

Zend Framework Page Caching: Part 3b: Tagging For Static File Caches

Continued from [Part 3...](#)

Let's make two more changes to complete the process. Here's the new Controller from earlier where we set the tag "entries" on any Actions where blog entries are displayed or listed.

```
<?php
class EntryController extends Zend_Controller_Action
{
    public function init()
    {
        $this->_helper->cache(array('index'), array('tag1','entries','tag3'));
        $this->_helper->cache->useCleaner('entry', array('process','delete'));
    }
    public function indexAction()
    {
        // show some entries, page should be cached
    }
    public function processAction()
    {
        // store the blog entry by whatever means necessary
    }
    public function deleteAction()
    {
        // delete one or more entries
    }
}
```

Based on the tags being used, we can now simplify our previously nightmarish Cleaner to a single line!

```
<?php
class ZFExt_Cache_Cleaner_Abstract
{
    protected $_cache = null;
    public function __construct()
    {
        // Needing the Action Helper here may suggest the need to extract the
        // functionality common to Helpers and Cleaners into it's own
        // class for sharing
        // Being an article, let's skip the obvious refactoring need before this
        // becomes another book...
        if (Zend_Controller_Action_HelperBroker::hasHelper('cache')) {
            $this->_cache = Zend_Controller_Action_HelperBroker::getExistingHelper('cache');
        }
        $this->_cache = Zend_Controller_Action_HelperBroker::getStaticHelper('cache');
    }
}
```


```

class EntryCleaner extends ZFExt_Cache_Cleaner_Abstract
{
    public function afterProcess()
    {
        $this->_cache->removeTaggedPageCache(array('entries'));
    }
    public function afterDelete()
    {
        $this->_cache->removeTaggedPageCache(array('entries'));
    }
}
}

```

I think we'll stop here for now!


Conclusion

Over the last three (well, four if you count I ran out of space on Part 2 and needed a 2b ) parts of this weekend series I've covered page caching with a focus on caching output to static HTML files to offload work from Apache and PHP.

This caching strategy allows the webserver to avoid PHP completely, or alternatively for a frontend nginx/lighty reverse proxy to avoid Apache completely. Either way, it results in a very fast and efficient caching mechanism for entire pages which can be linked to Tags and Cleaners so they are updated only when the data they were originally generated from has changed.

It should be noted that while static HTML caching is one of the fastest possible caching mechanisms, it is not the most flexible. It is best suited to standalone, or small 2-3 unit scaled solutions, where filesystem operations can be unified with a frontend HTTP server. Trying to sync this style of caching across servers is to be avoided. It is also not suitable for highly dynamic pages which continually update or are specific to each visitor. In these cases the more expensive caching methods are often needed, though if the dynamic portions are small enough it might be worth delegating the generation of those dynamic portions to AJAX requests from the statically cached page, so that at least the bulk of any page is cached.

Additionally, once you do reach the point where static caching cannot be used, you should of course examine other caching options across the parts of the application impacted. Remember that you can cache database results, individual template output (even partials), CPU or memory intensive operations, etc.

In closing, I hope this series has offered a few good ideas 

Posted by Pádraic Brady in Irishisms, Zend Framework at 01:33

Zend Framework Page Caching: Part 3: Tagging For Static File Caches

Tracking What's Being Cached

Expiring multiple caches linked to a specific change is easy to accomplish using Tagging, where we tag caches with keywords and clean caches based on those keywords. Unfortunately, static files can't be tagged in the normal way since their filenames must be constant. To deliver a similar system static file caching, we need to tag caches outside of the cache filename/contents itself which requires a Model or similar storage backend to keep track of tags and the caches to which they relate.

In a sense, we're creating a cache within a cache. Although this is the simplest approach for small numbers of tags and URLs, a more robust system would be backed by a database to allow a greater degree of flexibility in minimising the size of the overall resultset needed to be loaded for any given page request.

For now, the form of the inner cache will be a simple array where for each Tag, we assign a list of tagged Request URIs. This array can then be cached either to a file or a memory slot in, for example, APC for retrieval in future requests. Its obvious flaw, without a database, is that it will be loaded in full for every request where the relevant Page Cache is utilised. We are applying some lazy loading however, but when it gets big enough the move to a database may be needed.

Let's start by adapting ZFExt_Controller_Action_Helper_Cache to host an array of tags which is what we will cache and retrieve from the inner cache when any Tag operations are utilised. We'll also make it possible to set tags when setting what Actions need to be cached. This will result in a system where tags are assigned for any cache we want, and those tags will be saved to the inner cache when the Action completes (using postDispatch()). We'll also add a new removeTaggedPageCache() method which we can either call directly from an Action, or from a Cleaner class.

