HTTP Cache

Introduction

Smart HTTP cache clearing

Smart HTTP cache clearing refers to the ability to clear cache for Locations/content that is in relation with the content being currently cleared.

When published, any Content item usually has at least one Location, identified by its URL. Therefore, HTTP cache being bound to URLs, if a Content item is updated (a new version is published), we want HTTP cache for all its Locations to be cleared, so the content itself can be updated everywhere it is supposed to be displayed. Sometimes, clearing cache for the content's Locations is not sufficient. You can, for instance, have an excerpt of it displayed in a list from the parent Location, or from within a relation. In this case, cache for the parent Location and/or the relation need to be cleared as well (at least if an ESI is not used).

The mechanism

Smart HTTP cache clearing is an event-based mechanism. Whenever a content item needs its cache cleared, the cache purger service sends an `ezpublish.cache_clear.content` event (also identified by `eZ\Publish\Core\MVC\Symfony\MVCEvents::CACHE_CLEAR_CONTENT` constant) and passes an `eZ\Publish\Core\MVC\Symfony\Event\ContentCacheClearEvent` event object. This object contains the ContentInfo object we need to clear the cache for. Every listener for this event can add Location objects to the *cache clear list*.

Once the event is dispatched, the purger passes collected Location objects to the purge client, which will effectively send the cache `BAN` request.

Note
The event is dispatched with a dedicated event dispatcher, `ezpublish.http_cache.event_dispatcher`.

Default behavior

By default, following Locations will be added to the cache clear list:

- All Locations assigned to content (`AssignedLocationsListener`)
- Parent Location of all Content item's Locations (`ParentLocationsListener`)
- Locations for content's relations, including reverse relations (`RelatedLocationsListener`)

Implementing a custom listener

By design, smart HTTP cache clearing is extensible. One can easily implement an event listener/subscriber to the `ezpublish.cache_clear.content` event and add Locations to the cache clear list.

Example

Here's a very simple custom listener example, adding an arbitrary Location to the list.

Important
Cache clear listener services **must** be tagged as `ezpublish.http_cache.event_subscriber` or `ezpublish.http_cache.event_listener`.

In this topic:

Related:

```
namespace Acme\AcmeTestBundle\EventListener;

use eZ\Publish\API\Repository\LocationService;
use
eZ\Publish\Core\MVC\Symfony\Event\ContentCacheClearEvent
;
use eZ\Publish\Core\MVC\Symfony\MVCEvents;
use
Symfony\Component\EventDispatcher\EventSubscriberInterfa
ce;

class ArbitraryLocationsListener implements
EventSubscriberInterface
{
    /**
     * @var LocationService
     */
    private $locationService;

    public function __construct( LocationService
$locationService )
    {
        $this->locationService = $locationService;
    }

    public static function getSubscribedEvents()
    {
        return [MVCEvents::CACHE_CLEAR_CONTENT =>
['onContentCacheClear', 100]];
    }

    public function onContentCacheClear(
ContentCacheClearEvent $event )
    {
        // $contentInfo is the ContentInfo object for
the content being cleared.
        // You can extract information from it (e.g.
ContentType from its contentTypeId), using appropriate
Repository services.
        $contentInfo = $event->getContentInfo();

        // Adding arbitrary locations to the cache clear
list.
        $event->addLocationToClear(
$this->locationService->loadLocation( 123 ) );
        $event->addLocationToClear(
$this->locationService->loadLocation( 456 ) );
    }
}
```

```
parameters:
    acme.cache_clear.arbitrary_locations_listener.class:
Acme\AcmeTestBundle\EventListener\ArbitraryLocationsList
ener

services:
    acme.cache_clear.arbitrary_locations_listener:
        class:
%acme.cache_clear.arbitrary_locations_listener.class%
        arguments: [@ezpublish.api.service.location]
        tags:
            - { name:
ezpublish.http_cache.event_subscriber }
```

Content Cache

eZ Platform uses Symfony HttpCache to manage content "view" cache with an expiration model. In addition it is extended *(using FOSHttpCache)* to add several advanced features. For content coming from the CMS the following is taken advantage of out of the box:

