it stands in isolation has its problems 

. There are lots of concerns to worry about here, and I still have a few parts of this little series to go to explore them in 

. Anonymous on Jan 18 2009, 22:24

hmm as much as I like lighty and especially the ideas behind nginx:

Both products would, in our environment) mean great changes. While I am concerned on the dev base at the nginx project I have no fully async ssi/cache feature on lighty (middle 2007 I checked last 

)

So the usage of a reverse proxy is great.

For a beginner state, the provided solution is good. But as soon as your load goes up, beyond what one server can handle, you're staring right at the beginning.

Yes, it's true, you need more than one cache. But one hint for every beginner php website that could save one or the other weekend: caching, when applied that way, is a big risk.

What is happening the day you need to empty your caches? May it be a server restart or some other stuff.

I am bolding this up everytime when a dev says: "Nah this is cached", it is as long as your caches work.

This is where fully async caches come to action. Only nginx has that feature as far as i did research. Remains the problem that, in my opinion, nginx is still too small of a project to really rely on it.

The cewl thing of a selfmade async mechanism and the combination with squid is having standard components with a lot of documentation and people who can operate i easily.

One additional thing: We replaced all filecached base dthings as described here with memcached backends. Especially when you have 15 Webservers all asking the same rude (oerformacnewise) page from your app, you have a epic win. In terms of surviving .

Looks like i need to logpost all that stuff.

/me is waiting for the next part of the series. Anonymous on Jan 19 2009, 00:36

I agree with one thing - if you have 15 servers caching as I describe will be...painful. Expiring centrally might make me cry 

Nginx is a small project, but it's used an awful lot without problems. I haven't had any issue worth complaining about since I started using it over a year ago. It's main weirdness is that it is an originally Russian documented project so older documentation is hard to follow. Anonymous on Jan 19 2009, 01:40

Static html page caching is a great idea. It's a shame the implementation is a bit messy.

Are there any proposals to get this backend into the core?

I will definitely be implementing that strategy into my site.

Thank you. Anonymous on Jan 20 2009, 01:12

I'm drawing up a proposal document to explain it, formalise its goals, and add in references to this prototype implementation. The code is messy, definitely, since it's a prototype I put together for the article to explain things.

If the proposal is accepted I'd love to move onto a more formal process with unit testing but I fear it's main problem is going to be Zend_Cache itself since it badly needs some refactoring to allow for backends and frontends which are not heavily restricted with rules like "Cache ID must be alphanumeric". It's those restrictions which require workarounds like the Adapter and hexadecimal encoding of cache ids to evade the validation methods. Anonymous on Jan 20 2009, 01:28

Note: you may have to supply a cache_file_umask of 644 since these cached files are going to be read by the webserver.

Changing the default to read by group and other may be more suitable for these kinds of files. Anonymous on Jan 20 2009, 07:11

Here is an option if you don't want to retrieve at the cached files for POST:

```
RewriteCond %{REQUEST_METHOD} ^POST
RewriteRule .* - [S=4]
RewriteRule ^(.*)/$ /$1 [QSA]
RewriteRule ^$ html/index.html [QSA]
RewriteRule ^([\^.]+)/$ html/$1.html [QSA]
RewriteRule ^([\^.]�$ html/$1.html [QSA] Anonymous on Jan 20 2009, 13:23
```