`<?php`

```
class ZFExt_Controller_Action_Helper_Cache extends Zend_Controller_Action_Helper_Abstract
{
    protected $_caches = array();
    protected $_cleaners = array();
    protected $_caching = array();
    protected $_obstarted = false;
    protected $_tags = array();
    protected $_tagged = array();
    public function addCache($cacheld, $cache) {
        if (!$cache instanceof Zend_Cache_Core && !$cache instanceof
ZFExt_Cache_Backend_Static_Adapter) {
            throw new Exception("Need to provide a valid cache!");
        }
        $this->_caches[$cacheld] = $cache;
    }
    public function createCache($cacheld, $frontend, $backend, $frontendOptions = array(),
$backendOptions = array(), $customFrontendNaming = false, $customBackendNaming = false,
$autoload = false) {
        $cache = Zend_Cache::factory($frontend, $backend, $frontendOptions, $backendOptions,
$customFrontendNaming, $customBackendNaming, $autoload);
        $this->addCache($cacheld, $cache);
        return $cache;
    }
    public function getCache($cacheld) {
        if ($this->hasCache($cacheld)) {
            return $this->_caches[$cacheld];
        }
        return false;
    }
}
```

```

public function hasCache($cacheld) {
    if (isset($this->_caches[$cacheld])) {
        return true;
    }
    return false;
}
// Pass array of actions to cache for the current Controller
// optionally pass array of tags to assign.
public function direct(array $actions, array $tags = array()) {
    $controller = $this->getRequest()->getControllerName();
    foreach ($actions as $action) {
        if (!isset($this->_caching[$controller])) {
            $this->_caching[$controller] = array();
        }
        if (!isset($this->_caching[$controller][$action])) {
            $this->_caching[$controller][] = $action;
        }
        if (!empty($tags)) {
            if (!isset($this->_tags[$controller])) {
                $this->_tags[$controller] = array();
            }
            if (!isset($this->_tags[$controller][$action])) {
                $this->_tags[$controller][$action] = array();
            }
            foreach ($tags as $tag) {
                if (!in_array($tag, $this->_tags[$controller][$action])) {
                    $this->_tags[$controller][$action][] = $tag;
                }
            }
        }
    }
}
// Remove page caches based on URL, with recursive matching directory
// removal for those where, for example, pagination is also being cached.
// Sec: remember what they say about "rm -R" - checks needed
public function removePageCache($relativeUrl, $recursive = false) {
    if ($recursive) {
        $this->getCache('page')->removeRecursive($relativeUrl);
    } else {
        $this->getCache('page')->remove($relativeUrl);
    }
}
// delete all statically cached files which are associated with the
// given array of tags
public function removeTaggedPageCache(array $tags)
{
    return $this->getCache('page')->clean(Zend_Cache::CLEANING_MODE_MATCHING_TAG, $tags);
}
// create a nested array assigning cleaners to various
// controller+action combinations
public function useCleaner($cleanerName, array $actions)
{
    foreach ($actions as $action) {

```

```

$controller = $this->getRequest()->getControllerName();
if (!isset($this->_cleaners[$controller])) {
    $this->_cleaners[$controller] = array();
}
if (!isset($this->_cleaners[$controller][$action])) {
    $this->_cleaners[$controller][$action] = array();
}
if (!isset($this->_caching[$controller][$action][$cleanerName])) {
    $this->_cleaners[$controller][$action][] = $cleanerName;
}
}
}
}
// Commence caching for matching Actions
// Will exit if caching has already started
public function preDispatch()
{
    $controller = $this->getRequest()->getControllerName();
    $action = $this->getRequest()->getActionName();
    if (!empty($this->_caching)) {
        if (isset($this->_caching[$controller]) &&
            in_array($action, $this->_caching[$controller])) {
            // do not start caching if started earlier in cycle
            // otherwise commence caching here
            $stats = ob_get_status(true);
            foreach ($stats as $status) {
                if ($status['name'] == 'Zend_Cache_Frontend_Page::_flush') {
                    return;
                }
            }
            $this->getCache('page')->start();
            $this->_obstarted = true;
        }
    }
}
// Run cache cleaning operations after actions are dispatched
// enforces Cleaner methods as being "after{ActionMethod}"
// Store the revised Tagged array into the inner cache.
public function postDispatch()
{
    if (!empty($this->_cleaners)) {
        $controller = $this->getRequest()->getControllerName();
        $action = $this->getRequest()->getActionName();
        if (isset($this->_cleaners[$controller][$action])) {
            $cleanerNames = $this->_cleaners[$controller][$action];
            foreach ($cleanerNames as $cleanerName) {
                $cleaner = $this->createCleaner($cleanerName);
                $method = 'after' . ucfirst($action);
                $cleaner->{$method}();
            }
        }
    }
    if ($this->_obstarted) {
        $this->getCache('page')->end();
    }
}

```

```

}
if (isset($this->_caching[$controller]) &&
in_array($action, $this->_caching[$controller])) {
    $requestUri = $this->getRequest()->getRequestUri();
    $this->_tagged = $this->_loadTagged();
    if (isset($this->_tags[$controller][$action]) && !empty($this->_tags[$controller][$action])) {
        foreach ($this->_tags[$controller][$action] as $tag) {
            if (!isset($this->_tagged[$tag])) {
                $this->_tagged[$tag] = array();
            }
            if (!in_array($requestUri, $this->_tagged[$tag])) {
                $this->_tagged[$tag][] = $requestUri;
            }
        }
    }
    $this->_saveTagged();
}
}

// Cheat by stealing functionality from the Dispatcher! Haha!
// In a real class, should really implement this natively
// to keep down on dependencies, and allow cleaners to
// exist elsewhere. Also this is not Module friendly yet.
public function createCleaner($cleanerName)
{
    $dispatcher = $this->getFrontController()->getDispatcher();
    $className = $cleanerName . 'Cleaner';
    $finalClassName = $dispatcher->loadClass($className);
    $cleaner = new $finalClassName;
    return $cleaner;
}

// load cached Tags from inner cache
protected function _loadTagged()
{
    if (!$this->hasCache('tagged')) {
        throw new Zend_Exception('No "tagged" cache has been defined therefore Tagging cannot be
utilised');
    }
    if ($result = $this->getCache('tagged')->load('zfextcache_tagged')) {
        $this->_tagged = $result;
    }
}

// save existing Tags to inner cache
protected function _saveTagged()
{
    if (!$this->hasCache('tagged')) {
        throw new Zend_Exception('No "tagged" cache has been defined therefore Tagging cannot be
utilised');
    }
    $this->getCache('tagged')->save($this->_tagged, 'zfextcache_tagged');
}
}
}

```

You'll notice that the Helper refers to a specific cache by name, and that the Static backend (presented below) refers to the same cache. This partial duplication may be another sign that the Action Helper is taking on too much responsibility, indicating the need for a discrete Cache Management class to centralise the inner cache with.