- To be able to always keep cache up to date, cache is made "content-aware" to allow updates to content to trigger *cache invalidation.*
    - Uses a custom `X-Location-Id` header, which both Symfony and Varnish Proxy are able to invalidate cache on *(for details see Cache purge.)*
- To be able to also cache requests by logged-in users, cache is made "context-aware ."
    - Uses a custom vary header `X-User-Hash` to allow pages to var by user rights *(so not per unique user, that is better served by browser cache.)*

Cache and Expiration Configuration

ezplatform.yml

```
ezpublish:
    system:
        my_siteaccess:
            content:
                view_cache: true      # Activates
HttpCache for content
                ttl_cache: true       # Activates
expiration based HttpCache for content (very fast)
                default_ttl: 60       # Number of
seconds an Http response is valid in cache (if ttl_cache
is true)
```

Making your controller response content-aware

Sometimes you need your controller's cache to be invalidated at the same time as specific content changes (i.e. ESI sub-requests with `render` twig helper, for a menu for instance). To be able to do that, you just need to add **`X-Location-Id`** header to the response object:

```
use Symfony\Component\HttpFoundation\Response;

// Inside a controller action
// "Connects" the response to location #123 and sets a
max age (TTL) of 1 hour.
$response = new Response();
$response->headers->set('X-Location-Id', 123);
$response->setSharedMaxAge(3600);
```

Making your controller response context-aware

If the content you're rendering depends on a user's permissions, then you should make the response context-aware:

```
use Symfony\Component\HttpFoundation\Response;

// Inside a controller action
// Tells proxy configured to support this header to take
the rights of a user (user hash) into account for the
cache
$response = new Response();
$response->setVary('X-User-Hash');
```

Configuration

Cache Purge

This page explains the content cache purge *(aka invalidate)* mechanism used when publishing content from the UI or from a container-aware script, resulting in cache being invalidated either in the built-in Symfony Reverse Proxy, or on the much faster Varnish reverse proxy.

Overview

eZ Platform returns content-related responses with an `X-Location-Id` header that are stored together by the configured HTTP cache. This allows you to clear *(invalidate)* HTTP cache representing specifically a given Content item. On publishing the content, a cache purger is triggered with the Content ID in question, which in turn figures out affected content Locations based on Smart HTTP cache clearing logic. The returned Location IDs are sent for purge using the purge type explained further below.

Purge types

Symfony Proxy: Local purge type

By default, invalidation requests will be emulated and sent to the Symfony Proxy cache Store. Cache purge will thus be synchronous, meaning no HTTP purge requests will be sent around when publishing.

ezplatform.yml

```
ezpublish:
    http_cache:
        purge_type: local
```

Varnish: HTTP purge type

With Varnish you can configure one or several servers that should be purged over HTTP. This purge type is asynchronous, and flushed by the end of Symfony kernel-request/console cycle *(during terminate event)*. Settings for purge servers can be configured per site group or site access:

ezplatform.yml

```
ezpublish:
    http_cache:
        purge_type: http

    system:
        my_siteacess:
            http_cache:
                purge_servers:
["http://varnish.server1", "http://varnish.server2",
"http://varnish.server3"]
```

For further information on setting up Varnish, see Using Varnish.

Purging

While purging on content, updates are handled for you; on actions against the eZ Platform APIs, there are times you might have to purge manually.

Manual by code

Manual purging from code which takes Smart HTTP cache clearing logic into account, this is using the service also used internally for cache clearing on content updates:

```
// Purging cache based on content id, this will trigger
cache clear of all locations found by Smart HttpCache
clear
// typically self, parent, related, ..
$container->get('ezpublish.http_cache.purger')->purgeFor
Content(55);
```

Manually by command with Symfony Proxy

Symfony Proxy stores its cache in the Symfony cache directory, so a regular `cache:clear` commands will clear it:

```
php app/console --env=prod cache:clear
```

Manual by HTTP BAN request on Varnish

When using Varnish and in need to purge content directly, then the following examples show how this is done internally by our FOSPurgeClient, and in turn FOSHttpCache Varnish proxy client:

For purging all:

```
BAN / HTTP 1.1
Host: localhost
X-Location-Id: .*
```

Or with given location ids *(here 123 and 234)*:

```
BAN / HTTP 1.1
Host: localhost
X-Location-Id: ^(123|234)$
```

Using Varnish

eZ Platform being built on top of Symfony, it uses standard HTTP cache headers. By default the Symfony reverse proxy, written in PHP, is used to handle cache, but it can be easily replaced with any other reverse proxy like Varnish.

