

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 17:49

Hi, Padraic! Thanks a lot for this series of articles about Static Caching. I try to implement it in my project. I came across with the error:

```
$backend = new ZFExt_Cache_Backend_Static($backendOptions);
$cache = new ZFExt_Cache_Backend_Static_Adapter(
    Zend_Cache::factory('Page', $backend, $frontendOptions)
);
```

but the fact is that Zend_Cache::factory does not accept the object as Backend - it needs string. Please, can you explain how to deal with this? Cheers, Natalia Anonymous on Feb 8 2009, 08:35

Double check your version of Zend Framework - I'm not sure when, but I think accepting objects only turned up in a recent version and your copy of the library might be outdated in that regard.

Also note, this is prototype code not backed by unit tests - I'm working on a revised version (which is tested!) for a set of Cache related ZF proposals over at:

<http://github.com/padraic/zfcache>.

That should be ready for community testing in another week. It would be ready almost immediately but I need to rework a few problem areas. Anonymous on Feb 9 2009, 10:51