Based on the above, we make some changes to the ZFExt_Cache_Backend_Static class to support cleaning the static cache files based on associated tags.

```
<?php
```

```
class ZFExt_Cache_Backend_Static extends Zend_Cache_Backend implements
```

```
Zend_Cache_Backend_Interface
```

```
{
    const DEBUG_HEADER = 'DEBUG HEADER : This is a cached page !';
    // Available options
    protected $_options = array(
        'public_dir' => null,
        'file_extension' => '.html',
        'index_filename' => 'index',
        'file_locking' => true,
        'cache_file_umask' => 0600,
        'debug_header' => false
    );
    protected $_innerCache = null;
    // 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 (empty($id) || $id == '_') {
            $id = $this->_detectId();
        }
        if (!$this->_verifyPath($id)) {
            Zend_Cache::throwException('Invalid cache id: does not match expected public_dir path');
        }
        if ($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)) {
            $content = file_get_contents($file);
            // move debug header to Frontend to prevent these gymnastics
            return str_replace(self::DEBUG_HEADER, "", $content);
        }
        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);
    if (!$this->_verifyPath($id)) {
        Zend_Cache::throwException('Invalid cache id: does not match expected public_dir path');
    }
    $fileName = basename($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
// Note: We're ignoring the ID as its not required.
public function save($data, $id, $tags = array(), $specificLifetime = false)
{
    clearstatcache();
    $requestUri = $this->_detectId();
    $fileName = basename($requestUri);
    if (empty($fileName)) {
        $fileName = $this->_options['index_filename'];
    }
    $pathName = $this->_options['public_dir'] . dirname($requestUri);
    if (!file_exists($pathName)) {
        mkdir($pathName, $this->_options['cache_file_umask'], true);
    }
    if ($id !== '_') { // empty ID since a Capture
        $dataUnserialized = unserialize($data);
    } else {
        $dataUnserialized = array();
        $dataUnserialized['data'] = $data;
    }
    if ($this->_options['debug_header']) {
        $dataUnserialized['data'] =
            self::DEBUG_HEADER . $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) > 0) {
        $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);
    if (!$this->_verifyPath($id)) {
        Zend_Cache::throwException('Invalid cache id: does not match expected public_dir path');
    }
    $fileName = basename($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 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);
    if (!$this->_verifyPath($id)) {
        Zend_Cache::throwException('Invalid cache id: does not match expected public_dir path');
    }
    $fileName = basename($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(dirname($path));
    }
    if (file_exists($file)) {
        if (!is_writable($file)) {
            return false;
        }
    }
}

```

```

    return unlink($file);
}
}
// Clean some cache records
// Not implemented here since we would need a backend tagging system given
// that static files themselves cannot be tagged in the filename.
public function clean($mode = Zend_Cache::CLEANING_MODE_ALL, $tags = array())
{
    switch ($mode) {
        case Zend_Cache::CLEANING_MODE_MATCHING_TAG:
            if (empty($tags)) {
                throw new Zend_Exception('Cannot use mode
Zend_Cache::CLEANING_MODE_MATCHING_TAG as no tags were defined');
            }
            $innerCache = $this->getInnerCache();
            if (!$tagged = $innerCache->load('zfextcache_tagged')) {
                throw new Zend_Exception('No "tagged" cache has been defined therefore Tagging
cannot be utilised');
            }
            foreach ($tags as $tag) {
                if (isset($tagged[$tag]) && !empty($tagged[$tag])) {
                    foreach ($tagged[$tag] as $requestUri) {
                        // no recursive handling here...
                        // hex conversion evades Zend_Cache_Core private validation
                        $this->remove(bin2hex($requestUri));
                        $index = array_search($requestUri,$tagged[$tag]);
                        unset($tagged[$tag][$index]);
                    }
                }
            }
            $innerCache->save($tagged, 'zfextcache_tagged');
            break;
        case Zend_Cache::CLEANING_MODE_ALL:
        case Zend_Cache::CLEANING_MODE_OLD:
        case Zend_Cache::CLEANING_MODE_NOT_MATCHING_TAG:
        case Zend_Cache::CLEANING_MODE_MATCHING_ANY_TAG:
            $this->log("ZFExt_Cache_Backend_Static : Selected Cleaning Mode Currently
Unsupported By This Backend");
            break;
        default:
            Zend_Cache::throwException('Invalid mode for clean() method');
            break;
    }
}
public function setInnerCache(Zend_Cache_Core $cache)
{
    $this->_innerCache = $cache;
}
public function getInnerCache()
{
    if (is_null($this->_innerCache)) {
        Zend_Cache::throwException('An Inner Cache has not been set; use setInnerCache()');
    }
}

```

```

    return $this->_innerCache;
}
// Encoded by ZFExt_Cache_Backend_Static_Adapter
protected function _decodeId($id)
{
    // another workaround since Zend_Cache_Core prevents
    // empty or null IDs which we'll have when Capturing
    // and before the REQUEST_URI is checked
    if ($id == '_') {
        return "";
    }
    return pack('H*', $id);
}
// "Danger, Will Robinson!"
// Sanity check to ascertain whether path is within the configured
// public_dir path
protected function _verifyPath($path)
{
    $path = realpath($path);
    $base = realpath($this->_options['public_dir']);
    return strcmp($path, $base, strlen($base)) !== 0;
}
protected function _detectId()
{
    // should strip query strings in future
    // along with other fragments
    return $_SERVER['REQUEST_URI'];
}
}

```

We now have support for cleaning the cache using matching tags. This relies on the Action Helper and Static Backend utilising the same cache which can be created as normal and passed to both from our 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()
    {
        $frontend = new ZFExt_Cache_Frontend_Capture();
    }
}


```