Use of Varnish is a requirement for use in Clustering setup, for overview of clustering feature see C lustering.

Prerequisites

- A working Varnish 3 or Varnish 4 setup.

Recommended VCL base files

For Varnish to work properly with eZ, you'll need to use one of the provided files as a basis:

- Varnish 3 VCL example
- Varnish 4 VCL example

> ***Note:*** *Http cache management is done with the help of FOSHttpCacheBundle.*
> *You may need to tweak your VCL further on according to FOSHttpCache documentation in order to use features supported by it.*

Configure eZ Publish

Update your Virtual Host

Somehow we need to tell php process that we are behind a Varnish proxy and not the built in Symfony Http Proxy. If you use fastcgi/fpm you can pass these directly to php process, but you can in all cases also specify them in your web server config.

On apache:

my_virtualhost.conf

```
<VirthualHost *:80>
    # Configure your VirtualHost with rewrite rules and
stuff

    # Force front controller NOT to use built-in reverse
proxy.
    SetEnv SYMFONY_HTTP_CACHE 0

 # Configure IP of your Varnish server to be trusted
proxy
    # Replace fake IP address below by your Varnish IP
address
    SetEnv SYMFONY_TRUSTED_PROXIES "193.22.44.22"
</VirtualHost>
```

On nginx:

mysite.com

```
fastcgi_param SYMFONY_HTTP_CACHE 0;
# Configure IP of your Varnish server to be trusted
proxy
# Replace fake IP address below by your Varnish IP
address
fastcgi_param SYMFONY_TRUSTED_PROXIES "193.22.44.22";
```

Update YML configuration

Secondly we need to tell eZ Platform to change to use http based purge client *(specifically FosHttpCache Varnish purge client is used)*, and specify url Varnish can be reached on:

ezplatform.yml

```
ezpublish:
    http_cache:
        purge_type: http

    system:
        # Assuming that my_siteaccess_group your
frontend AND backend siteaccesses
        my_siteaccess_group:
            http_cache:
                # Fill in your Varnish server(s)
address(es).
                purge_servers:
[http://my.varnish.server:8081]
```

Usage

Context-aware HTTP cache

Use case

As it is based on Symfony 2, eZ Platform uses HTTP cache extended with features like content awareness. However, this cache management is only available for anonymous users due to HTTP restrictions.

It is of course possible to make HTTP cache vary thanks to the `Vary` response header, but this header can only be based on one of the request headers (e.g. `Accept-Encoding`). Thus, to make the cache vary on a specific context *(for example a hash based on a user roles and limitations)*, this context must be present in the original request.

Feature

As the response can vary on a request header, the base solution is to make the kernel do a sub-request in order to retrieve the user context hash (aka **user hash**). Once the *user hash* has been retrieved, it's injected in the original request in the `X-User-Hash` custom header, making it possible to *vary* the HTTP response on this header:

```
<?php
use Symfony\Component\HttpFoundation\Response;

// ...

// Inside a controller action
$response = new Response();
$response->setVary('X-User-Hash');
```

This solution is implemented in Symfony reverse proxy (aka *HttpCache*) and is also accessible to dedicated reverse proxies like Varnish.

Note that sharing ESIs across SiteAccesses is not possible by design (see
EZP-22535 - Cached ESI can not be shared across pages/siteaccesses due to "pathinfo" property **CLOSED**
for technical details)

Vary by User
In cases where you need to deliver content uniquely to a given user, and tricks like using JavaScript and cookie values, hinclude, or disabling cache is not an option. Then remaining option is to vary response by cookie:

```
$response->setVary('Cookie');
```

Unfortunately this is not optimal as it will by default vary by all cookies, including those set by add trackers, analytics tools, recommendation services, etc. However, as long as *your* application backend does not need these cookies, you can solve this by stripping everything but the session cookie. Example for Varnish can be found in the default VCL examples in part dealing with User Hash, for single-server setup this can easily be accomplished in Apache / Nginx as well.

HTTP cache clear

As eZ Platform uses FOSHttpCacheBundle, this impacts the following features:

- HTTP cache purge
- User context hash

Varnish proxy client from FOSHttpCache lib is used for clearing eZ HTTP cache, even when using Symfony HttpCache. A single `BAN` request is sent to registered purge servers, containing a `X-Location-Id` header. This header contains all Location IDs for which objects in cache need to be cleared.

