Magento

Full Page Caching

Author: Ivan Shcherbakov

Version 1.0

Summary

1. Introduction

This document presents how to implement full page cache system to Magento project using “Smile_MageCache” module.

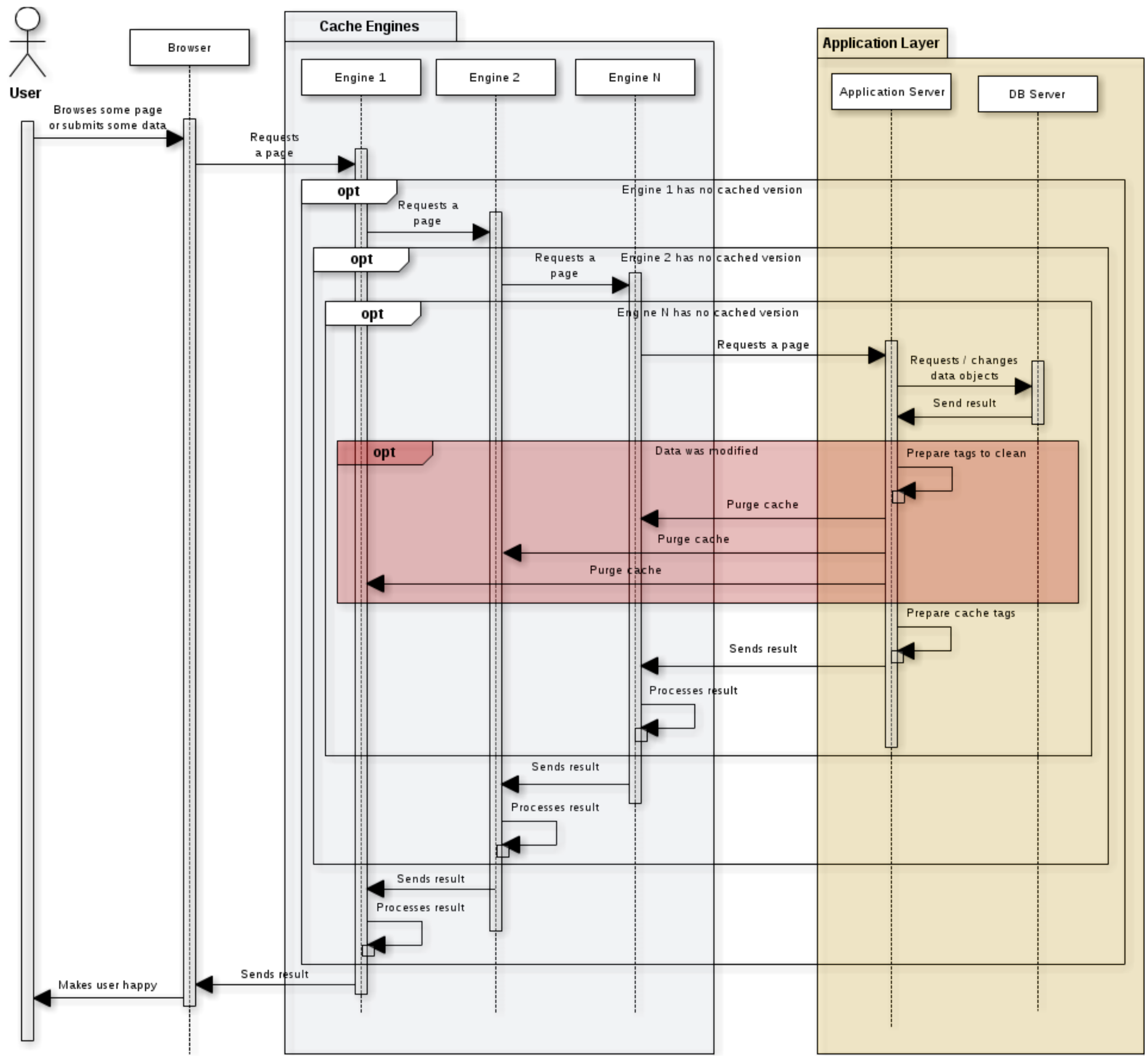
2. Basics

“Smile_MageCache” module implements 2 fundamental mechanisms required for efficient cache management:

1. **Tagging,** that is used to define all possible dependencies and attach proper tags to each page
2. **Cache invalidation,** that is used to purge an object stored in 3rd party cache engine like Varnish, CDN, etc. when one of several tags are expired

This module contains several predefined components and strategies for each of these mechanisms. It also provides a flexible API in order to customize them adapting to the needs of specific project.

The general schema of the module is following:



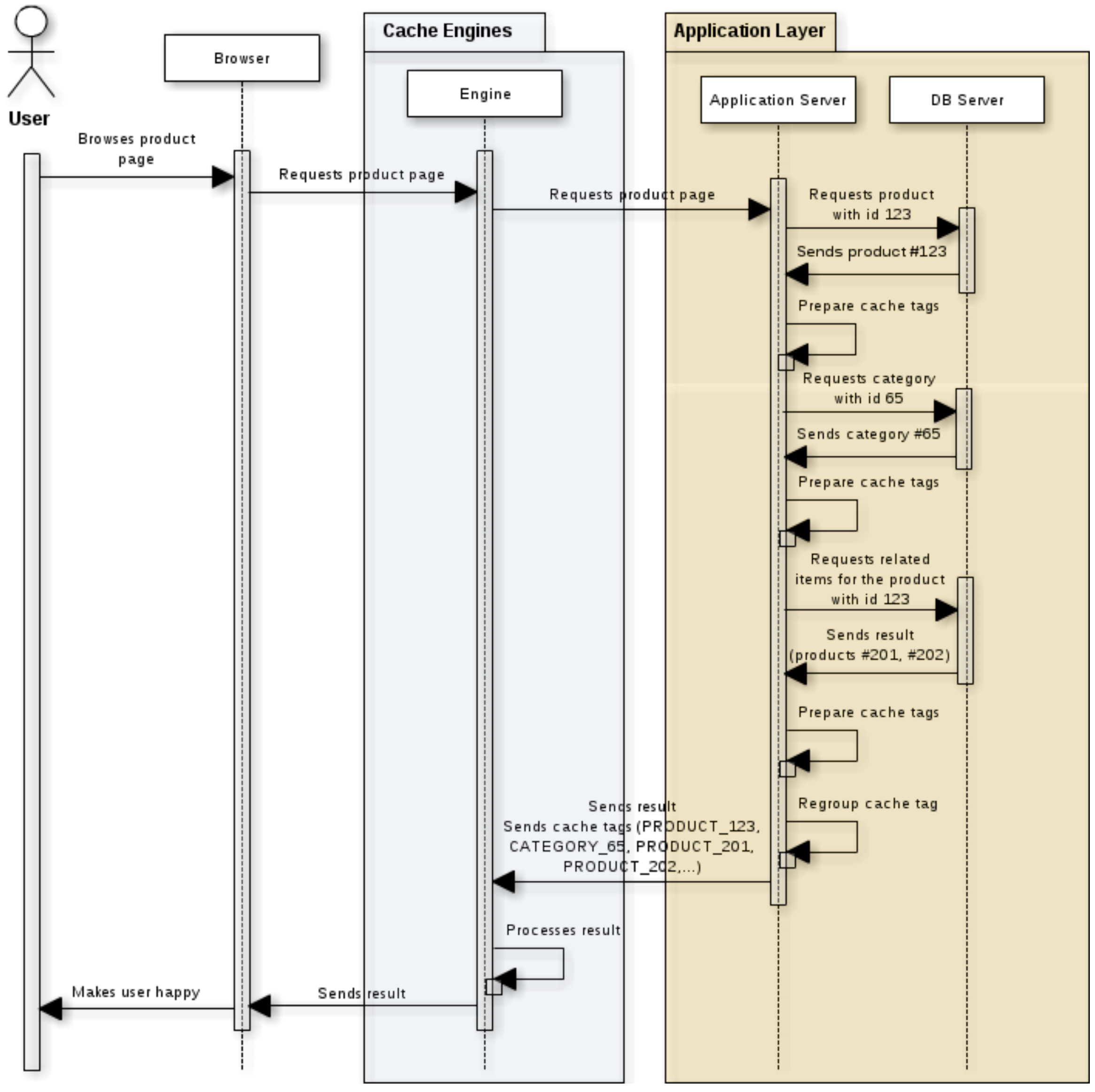
Note that invalidation mechanism can be also invoked from non-Web context (example: import script launched in CLI). In this case we don't pass through cache engines but other elements of the schema stay almost the same.