```

$backendOptions = array(
    'debug_header' => true,
    'public_dir' => self::$root . '/public'
);
$backend = new ZFExt_Cache_Backend_Static($backendOptions);
// use our Adapter to deal with the Core's private validation
$cache = new ZFExt_Cache_Backend_Static_Adapter(
    Zend_Cache::factory($frontend, $backend)
);
// create an inner cache so the backend can store tags
$taggedCache = Zend_Cache::factory('Core', 'File',
    array('automatic_serialization'=>true),
    array('cache_dir'=>self::$root . '/cache')
);
$backend->setInnerCache($taggedCache);
// Add the new cache to the Cache Control Action Helper
Zend_Controller_Action_HelperBroker::addPrefix('ZFExt_Controller_Action_Helper');
Zend_Controller_Action_HelperBroker::getStaticHelper('Cache')->addCache('page', $cache);
Zend_Controller_Action_HelperBroker::getStaticHelper('Cache')->addCache('tagged',
$taggedCache);
}
}

```

Et voilà ! We now have a prototype static HTML caching system, fully controllable from either the bootstrap or our Controllers, and which supports a tag system meaning we can tag lots of related Actions, and expire them all in one go rather than worry about what they all are up front.

This entry continues in [Part 3b](#) due to space restrictions in Serendipity which may yet drive me insane... 

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 01:14

Zend Framework Page Caching: Part 2: Controller Based Cache Management

Controller Based Cache Controls

Caches are really cool, but when you get right down to it dumping them into Zend_Registry is ugly. I really really obsessively like Dependency Injection, and when you throw in a DI problem with cache control within the application it won't be long before you're itching for a simple cache management method. One of the simplest solutions for controlling caches from Controllers is to create a new Action Helper specifically for that purpose. This new helper will interface with a named cache (so it holds and can tidy up control for numerous registered caches) stored either within itself (better injection) or Zend_Registry (no so pretty injection). Here's one I prepared earlier.

```
<?php
```

```
class ZFExt_Controller_Action_Helper_Cache extends Zend_Controller_Action_Helper_Abstract
```

```
{
    protected $_caches = array();
    public function addCache($cacheld, $cache) {
        if (!$cache instanceof Zend_Cache_Core && !$cache instanceof
ZFExt_Cache_Backend_Static_Adapter) {
            throw new Exception("Need to provide a valid cache!");
        }
        $this->_caches[$cacheld] = $cache;
    }
    public function createCache($cacheld, $frontend, $backend, $frontendOptions = array(),
$backendOptions = array(), $customFrontendNaming = false, $customBackendNaming = false,
$autoload = false) {
        $cache = Zend_Cache::factory($frontend, $backend, $frontendOptions, $backendOptions,
$customFrontendNaming, $customBackendNaming, $autoload);
        $this->addCache($cacheld, $cache);
        return $cache;
    }
    public function getCache($cacheld) {
        if ($this->hasCache($cacheld)) {
            return $this->_caches[$cacheld];
        }
        return false;
    }
    public function hasCache($cacheld) {
        if (isset($this->_caches[$cacheld])) {
            return true;
        }
        return false;
    }
}
// Remove page caches based on URL, with recursive matching directory
// removal for those where, for example, pagination is also being cached.
// Sec: remember what they say about "rm -R" - checks needed
```



```

    Zend_Cache::factory('Page', $backend, $frontendOptions)
);
// Add the new cache to the Cache Control Action Helper
Zend_Controller_Action_HelperBroker::addPrefix('ZFExt_Controller_Action_Helper');
Zend_Controller_Action_HelperBroker::getStaticHelper('Cache')->addCache('page', $cache);
// cache all output after this point to static HTML
// assuming caching is enabled for the current URL
$cache->start();
}
}

```

This is going well! The stage is now set, so let's revisit our problem! As we mentioned earlier, setting up a static page cache would mean all requests for that page would never ever see PHP or our application. In order to get rid of these static caches, we need to manually invalidate and remove them whenever the underlying data (or whatever) has changed. One way of doing this is to use the above Action Helper whenever such changes are passing through a relevant Controller.

Let's imagine, based on the cached output from our `/blog/tags/zend-framework` route earlier, there is a Controller for adding new blog entries. Obviously, once a new entry is added we might want to update any pages linked to the tags for that entry. You might do something along the lines of:

```

<?php
class EntryController extends Zend_Controller_Action
{
    public function processAction()
    {
        // store the blog entry by whatever means necessary, then...

        // Also clear index page to show new entry!
        $this->_helper->cache->removePageCache('/');
        // Need to add support for variations on
        // the URL matching the route in future
        // "/" is equivalent to "/index" and "/index/index"!
        // And.... "/default/index/index"!
        $this->_helper->cache->removePageCache('/index');
        $this->_helper->cache->removePageCache('/index/index');
        $this->_helper->cache->removePageCache('/default/index/index');

        // You could do this all day...
        // based on tags for the new entry, selectively clear effected static
        // HTML caches (we'll assume pagination is used, and remove recursively)
        foreach ($tags as $tag) {
            $this->_helper->cache->removePageCache('/blog/tags/' . $tag, true);
            $this->_helper->cache->removePageCache('/default/blog/tags/' . $tag, true);
        }
    }
}
}

```

Hmm, we appear to have hit a snag using our Cache Helper, but nothing we can't fix. The problem is that any change might spark a massive net of cache expirations across all equivalent URLs. Tracking these manually

is difficult within the Controller, so it's time to extract the cache removals to a separate class, from which we can have a better view of the the situation and perhaps arrive at a solution.

Extracting Cache Controls

One fairly common approach to use when Controller based cache control is becoming too complicated, is to allow the caches to do all the work and indirectly observe actions. The theory is a simple one to apply to the Zend Framework, we add an Action Helper (or change the one we just added!) which detects or is told which Actions have been performed and, based on a list of Controller Actions which effect caches, the Helper can then perform a set of cache removals associated with that Action. The actual cache removals are extracted from the Controller and maintained in a more easily configurable class.

Since these new classes are not going to be driven from a configuration file as such, but are all written by hand given how cache mapping works, they should be merged into the same category as Controllers, Views and Models. Unfortunately it's not so easy or fast to design yet another class location without inflection or directory/classname maps. So I've elected to do what I usually do, and hijack the functionality of a similar mapping system. By assuming all "cleaner" classes are located within /application/{module}/controllers (same as Controllers), we can reuse the Controller loading logic that exists in the Dispatcher of the framework. When we have more time, we could create support for an actual /cleaners directory. Here's the updated ZFExt_Controller_Action_Helper_Cache class with some additions.