Symfony reverse proxy

Symfony reverse proxy (aka HttpCache) is supported out of the box, all you have to do is to activate it.

Varnish

Please refer to Using Varnish

User context hash

FOSHttpCacheBundle *User Context feature* is activated by default.

As the response can vary on a request header, the base solution is to make the kernel do a sub-request in order to retrieve the context (aka **user context hash**). Once the *user hash* has been retrieved, it's injected in the original request in the `X-User-Hash` header, making it possible to *vary* the HTTP response on this header:

Name of the user hash header is configurable in FOSHttpCacheBundle. By default eZ Platform sets it to `**X-User-Hash**`.

```
<?php
use Symfony\Component\HttpFoundation\Response;

// ...

// Inside a controller action
$response = new Response();
$response->setVary('X-User-Hash');
```

This solution is implemented in Symfony reverse proxy (aka *HttpCache*) and is also accessible to dedicated reverse proxies like Varnish.

Workflow

Please refer to FOSHttpCacheBundle documentation on how user context feature works.

User hash generation

Please refer to FOSHttpCacheBundle documentation on how user hashes are being generated.

eZ Platform already interferes with the hash generation process by adding current user permissions and limitations. You can also interfere in this process by implementing custom context provider(s).

User hash generation with Varnish 3

The behavior described here comes out of the box with Symfony reverse proxy, but it's of course possible to use Varnish to achieve the same.

```
# Varnish 3 style for eZ Platform
# Our Backend - We assume that eZ Platform Web server
listen on port 80 on the same machine.
backend ezplatform {
    .host = "127.0.0.1";
    .port = "80";
}

# Called at the beginning of a request, after the
complete request has been received
sub vcl_recv {

    # Set the backend
    set req.backend = ezplatform;

    # ...

    # Retrieve client user hash and add it to the
forwarded request.
    call ez_user_hash;

    # If it passes all these tests, do a lookup anyway;
    return (lookup);
}

# Sub-routine to get client user hash, for context-aware
HTTP cache.
# Don't forget to correctly set the backend host for the
Curl sub-request.
sub ez_user_hash {

    # Prevent tampering attacks on the hash mechanism
    if (req.restarts == 0
        && (req.http.accept ~
"application/vnd.fos.user-context-hash"
            || req.http.x-user-context-hash
        )
    ) {
        error 400;
    }

    if (req.restarts == 0 && (req.request == "GET" ||
req.request == "HEAD")) {
        # Get User (Context) hash, for varying cache by
what user has access to.
        #
https://doc.ez.no/display/EZP/Context+aware+HTTP+cach
```

```
        # Anonymous user w/o session => Use hardcoded
anonymous hash to avoid backend lookup for hash
        if (req.http.Cookie !~ "eZSESSID" &&
!req.http.authorization) {
            # You may update this hash with the actual
one for anonymous user
            # to get a better cache hit ratio across
anonymous users.
            # Note: Then needs update every time
anonymous user role assignments change.
            set req.http.X-User-Hash =
"38015b703d82206ebc01d17a39c727e5";
        }
        # Pre-authenticate request to get shared cache,
even when authenticated
        else {
            set req.http.x-fos-original-url    =
req.url;
            set req.http.x-fos-original-accept =
req.http.accept;
            set req.http.x-fos-original-cookie =
req.http.cookie;
            # Clean up cookie for the hash request to
only keep session cookie, as hash cache will vary on
cookie.
            set req.http.cookie = ";" + req.http.cookie;
            set req.http.cookie =
regsuball(req.http.cookie, "; +", ";");
            set req.http.cookie =
regsuball(req.http.cookie, ";(eZSESSID[^=]*)=", ";
\1=");
            set req.http.cookie =
regsuball(req.http.cookie, ";[^ ][^;]*", "");
            set req.http.cookie =
regsuball(req.http.cookie, "^[; ]+|[; ]+$", "");

            set req.http.accept =
"application/vnd.fos.user-context-hash";
            set req.url = "/_fos_user_context_hash";

            # Force the lookup, the backend must tell
not to cache or vary on all
            # headers that are used to build the hash.

            return (lookup);
        }
    }

    # Rebuild the original request which now has the
hash.
    if (req.restarts > 0
        && req.http.accept ==
"application/vnd.fos.user-context-hash"
    ) {
        set req.url         =
req.http.x-fos-original-url;
        set req.http.accept =
req.http.x-fos-original-accept;
```

```
        set req.http.cookie =
req.http.x-fos-original-cookie;

        unset req.http.x-fos-original-url;
        unset req.http.x-fos-original-accept;
        unset req.http.x-fos-original-cookie;

        # Force the lookup, the backend must tell not to
cache or vary on the
        # user hash to properly separate cached data.

        return (lookup);
    }
}

sub vcl_fetch {

    # ...

    if (req.restarts == 0
        && req.http.accept ~
"application/vnd.fos.user-context-hash"
        && beresp.status >= 500
    ) {
        error 503 "Hash error";
    }
}

sub vcl_deliver {
    # On receiving the hash response, copy the hash
header to the original
    # request and restart.
    if (req.restarts == 0
        && resp.http.content-type ~
"application/vnd.fos.user-context-hash"
        && resp.status == 200
    ) {
        set req.http.x-user-hash =
resp.http.x-user-hash;

        return (restart);
    }

    # If we get here, this is a real response that gets
sent to the client.

    # Remove the vary on context user hash, this is
nothing public. Keep all
    # other vary headers.
    set resp.http.Vary = regsub(resp.http.Vary, "(?i),?
*x-user-hash *", "");
    set resp.http.Vary = regsub(resp.http.Vary, "^, *",
"");
    if (resp.http.Vary == "") {
        remove resp.http.Vary;
    }

    # Sanity check to prevent ever exposing the hash to
a client.
```

```
    remove resp.http.x-user-hash;
}
```