3. Tagging

This chapter covers all points of "Smile_MageCache" module related to tagging mechanism.

Basic tags

Default tag system of "Smile_MageCache" module is based on a very simple rule: *"every object used during generation of some page might impact its final view"*. That's why each time we load a single object or objects collection related cache tags will be added to the current page. For example when we work with a product page template we need to show product details, but probably some another information like category, related items and some another stuff. In order to do it we will need to retrieve correspondent data in DB. Using described approach system will automatically add required tags:



Once we OK with this principle the second question appears: *"which tags should we use to mark an object"*. The best approach in this case it to use native Magento object tags (the ones provided by `Mage_Core_Model_Abstract::getCacheIdTags` method). This way will be able to get all benefits from invalidation system already placed in Magento if we plug into the function that purges object's cache. For example there will be no need to add a code to purge product cache if we save it in BO as Magento already does it. That's why this system was chosen for "Smile_MageCache" module. **It's very important to understand this point.**

Default tags system gives quite good results but it has several limitations:

- Some objects do not dispose any cache tag (example: cache depends on some configuration value).
- In some cases we do not know which object may be used in a page in the future. Good example: collections that use custom filter. If some product that matches the filter was added in BO we can not link it to the page. This problem is quite common for search pages for example.
- Code that loads an object may be cached in Magento.
- Some pages may use a lot of objects and may probably need some custom invalidation strategy.
- Some object may be used in every page. For example items used to render top menu of the site (top level categories, CMS pages, blocs, etc). If we change one of these items we'll flush everything (This particular problem may be solved using ESI system)

That's why it stays possible to tune this system and complete it with custom tags.

Custom tags

General idea

In order to put custom tag you just need to put following line in your code:

```
Mage::helper('smile_magecache')->addTags('MY_CUSTOM_TAG');
```

You may use this function to put several tags as well:

```
Mage::helper('smile_magecache')->addTags(array('MY_TAG_1, MY_TAG_2'));
```

Once this is done you should manage yourself the invalidation of this tag. As we saw in previous section "Smile_MageCache" module is plugged to general purge functionality. So to clean resources related to our custom tag you need to use correspondent Magento function

```
Mage::app()->cleanCache(array('MY_TAG_2'));
```

This code may be placed everywhere. You may create a special action in BO or put it to some observer to make purge process transparent.

Pay attention that is important to use native Magento function to clean the cache and not direct call to "Smile_MageCache" processor. This way you can reuse your custom tags for other application elements (like HTML blocks for example) without thinking about cache invalidation – it is already done.