```
<?php
```

```
class ZFExt_Controller_Action_Helper_Cache extends Zend_Controller_Action_Helper_Abstract
{
    protected $_caches = array();
    protected $_cleaners = array();
    public function addCache($cacheld, $cache) {
        if (!$cache instanceof Zend_Cache_Core && !$cache instanceof
ZFExt_Cache_Backend_Static_Adapter) {
            throw new Exception("Need to provide a valid cache!");
        }
        $this->_caches[$cacheld] = $cache;
    }
    public function createCache($cacheld, $frontend, $backend, $frontendOptions = array(),
$backendOptions = array(), $customFrontendNaming = false, $customBackendNaming = false,
$autoload = false) {
        $cache = Zend_Cache::factory($frontend, $backend, $frontendOptions, $backendOptions,
$customFrontendNaming, $customBackendNaming, $autoload);
        $this->addCache($cacheld, $cache);
        return $cache;
    }
    public function getCache($cacheld) {
        if ($this->hasCache($cacheld)) {
            return $this->_caches[$cacheld];
        }
        return false;
    }
    public function hasCache($cacheld) {
        if (isset($this->_caches[$cacheld])) {
            return true;
        }
    }
}
```

```

}
return false;
}
// Remove page caches based on URL, with recursive matching directory
// removal for those where, for example, pagination is also being cached.
// Sec: remember what they say about "rm -R" - checks needed
public function removePageCache($relativeUrl, $recursive = false) {
    if ($recursive) {
        $this->getCache('page')->removeRecursive($relativeUrl);
    } else {
        $this->getCache('page')->remove($relativeUrl);
    }
}
// create a nested array assigning cleaners to various
// controller+action combinations
public function useCleaner($cleanerName, array $actions)
{
    foreach ($actions as $action) {
        $controller = $this->getRequest()->getControllerName();
        if (!isset($this->_cleaners[$controller])) {
            $this->_cleaners[$controller] = array();
        }
        if (!isset($this->_cleaners[$controller][$action])) {
            $this->_cleaners[$controller][$action] = array();
        }
        if (!isset($this->_caching[$controller][$action][$cleanerName])) {
            $this->_cleaners[$controller][$action][] = $cleanerName;
        }
    }
}
// Run cache cleaning operations after actions are dispatched
// enforces Cleaner methods as being "after{ActionMethod}"
public function postDispatch()
{
    if (!empty($this->_cleaners)) {
        $controller = $this->getRequest()->getControllerName();
        $action = $this->getRequest()->getActionName();
        if (isset($this->_cleaners[$controller][$action])) {
            $cleanerNames = $this->_cleaners[$controller][$action];
            foreach ($cleanerNames as $cleanerName) {
                $cleaner = $this->createCleaner($cleanerName);
                $method = 'after' . ucfirst($action);
                $cleaner->{$method}();
            }
        }
    }
}
// Cheat by stealing functionality from the Dispatcher! Haha!
// In a real class, should really implement this natively
// to keep down on dependencies, and allow cleaners to
// exist elsewhere. Also this is not Module friendly yet.
public function createCleaner($cleanerName)
{

```

```

$dispatcher = $this->getFrontController()->getDispatcher();
$className = $cleanerName . 'Cleaner';
$finalClassName = $dispatcher->loadClass($className);
$cleaner = new $finalClassName;
return $cleaner;
}
}

```

The new ZFExt_Controller_Action_Helper_Cache::useCleaner() method accepts the name of a Cleaner to use for the array of actions passed to the second parameter. For example:

```
$this->_helper->cache->useCleaner('entry', array('process', 'delete'));
```

If you call this from the relevant Controller, it tells the Cache Helper to locate and instantiate the EntryCleaner class located in /application/controllers/EntryCleaner.php. When the Controller's processAction() method finishes, the Cleaner's afterProcess() action method will be called (and the same for the delete action).

By now, some of you should be remembering a similar API from another framework .

To be fair, I can't take credit for inventing something that already exists but I think the ZF would benefit from the same solutions.

Here's what the Cleaner would look like (Abstract parent added for completeness):

```

<?php
class ZFExt_Cache_Cleaner_Abstract
{
    protected $_cache = null;
    public function __construct()
    {
        // Needing the Action Helper here may suggest the need to extract the
        // functionality common to Helpers and Cleaners into it's own
        // class for sharing
        // Being an article, let's skip the obvious refactoring need before this
        // becomes another book...
        if (Zend_Controller_Action_HelperBroker::hasHelper('cache')) {
            $this->_cache = Zend_Controller_Action_HelperBroker::getExistingHelper('cache');
        }
        $this->_cache = Zend_Controller_Action_HelperBroker::getStaticHelper('cache');
    }
}
class EntryCleaner extends ZFExt_Cache_Cleaner_Abstract
{
    public function afterProcess()
    {
        // We won't cover it yet, but Cache control needs a way to pass
        // contexts to the Cache Cleaner (the same as Views need Models to function)
        foreach ($this->_cache->tags as $tag) {
            $this->_cache->removePageCache('/blog/tags/' . $tag, true);
            $this->_cache->removePageCache('/default/blog/tags/' . $tag, true);
        }
        // remove all possible cached routes to Index page
    }
}

```