User hash generation with Varnish 4

```
// Varnish 4 style for eZ Platform
// Complete VCL example

vcl 4.0;

// Our Backend - Assuming that web server is listening
on port 80
// Replace the host to fit your setup
backend ezplatform {
    .host = "127.0.0.1";
    .port = "80";
}

// Called at the beginning of a request, after the
complete request has been received
sub vcl_recv {

    // Set the backend
    set req.backend_hint = ezplatform;

    // ...

    // Retrieve client user hash and add it to the
forwarded request.
    call ez_user_hash;

    // If it passes all these tests, do a lookup anyway.
    return (hash);
}

// Called when the requested object has been retrieved
from the backend
sub vcl_backend_response {
    if (bereq.http.accept ~
"application/vnd.fos.user-context-hash"
        && beresp.status >= 500
    ) {
        return (abandon);
    }

    // ...
}

// Sub-routine to get client user hash, for
context-aware HTTP cache.
sub ez_user_hash {

    // Prevent tampering attacks on the hash mechanism
    if (req.restarts == 0
        && (req.http.accept ~
"application/vnd.fos.user-context-hash"
            || req.http.x-user-hash
        )
    ) {
        return (synth(400));
    }

    if (req.restarts == 0 && (req.method == "GET" ||
req.method == "HEAD")) {
        // Get User (Context) hash, for varying cache by
what user has access to.
        //
https://doc.ez.no/display/EZP/Context+aware+HTTP+cache

        // Anonymous user w/o session => Use hardcoded
anonymous hash to avoid backend lookup for hash
        if (req.http.Cookie !~ "eZSESSID" &&
!req.http.authorization) {
            // You may update this hash with the actual
one for anonymous user
            // to get a better cache hit ratio across
anonymous users.
            // Note: You should then update it every
time anonymous user rights change.
            set req.http.X-User-Hash =
"38015b703d82206ebc01d17a39c727e5";
        }
        // Pre-authenticate request to get shared cache,
even when authenticated
        else {
            set req.http.x-fos-original-url    =
req.url;
            set req.http.x-fos-original-accept =
req.http.accept;
            set req.http.x-fos-original-cookie =
req.http.cookie;
            // Clean up cookie for the hash request to
only keep session cookie, as hash cache will vary on
cookie.
            set req.http.cookie = ";" + req.http.cookie;
            set req.http.cookie =
regsuball(req.http.cookie, "; +", ";");
            set req.http.cookie =
regsuball(req.http.cookie, ";(eZSESSID[^=]*)=", ";
\1=");
            set req.http.cookie =
regsuball(req.http.cookie, ";[^ ][^;]*", "");
            set req.http.cookie =
regsuball(req.http.cookie, "^[; ]+|[; ]+$", "");
            set req.http.accept =
"application/vnd.fos.user-context-hash";
            set req.url = "/_fos_user_context_hash";

            // Force the lookup, the backend must tell
how to cache/vary response containing the user hash
            return (hash);
        }
    }
    // Rebuild the original request which now has the
hash.
    if (req.restarts > 0
        && req.http.accept ==
"application/vnd.fos.user-context-hash"
    ) {
        set req.url         =
req.http.x-fos-original-url;
        set req.http.accept =
req.http.x-fos-original-accept;
        set req.http.cookie =
req.http.x-fos-original-cookie;
        unset req.http.x-fos-original-url;
        unset req.http.x-fos-original-accept;
        unset req.http.x-fos-original-cookie;

        // Force the lookup, the backend must tell not
to cache or vary on the
        // user hash to properly separate cached data.

        return (hash);
    }
}

sub vcl_deliver {
    // On receiving the hash response, copy the hash
header to the original
    // request and restart.
    if (req.restarts == 0
        && resp.http.content-type ~
"application/vnd.fos.user-context-hash"
    ) {
        set req.http.x-user-hash =
resp.http.x-user-hash;
        return (restart);
    }

    // If we get here, this is a real response that gets
sent to the client.
    // Remove the vary on context user hash, this is
nothing public. Keep all
    // other vary headers.
    set resp.http.Vary = regsub(resp.http.Vary, "(?i),?
*x-user-hash *", "");
    set resp.http.Vary = regsub(resp.http.Vary, "^, *",
"");
    if (resp.http.Vary == "") {
        unset resp.http.Vary;
    }

    // Sanity check to prevent ever exposing the hash to
a client.
    unset resp.http.x-user-hash;
    if (client.ip ~ debuggers) {
        if (obj.hits > 0) {
            set resp.http.X-Cache = "HIT";
            set resp.http.X-Cache-Hits = obj.hits;
        } else {
            set resp.http.X-Cache = "MISS";
        }
    }
}
```