Exceptional blocks

In a lot of situations custom tags may be used in order to change basic tag system for some HTML block. Imagine that, you have a block that lists all categories of your site. If you use classic tag system the cache of this block will not be very persistent. Each single modification on any category will bring it to expiration. Meanwhile normally you do not expect any changes on this block unless you add a category, you delete one or you change its name. To implement such behavior “Smile_MageCache” module provide a mechanism called “exceptional block”. The idea is quite simple:

- We don't collect any tag during block rendering
- We put specific tags in a special callback function called when block is rendered
- Cache invalidation is still managed manually

To implement the example given above using this mechanism we need to:

1. Declare exceptional block and 2 observers in the configuration

```
<global>
    <events>
        <catalog_category_save_after>
            <observers>
                <clean_categories_cache>
                    <class>yourproject_pagecache/observer</class>
                    <method>cleanCategoriesCacheOnSave</method>
                </clean_categories_cache>
            </observers>
        </catalog_category_save_after>
        <catalog_category_delete_after>
            <observers>
                <clean_categories_cache>
                    <class>yourproject_pagecache/observer</class>
                    <method>cleanCategoriesCacheOnDelete</method>
                </clean_categories_cache>
            </observers>
        </catalog_category_delete_after>
    </events>
</global>
<frontend>
    <smile_magecache>
        <block_exceptions>
            <categories>
                <type>yourproject_catalog/categories</type>
                <callback>
                    <model>yourproject_pagecache/callback</model>
                    <method>categories</method>
                </callback>
            </categories>
        </block_exceptions>
    </smile_magecache>
</frontend>
```

2. Implement the callback

```
/**
 * Put cache tag for categories block
 *
 * @param YourProject_Catalog_Block_Categories $block categories block
 *
 * @return void
 */
public function categories(YourProject_Catalog_Block_Categories $block)
{
    Mage::helper('smile_magecache')->addTags('CATEGORIES_BLOCK');
}
```

3. Implement observers

```
/**
 * Clean categories block cache when category is saved
 *
 * @param Varien_Event_Observer $observer observer
 *
 * @return void
 */
public function cleanCategoriesCacheOnSave(Varien_Event_Observer $observer)
{
    $category = $observer->getEvent()->getDataObject();
    if ($category->getData('name') != $category->getOrigData('name')) {
        Mage::app()->cleanCache(array('CATEGORIES_BLOCK'));
    }
}

/**
 * Clean categories block cache when category is deleted
 *
 * @param Varien_Event_Observer $observer observer
 *
 * @return void
 */
public function cleanCategoriesCacheOnDelete(
    Varien_Event_Observer $observer
) {
    Mage::app()->cleanCache(array('CATEGORIES_BLOCK'));
}
```

Shortcuts

Another disadvantage of using native Magento tags is their size. Classic Magento tags may be quite long and storing them will take some place. That does not make a huge difference for a couple of tags but it will do it for dozens entities. In order to avoid this “Smile_MageCache” implements a shortcuts system. These shortcuts are applied automatically before page cache processor sends a request to engine, so you continue to use “full” versions in your code. There are 4 default shortcuts in the module

```
<smile_magecache>
    <cache_tag_shortcuts>
```

```
        <catalog_category>
            <source>CATALOG_CATEGORY</source>
            <target>CAT</target>
        </catalog_category>
        <catalog_product>
            <source>CATALOG_PRODUCT</source>
            <target>PRO</target>
        </catalog_product>
        <cms_block>
            <source>CMS_BLOCK</source>
            <target>BLO</target>
        </cms_block>
        <cms_page>
            <source>CMS_PAGE</source>
            <target>PAG</target>
        </cms_page>
    </cache_tag_shortcuts>
</smile_magecache>
```

You're always able to declare custom shortcuts if necessary.

4. Engines

This chapter describes how "Smile_MageCache" interact with different cache engines, which engine connectors are available by default a how to integrate custom cache engine to the system.

Basics

In order to integrate a cache engine we should be able to implement 3 basic actions:

- Verify that current page is cacheable and calculate its cache key
- Store page with its cache tags
- Purge cache by given cache tags

Note that part of these functions may be integrated at engine's configuration level and not by the module.

By default "Smile_MageCache" module comes with 2 engines:

- Varnish
- Default Application Cache

Note that you can use one or several engines at the same time.

Let's see how it works.