```

$this->_cache->removePageCache('/');
$this->_cache->removePageCache('/index');
$this->_cache->removePageCache('/index/index');
$this->_cache->removePageCache('/default/index/index');
}
public function afterDelete()
{
    // similar cache expiration for deletes (or extract common
    // expirations to shared protected methods).
}
}
}

```

With the extracted Cleaner in place, here's what the original Controller moves to:

```

<?php
class EntryController extends Zend_Controller_Action
{
    public function init()
    {
        $this->_helper->cache->useCleaner('entry', array('process','delete'));
    }
    public function processAction()
    {
        // store the blog entry by whatever means necessary
    }
    public function deleteAction()
    {
        // delete one or more entries
    }
}
}

```

Ticking our boxes, everything seems to be working with these changes. We still have two problems though.

URL variations that match the same route will create different caches depending on the URL used, so all cache expiries need to account for all alternative but equivalent URLs. Secondly, our Cleaner classes will work fine for limited cache management, but the more complex things get the more difficult to maintain it will become.

In Part 3 we'll attempt to deal with this problem, but for now let's quickly add one more piece of functionality and present the final version of all concerned classes until next time.

Integrating Cache Initialisation Into Controllers

Since we can now expire static file caches from Controllers, it stands to reason we can also create them the same way!

This needs a few more small changes to all classes (mainly for that Frontend private static method I referred to in Part 1), Here's our 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 removeRecursive($id) {
        $this->_cache->getBackend()->removeRecursive($id);
    }
    public function __call($method, array $args)
    {
        return call_user_func_array(array($this->_cache, $method), $args);
    }
    protected function _encodeId($id) {
        if (!isset($id) || empty($id)) {
            return '_';
        }
        return bin2hex($id);
    }
}
}

```

The `_encodeId()` method now does some mystery underscore encoding (basically it should be null, but `Zend_Cache_Core` forbids having a null ID). Let's move onto a replacement for the current `Zend_Cache_Frontend_Page` class we've been using so far. While it's great, it has a few assumptions and is not completely suitably. Here's a new custom Frontend which allows you to Capture output without being too concerned over IDs:

```

<?php
class ZFExt_Cache_Frontend_Capture extends Zend_Cache_Core
{

```



```

$customFrontendNaming, $customBackendNaming, $autoload);
    $this->addCache($cacheld, $cache);
    return $cache;
}
public function getCache($cacheld) {
    if ($this->hasCache($cacheld)) {
        return $this->_caches[$cacheld];
    }
    return false;
}
public function hasCache($cacheld) {
    if (isset($this->_caches[$cacheld])) {
        return true;
    }
    return false;
}
}
// Pass array of actions to cache for the current Controller
public function direct(array $actions) {
    $controller = $this->getRequest()->getControllerName();
    foreach ($actions as $action) {
        if (!isset($this->_caching[$controller])) {
            $this->_caching[$controller] = array();
        }
        if (!isset($this->_caching[$controller][$action])) {
            $this->_caching[$controller][] = $action;
        }
    }
}
}
// Remove page caches based on URL, with recursive matching directory
// removal for those where, for example, pagination is also being cached.
// Sec: remember what they say about "rm -R" - checks needed
public function removePageCache($relativeUrl, $recursive = false) {
    if ($recursive) {
        $this->getCache('page')->removeRecursive($relativeUrl);
    } else {
        $this->getCache('page')->remove($relativeUrl);
    }
}
}
// create a nested array assigning cleaners to various
// controller+action combinations
public function useCleaner($cleanerName, array $actions)
{
    foreach ($actions as $action) {
        $controller = $this->getRequest()->getControllerName();
        if (!isset($this->_cleaners[$controller])) {
            $this->_cleaners[$controller] = array();
        }
        if (!isset($this->_cleaners[$controller][$action])) {
            $this->_cleaners[$controller][$action] = array();
        }
        if (!isset($this->_caching[$controller][$action][$cleanerName])) {
            $this->_cleaners[$controller][$action][] = $cleanerName;
        }
    }
}
}

```

```

}
}
// Commence caching for matching Actions
// Will exit if caching has already started
public function preDispatch()
{
    if (!empty($this->_caching)) {
        $controller = $this->getRequest()->getControllerName();
        if (isset($this->_caching[$controller]) &&
            in_array($this->getRequest()->getActionName(), $this->_caching[$controller])) {
            // do not start caching if started earlier in cycle
            // otherwise commence caching here
            $stats = ob_get_status(true);
            foreach ($stats as $status) {
                if ($status['name'] == 'Zend_Cache_Frontend_Page::_flush') {
                    return;
                }
            }
            $this->getCache('page')->start();
            $this->_obstarted = true;
        }
    }
}
// Run cache cleaning operations after actions are dispatched
// enforces Cleaner methods as being "after{ActionMethod}"
public function postDispatch()
{
    if (!empty($this->_cleaners)) {
        $controller = $this->getRequest()->getControllerName();
        $action = $this->getRequest()->getActionName();
        if (isset($this->_cleaners[$controller][$action])) {
            $cleanerNames = $this->_cleaners[$controller][$action];
            foreach ($cleanerNames as $cleanerName) {
                $cleaner = $this->createCleaner($cleanerName);
                $method = 'after' . ucfirst($action);
                $cleaner->{$method}();
            }
        }
    }
    if ($this->_obstarted) {
        $this->getCache('page')->end();
    }
}
// Cheat by stealing functionality from the Dispatcher! Haha!
// In a real class, should really implement this natively
// to keep down on dependencies, and allow cleaners to
// exist elsewhere. Also this is not Module friendly yet.
public function createCleaner($cleanerName)
{
    $dispatcher = $this->getFrontController()->getDispatcher();
    $className = $cleanerName . 'Cleaner';
    $finalClassName = $dispatcher->loadClass($className);
    $cleaner = new $finalClassName;
}

```

```

    return $cleaner;
}
}

```

This Article continues in Part 2b (due to the stupid character limitations of this blog which I broke!): [Part 2\(b\)](#)

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 01:49

Zend Framework Page Caching: Part 2b: Controller Based Cache Management

Continuing from Part 2!