New anonymous X-User-Hash

The anonymous X-User-Hash is generated based on the *anonymous user*, *group* and *role*. The `38015b703d82206ebc01d17a39c727e5` hash that is provided in the code above will work only when these three variables are left unchanged. Once you change the default permissions and settings, the X-User-Hash will change and Varnish won't be able to effectively handle cache anymore.

In that case you need to find out the new anonymous X-User-Hash and change the VCL accordingly, else Varnish will return a no-cache header.

The easiest way to find the new hash is:

1. Connect to your server (*shh* should be enough)

2. Add <`your-domain.com`> to your `/etc/hosts` file

3. Execute the following command:

```
curl -I -H "Accept: application/vnd.fos.user-context-hash" http://<your-domain.com>/_fos_user_context_hash
```

You should get a result like this:

```
HTTP/1.1 200 OK
Date: Mon, 03 Oct 2016 15:34:08 GMT
Server: Apache/2.4.18 (Ubuntu)
X-Powered-By: PHP/7.0.8-0ubuntu0.16.04.2
X-User-Hash:
b1731d46b0e7a375a5b024e950fdb8d49dd25af85a5c7dd5116ad2a18cda82cb
Cache-Control: max-age=600, public
Vary: Cookie,Authorization
Content-Type: application/vnd.fos.user-context-hash
```

4. Now, replace the existing X-User-Hash value with the new one:

```
# Note: This needs update every time anonymous user role
assignments change.
set req.http.X-User-Hash =
"b1731d46b0e7a375a5b024e950fdb8d49dd25af85a5c7dd5116ad2a18cda82cb";
```

5. Restart the Varnish server and everything should work fine.

Default options for FOSHttpCacheBundle defined in eZ

The following configuration is defined in eZ by default for FOSHttpCacheBundle. You may override these settings.

```
fos_http_cache:
    proxy_client:
        # "varnish" is used, even when using Symfony
HttpCache.
        default: varnish
        varnish:
            # Means http_cache.purge_servers defined for
current SiteAccess.
            servers: [$http_cache.purge_servers$]

    user_context:
        enabled: true
        # User context hash is cached during 10min
        hash_cache_ttl: 600
        user_hash_header: X-User-Hash
```

Credits

This feature is based on Context aware HTTP caching post by asm89.