Varnish

Normally if we use Varnish cache for some project we already have cache rules configured in VCL file. In order to finalize the integration we need to found how to store cache tags and to how to manage cache invalidation.

In order to tag some page in Varnish we add the list of collected tags as a specific HTTP header. It gives 3 major benefits:

- Varnish keeps in cache all response headers so we don't need to use 3rd party service to store page-tag relations
- Varnish gives a possibility to purge resources using regular expression on different parts of stored objects (including HTTP headers)
- Varnish processes each object (even a cached one) before send the response. This way can remove our specific header making implementation transparent for final user

In our case we use "X-Cache-Tags" for header name and "~" symbol to concatenate all tags together. Example:

```
ivan.shcherbakov@localhost:~$ curl -I http://project.com
HTTP/1.1 200 OK
…
X-Cache-Tags: ~PAG_49~PAG_133~PAG_11~PAG_121~CAT_3~CAT_1863~
…
```

To invalidate all pages related to "CAT_3" tag we have to launch purge command with specifying our tag:

```
obj.http.X-Cache-Tags ~ ~CAT_3~
```

This command could be launched by Varnish Administrator tool or by simple HTTP request (special rule in VCL file is required).

Note that if several Varnish instances are used each purge request must be to every single instance

"Smile_MageCache" gives a possibility to choose which purge method to use.

Varnish Administrator

This method does not require any specific configuration. You just need to make sure that Varnish Administrator is turned on (check -T parameter in daemon options) and it is accessible from each server that needs to send purge requests:

```
smile@back1.project.com:~$ telnet front1 6082
Trying 172.10.10.1...
Connected to front1.
Escape character is '^]'.
200 194
-----------------------------
Varnish Cache CLI 1.0
-----------------------------
Linux,3.2.0-38-generic,x86_64,-sfile,-hcritbit

Type 'help' for command list.
Type 'quit' to close CLI session.
```

If Varnish is started with a secret key (-s parameter) you will also need to retrieve a content of this file.

Once it's done you can activate the engine in BO configuration:

Page Cache > Varnish Engine > Enabled	Yes
Page Cache > Varnish Engine > Server	front1:6082:code1;front2:6082:code2
Page Cache > Varnish Engine > Connection Mode	Varnish administator socket

Even if this method is the recommended one sometimes it can not be used. Several hosting companies hesitate to grant access to Varnish administration tool because of security reason.

HTTP Service

This method does not require any special access but impose some changes in Varnish configuration. To make it working you will need to add following block to `vcl_recv` method of you VCL file:

```
# Smile_MageCache - Varnish connection request handler
if (req.http.X-Command && req.http.X-Secret == "secret") {
    if (req.http.X-Command == "Purge-By-Header") {
        if (req.http.X-Arg-Name && req.http.X-Arg-Value) {
            purge("obj.http."req.http.X-Arg-Name"   ~   "   req.http.X-Arg-
Value);
            error 200 "Done";
        } else {
            error 500 "Argument is missing";
        }
    } elsif (req.http.X-Command == "Purge-By-URL") {
        if (req.http.X-Arg-Pattern) {
            purge("req.url ~ " req.http.X-Arg-Pattern);
            error 200 "Done";
        } else {
            error 500 "Argument is missing";
        }
    } else {
        error 500 "Unknown command";
    }
}
```

This block implements a very basic service giving a possibility to purge resources by header taking the tags from HTTP header.

Please pay attention at "X-Secret" parameter. It's very important to customize it as in general case front servers have public IP addresses so they are accessible by everyone via HTTP. In this case secret value is the only available protection.

Once it's done you can activate engine in BO configuration:

Page Cache > Varnish Engine > Enabled	Yes
Page Cache > Varnish Engine > Server	front1:80:code1;front2:80:code2
Page Cache > Varnish Engine > Connection Mode	HTTP

You may see that this code implements additional service to purge resources by URL. We'll see later how this feature can be used.