Check out the `direct()` and `preDispatch()` methods - using these we can setup what to cache, start caching, and all without needing regular expressions since our Static backend is driven by the current REQUEST URI.

Given all the above changes, it's no surprise the Static Backend has had a few updates!

```
<?php
```

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

```

{
    const DEBUG_HEADER = 'DEBUG HEADER : This is a cached page !';
    // 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 (empty($id) || $id == '_') {
            $id = $this->_detectId();
        }
        if (!$this->_verifyPath($id)) {
            Zend_Cache::throwException('Invalid cache id: does not match expected public_dir path');
        }
        if ($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)) {

```

```

$content = file_get_contents($file);
// move debug header to Frontend to prevent these gymnastics
return str_replace(self::DEBUG_HEADER, "", $content);
}
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);
    if (!$this->_verifyPath($id)) {
        Zend_Cache::throwException('Invalid cache id: does not match expected public_dir path');
    }
    $fileName = basename($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
// Note: We're ignoring the ID as its not required.
public function save($data, $id, $tags = array(), $specificLifetime = false)
{
    clearstatcache();
    $requestUri = $this->_detectId();
    $fileName = basename($requestUri);
    if (empty($fileName)) {
        $fileName = $this->_options['index_filename'];
    }
    $pathName = $this->_options['public_dir'] . dirname($requestUri);
    if (!file_exists($pathName)) {
        mkdir($pathName, $this->_options['cache_file_umask'], true);
    }
    if ($id !== '_') { // empty ID since a Capture
        $dataUnserialized = unserialize($data);
    } else {
        $dataUnserialized = array();
        $dataUnserialized['data'] = $data;
    }
    if ($this->_options['debug_header']) {
        $dataUnserialized['data'] =
            self::DEBUG_HEADER . $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);
    if (!$this->_verifyPath($id)) {
        Zend_Cache::throwException('Invalid cache id: does not match expected public_dir path');
    }
    $fileName = basename($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 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);
    if (!$this->_verifyPath($id)) {
        Zend_Cache::throwException('Invalid cache id: does not match expected public_dir path');
    }
    $fileName = basename($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(dirname($path));
}
if (file_exists($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.
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:
            $this->_log("ZFExt_Cache_Backend_Static : Cleaning Modes Currently Unsupported By This
Backend");
            break;
        default:
            Zend_Cache::throwException('Invalid mode for clean() method');
            break;
    }
}
}
// Encoded by ZFExt_Cache_Backend_Static_Adapter
protected function _decodeId($id)
{
    // another workaround since Zend_Cache_Core prevents
    // empty or null IDs which we'll have when Capturing
    // and before the REQUEST_URI is checked
    if ($id == '_') {
        return "";
    }
    return pack('H*', $id);
}
// "Danger, Will Robinson!"
// Sanity check to ascertain whether path is within the configured
// public_dir path
protected function _verifyPath($path)
{
    $path = realpath($path);
    $base = realpath($this->_options['public_dir']);
    return strncmp($path, $base, strlen($base)) !== ;
}
protected function _detectId()
{
    // should strip query strings in future

```

```

// along with other fragments
return $_SERVER['REQUEST_URI'];
}
}

```

Two more source code listings remain - here's the revised Bootstrap fragment:

```

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()
    {
        $frontend = new ZFExt_Cache_Frontend_Capture();
        $backendOptions = array(
            'debug_header' => true,
            'public_dir' => self::$root . '/public'
        );
        $backend = new ZFExt_Cache_Backend_Static($backendOptions);
        // use our Adapter to deal with the Core's private validation
        $cache = new ZFExt_Cache_Backend_Static_Adapter(
            Zend_Cache::factory($frontend, $backend)
        );
        // Add the new cache to the Cache Control Action Helper
        Zend_Controller_Action_HelperBroker::addPrefix('ZFExt_Controller_Action_Helper');
        Zend_Controller_Action_HelperBroker::getStaticHelper('Cache')->addCache('page', $cache);
        // Assume all caching initiated by Controllers (need to detect OB otherwise)
    }
}
}

```

After all of that...our Controller has morphed into the following:


```

<?php
class EntryController extends Zend_Controller_Action
{
    public function init()
    {
        $this->_helper->cache(array('index'));
        $this->_helper->cache->useCleaner('entry', array('process','delete'));
    }
}

```

```
}  
public function indexAction()  
{  
    // show some entries, page should be cached  
}  
public function processAction()  
{  
    // store the blog entry by whatever means necessary  
}  
public function deleteAction()  
{  
    // delete one or more entries  
}  
}
```

Conclusion

Part 3 will start to address some of the weakness in the system so far (like track who has cached what and where 

), but for now the system, in its prototype format, is operational. We can now statically cache HTML output on the fly from our controllers and implement Cleaners to clear that cache as actions dictate. We've even established the beginnings of a cache management system which favours Controller based instructions over regular expression url definitions. It's still not fluid enough to be configurable, but we're very close to that level.

Part 3 tomorrow!

Posted by Pádraic Brady in Irishisms, Zend Framework at 01:45


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()
    {
        $frontendOptions = array(
            'default_options' => array(
                'cache' => false
            ),
            // Only cache URLs for Index and News controllers
            // matching the following patterns
            'regexps' => array(
                '^/$' => 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

Wednesday, January 14. 2009

Zend Framework: Surviving The Deep End, Chapter 3 - The Model Available

Another day, another chapter!

Today Chapter 3, The Model, has entered the online book. Apart from some early jitters when PHPIDS (an intrusion detection system I use) lost the plot and commenced a vendetta against Opera, the new chapter is fully operational.