Default Application Cache

Default application cache implemented in "Smile_MageCache" module is quite similar to "Enterprise_PageCache" module it is but much simpler. It does not handle any specific exceptions, placeholders or other stuff like that is done in Magento module. The main purpose of this layer is to accelerate page rendering in the case where several Varnish instances are used.

This layer is called application layer as it uses native Magento cache class to handle all operation. By default it will share default application cache storage but it stays possible (and recommended) to use dedicated cache connection. It can be done by declaring additional configuration in `app/etc/magecache.xml` file:

```
<smile_magecache>
    <backend_options>
        <id_prefix>PREFIX_</id_prefix>
        <backend>memcached</backend>
        <slow_backend>database</slow_backend>
        <memcached>
            <servers>
                <server>
                    <host><![CDATA[host]]></host>
                    <port><![CDATA[port]]></port>
                    <persistent><![CDATA[1]]></persistent>
                </server>
            </servers>
            <compression><![CDATA[0]]></compression>
        </memcached>
    </backend_options>
</smile_magecache>
```

Note that you can use all backend types that are compatible with Magento.

Default engine uses URL as unique identifier to construct a cache key. Each page is stored as 2 objects:

- One object for page content (HTML code is compressed)
- One object for page headers

By default this engine cache only product and category pages. You can customize it via engine's configuration:

```
<smile_magecache>
    <engines>
        <default>
            <config>
                <actions>
                    <catalog_category_view />
                    <catalog_product_view />
                </actions>
            </config>
        </default>
    </engines>
</smile_magecache>
```

If you want activate this engine in BO use following settings:

Page Cache > Default Engine > Enabled	Yes

Implement a custom engine

As it was said it is possible to plug a custom engine to the system. There are 2 steps to do it:

1. Create a special class that extends from `Smile_MageCache_Model_Engine_Abstract` and implements 2 methods to process request tags and manage invalidation
2. Plug engine into configuration. Example:

```
<smile_magecache>
    <engines>
        <cotendo>
            <model>yourproject_pagecache/engine_cotendo</model>
            <order>5</order>
            <config>
                <!-- Custom configuration may go here -->
            </config>
        </cotendo >
    </engine>
</smile_magecache>
```

Note that smallest value of "order" must correspond to the front engine when highest to the one is the "closest" to application.

5. Features

This chapter describes different features that are available in "Smile_MageCache".

Purge mode

In order to purge a cache Magento must send a purge command to each cache server (of each engine). There are 2 ways to do it:

- **Synchronous mode.** Magento collects tags during the execution of the script and sends a purge request in the end of page generation. This mode can be used with development and integration environments giving a possibility to test different use-cases. It can be also used with production environment where data modification flow is not very important.
- **Asynchronous mode.** Magento still collects cache tags but instead of making a purge it put them to the queue. A special script purge collected tags each X minutes. This approach reduces a BO load and ensures a TTL that could protect you in case of unpredicted situations. This mode is recommended for high-load production environments with a lot of data modifications.

Default mode is synchronous one. You can activate asynchronous mode in BO:

Page Cache > General > Asynchronous Cache Flush   Yes

Once that is done you will also need to put the script to the crontab of apache

```
*/10 * * * * php /var/www/project/www/shell/magecache.php
```

Ignored routes

This functionality is used to switch off tag collections for routes that are never cacheable (ex: checkout, customer area, etc.). By default there are 2 routes declared in XML

```
<smile_magecache>
    <ignored_routes>
        <customer>/customer</customer>
        <checkout>/checkout</checkout>
    </ignored_routes>
</smile_magecache>
```

You can always enrich this list if necessary.

Actions

Sometimes it's necessary to have a possibility to purge some resources just clicking a button in BO. "Smile_MageCache" module provides an interface to implement this kind of actions easily and quickly.

There 2 steps to implement an action:

1. Declare it in configuration of a module

```
<smile_magecache>
    <actions>
        <categories>yourproject_pagecache/action_categories</categories>
    </actions>
</smile_magecache>
```

2. Declare action class with action body

```
class YourProject_PageCache_Model_Action_Categories
    implements Smile_MageCache_Model_Action
{
    /**
     * Retrieve action name
     *
     * @return string
     */
    public function getLabel()
    {
        return Mage::helper('yourproject_pagecache')->__(
            'Special Categories'
        );
    }

    /**
     * Retrieve action description
     *
     * @return string
     */
    public function getDescription()
    {
        return Mage::helper('yourproject_pagecache')->__(
            'Purge special categories cache'
        );
    }

    /**
     * Run action
     *
     * @return void
     */
    public function run()
    {
        $categories = Mage::helper('some_module')->getSpecialCategories();
        foreach ($categories as $category) {
            $category->cleanModelCache();
        }
    }
}
```

Once this is done your action is available in BO under “System” > “Page Cache” menu item.

You can also use this interface to implement an action to purge some URL (like images, JS or CSS files or another data). It is possible with a feature described below.

Purge by URL

Several engines (like Varnish for example) give a possibility to purge resources by URL pattern. This feature is implemented by general connector of “Smile_MageCache” module and can be used with all engines that implement `Smile_MageCache_Model_Engine_Feature_PurgeUrl` interface. By default it is available to for Varnish engine. To use this function just call:

```
Mage::getSingleton('smile_magecache/processor')-> purgeUrl($pattern);
```

Shell script

By default the script that we used for asynchronous purge mode cleans all cache tags stored in queue. But it is also possible to use it for purge some specific tags or URL. To do it you just need to put a special parameter when you call the script.

Purge by tags:

```
php magecache.php --purge_tags CAT_3,CATEGORIES_BLOCK
```

Purge by URL:

```
php magecache.php --purge_url skin
```

This feature can be useful in order to purge some resources when we make a delivery.

Debug

This feature gives a possibility to log purge requests in order to debug possible connection problems. Each engine is responsible to log its own requests. By default debug function is available for Varnish engine only. Requests are logged to `var/log/varnish.log` file.

To activate debug feature in BO please set:

Page Cache > General > Debug	Yes

Log messages that correspond to purge requests above look like this:

```
2013-03-19T10:32:59+00:00 INFO (6): [localhost] X-Command: Purge-By-Header;
X-Secret: secret; X-Arg-Name: X-Cache-Tags; X-Arg-Value:
~CAT_3~|~CATEGORIES_BLOCK~
2013-03-19T10:32:59+00:00 INFO (6): OK
2013-03-19T10:33:29+00:00 INFO (6): [localhost] X-Command: Purge-By-URL; X-
Secret: secret; X-Arg-Pattern: skin
2013-03-19T10:33:30+00:00 INFO (6): OK
```

You may see that HTTP mode was chosen in this case

6. Other points

Varnish Library

As most features of Varnish connector are quite generic and are not related to Magento core they were implemented as an independent library. You can find it under `lib/Varnish` folder. This library has all necessary functions to purge the cache, log requests, etc. It can be reused for any PHP application.

Usage example:

```
$connector = new Varnish_Connector();
$servers = array(
    array('host' => 'localhost', 'port' => 80, 'secret' => 'chubaka')
);
$connector->setConnectionType('http');
$connector->init($servers);
$connector->purgeByResponseHeader('X-Cache-Tags', '~CAT_3~|~CAT_5~');
$connector->purgeByUrl('skin');
```

Plug a logger:

```
class MyLogger implements Varnish_Connector_Logger
{
    /**
     * Put a log message
     *
     * @param string $message message text
     * @param int    $level   log level
     *
     * @return void
     */
    public function log($message, $level = null)
    {
        $filename = '/var/log/varnish-purge.log';
        $data = '['.$level.']'.$message."\n";
        file_put_contents($filename, $data,  FILE_APPEND);
    }
}

$logger = new MyLogger();
Varnish_Connector::setLogger($logger);
```

THE END