I've decided to start noting in these announcements that these early Chapters are akin to drafts. While they are written as complete units, it's intended to revisit them every few weeks for improvement until a final version emerges to be forever immortalised as a first edition of the book. Your comments are therefore an invaluable guide to this process. You'll have a lot more to comment on when we get down and dirty with source code 

To mark the status of the book, I'll be implementing a colour coded status label for each chapter as a rough indication of its level of completion. It should offer some quick visual feedback on whether you're reading a rough draft or a pristine final version.

I'd also like to thank those who have donated to support the book (and my aspirations for a new Macbook Pro!). I haven't gotten around to thanking donators personally but I appreciate the early donations. I need to tinker with the donation model a bit after sorting through some ideas I have, but my intention is that while the book will remain freely available online I'd like to reward donators with some additional benefits.

Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 17:29

Monday, January 12, 2009

Zend Framework: Surviving The Deep End - Performance Optimisation For Zend Framework Applications

It took longer than expected to prepare this weeks Chapter (or Appendix rather!) but it's finally available as part of the free Zend Framework: Surviving The Deep End book online at <http://www.survivethedeepend.com>.

This week's chapters explores some of the performance issues and tactics worth being aware of when developing Zend Framework, and indeed other, applications. It's a rather long piece of work but I hope it proves enlightening. Comments are, as usual, welcome and you can attach them to any paragraph in need of attention.

In the next day or two I'll push Chapter 3 about The Model (a topic I pre-emptively released here a few weeks back) to the online book.


Special thanks to those who have commented on the book so far - remember that no comment is too small or petty!

As comments roll in and I read the appendix myself a few times, I'll compile a list of additional pointers worth adding in a future edit.


Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 03:26

Saturday, January 3. 2009

Seven Things: Chained To Infinity!

Thanks to [Rob Allen](#), I've been tagged in this chained blog posting phenomenon so here's my 7x7 to preserve my good fortunes for this coming year and stave off bankruptcy or whatever could happen if I don't 

Seven random/weird things about me:

- I don't have a formal Computer Science degree, in fact I studied Commerce originally (Honors!) and Computer Science as a sideline.
- My first programming experience was copying BASIC code from adventure books typed into a Commodore 64k a few centuries ago. I still have the Commodore - and it's still operational!
- My first frequent activity on a real PC was playing Duke Nukem 3D.
- My most recent physical defect is a scarcity of hair which started during 2008. I blame Dad. I hear he blames his Dad. Pretty sure that goes back a ways 

- I write sci-fi/fantasy short stories in some of my free time.
- Contrary to the Irish stereotype, I am not frequently intoxicated as some have suggested. That's just a vile rumour with absolutely no concrete evidence anyone who wishes to continue living can produce.
- I first learned PHP to contribute to a game called Solar Empire in 1998.

To avenge my own tagging, I'm taking out my ire on the following victims:

- [Travis Swicegood](#) since he wrote the first published GIT book and made me pay cash to read it
- [Jurrien Stutterheim](#) for volunteering to assist developing Zend_Feed_Reader (no good deed goes unpunished!)
- [Tobias Gies](#) for smugly pointing out spelling errors on my new online book!
- [Bill Karwin](#) whose contributions to Phing power the Docbook automation of the book
- [Thomas Weidner](#) who lets me safely use Gaelic characters in my ZF applications and insulate us all from the Anglic hordes.
- [Jani Hartikainen](#) since he said nice things on Twitter, and that surely can't pass without punishment!
- [D.J. Capelis](#) who I just reconnected with over Twitter - been years since we were active members of the old Solar Empire browser game developer community.

The rules according to Mr. Allen:

Link your original tagger(s), and list these rules on your blog.

Share seven facts about yourself in the post - some random, some wierd.

Tag seven people at the end of your post by leaving their names and the links to their blogs.


Let them know they've been tagged by leaving a comment on their blogs and/or Twitter.

Posted by Pádraic Brady in Irishisms, PHP General, PHP Security at 09:06

Zend Framework: Surviving The Deep End - Chapter 2 Released

Chapter 2, The Architecture Of Zend Framework Application, has been released on <http://www.survivethedeepend.com>, or more specifically [The Architecture Of Zend Framework Application](#).

Remember that you can comment on each paragraph individually, no matter how seemingly small it might

seem 

The new chapter makes an initial pass at exploring the Model-View-Controller. Chapter 3 will deal more directly with the Model.


Posted by Pádraic Brady in PHP General, PHP Security, Zend Framework at 03:12

Friday, January 2, 2009

Book Launched! Zend Framework: Surviving The Deep End


Many twists and turns have been encountered since the Summer, but the online version of Zend Framework: Surviving The Deep End has escaped into the wild with the release of Chapter 1. The online book is hosted at <http://www.survivethedeepend.com>.

Chapter 1, the Introduction, is first on the menu to open the new year. The next chapter should turn up within a few days and then we'll see about scheduling regular chapter releases. It should be noted the new website is still a work in progress and I have a few features in testing to bring it to completion and fix some final design wishes I had, but the main feature I'm pleased to have added is an inline comment system. You may now comment on (or complain about) each paragraph individually using the jQuery powered comment system. Every paragraph has an unobstrusive link next to it showing the current number of comments for that paragraph - clicking the link shows all existing comments for that paragraph and a comment form to add another.

As many of you know, the book is available online without charge. If you are feeling generous in the coming weeks I am accepting donations. I promise that Steve Jobs will appreciate the gesture when I buy that new Macbook Pro 

. Obviously a chunk of any donations will also vanish into the coffers of Slicehost.com where the book is being hosted on their excellent VPS offering.

Let me know your thoughts, and post any general comments or questions on the book or website here. I'll be adding a general end-of-page comment system to the mix during the next day or two so posting general comments here is a temporary stopgap.

Huge thanks to all my readers and PHP acquaintances who kept me motivated 

Posted by Pádraic Brady in Irishisms, PHP General, PHP Security, Zend Framework at 01:34