FOSHttpCache Documentation

Release 1.0.0

David Buchmann, David de Boer

Sep 22, 2017

Contents

This is the documentation for the FOSHttpCache library.

Note: This documentation is for the 2.* version of the library. For the 1.* version, please refer to the 1.4 documentation.

This library integrates your PHP applications with HTTP caching proxies such as Varnish, NGINX or the Symfony HttpCache class. Use this library to send invalidation requests from your application to the proxy server and to test your caching and invalidation setup.

If you use the Symfony full stack framework, have a look at the FOSHttpCacheBundle. The bundle provides the Invalidator as a service, support for the built-in cache kernel of Symfony and a number of Symfony-specific features to help with caching and caching proxies.

Contents:

CHAPTER 1

Getting started

Installation

The FOSHttpCache library is available on Packagist. You can install the library and its dependencies using Composer.

The library relies on HTTPlug for sending invalidation requests over HTTP, so you need to install an HTTPlug-compatible client or adapter first:

```
$ composer require php-http/guzzle6-adapter
```

You also need a PSR-7 message implementation. If you use Guzzle 6, Guzzle's implementation is already included. If you use another client, you need to install one of the message implementations. Recommended:

```
$ composer require guzzlehttp/psr7
```

Alternatively:

```
$ composer require zendframework/zend-diactoros
```

Then install the FOSHttpCache library itself:

```
$ composer require friendsofsymfony/http-cache
```

Note: This library follows Semantic Versioning. Except for major versions, we aim to not introduce BC breaks in new releases.

Configuration

There are three things you need to do to get started:

1. *configure your proxy server*

2. *set up a client for your proxy server*
3. *set up the cache invalidator*

Overview

This library mainly consists of:

- low-level clients for communicating with a proxy server (Varnish, NGINX and Symfony HttpCache)
- a cache invalidator that acts as an abstraction layer for the proxy client
- test classes that you can use for integration testing your application against a proxy server.

Measures have been taken to minimize the performance impact of sending invalidation requests:

- Requests are not sent immediately, but aggregated to be sent in parallel.
- You can determine when the requests should be sent. For optimal performance, do so after the response has been sent to the client.

CHAPTER 2

An Introduction to Cache Invalidation

This general introduction explains cache invalidation concepts. If you are already familiar with cache invalidation, you may wish to skip this chapter.

HTTP Caching Terminology

Client The client that requests web representations of the application data. This client can be visitor of a website, or for instance a client that fetches data from a REST API.

Application Also *backend application* or *origin server*. The web application that holds the data.

Proxy Server Also reverse caching proxy. Examples: Varnish, NGINX, Symfony `HttpCache`.

Time to live (TTL) Maximum lifetime of some content. Expressed in either an expiry date for the content (the `Expires:` header) or its maximum age (the `max-age` and `s-maxage` cache control directives).

Invalidation Invalidating a piece of content means telling the proxy server to no longer serve that content to clients. The proxy can choose to either discard the content immediately, or do so when it is next requested. On that next request, the proxy will fetch a fresh copy from the application.

What is Cache Invalidation?

> There are only two hard things in Computer Science: cache invalidation and naming things.
>
> – *Phil Karlton*

The problem

HTTP caching is a great solution for improving the performance of your web application. For lower load on the application and fastest response time, you want to cache content for a long period. But at the same time, you want your clients to see fresh content as soon as there is an update.

Instead of finding some compromise, you can have both with cache invalidation. When application data changes, the application takes care of invalidating its web representation as out-of-date. Although caching proxies may handle invalidation differently, the effect is always the same: the next time a client requests the data, he or she gets a new version instead of the outdated one.

Alternatives

There are three alternatives to cache invalidation.

1. The first is to *expire* your cached content quickly by reducing its time to live (TTL). However, short TTLs cause a higher load on the application because content must be fetched from it more often. Moreover, reduced TTL does not guarantee that clients will have fresh content, especially if the content changes very rapidly as a result of client interactions with the application.
2. The second alternative is to *validate* the freshness of cached content at every request. Again, this means more load on your application, even if you return early (for instance by using HEAD requests).
3. The last resort is to *not cache* volatile content at all. While this guarantees the user always sees changes without delay, it obviously increases your application load even more.

Cache invalidation gives you the best of both worlds: you can have very long TTLs, so when content changes little, it can be served from the cache because no requests to your application are required. At the same time, when data does change, that change is reflected without delay in the web representations.

Disadvantages

Cache invalidation has two possible downsides:

- Invalidating cached web representations when their underlying data changes can be very simple. For instance, invalidate `/articles/123` when article 123 is updated. However, data usually is represented not in one but in multiple representations. Article 123 could also be represented on the articles index (`/articles`), the list of articles in the current year (`/articles/current`) and in search results (`/search?name=123`). In this case, when article 123 is changed, a lot more is involved in invalidating all of its representations. In other words, invalidation adds a layer of complexity to your application. This library tries to help reduce complexity, for instance by *tagging* cached content. Additionally, if you use Symfony, we recommend you use the FOSHttpCacheBundle. which provides additional functionality to make invalidation easier.
- Invalidation is done through requests to your proxy server. Sending these requests could negatively influence performance, in particular if the client has to wait for them. This library resolves this issue by optimizing the way invalidation requests are sent.

Invalidation Methods

Cached content can be invalidated in three ways. Not all caching proxies support all methods, please refer to proxy specific documentation for the details.

Purge Purge removes content from the proxy server immediately. The next time a client requests the URL, data is fetched from the application, stored in the proxy server, and returned to the client.

A purge removes all *variants* of the cached content, as per the `Vary` header.

Refresh Fetch the requested page from the backend immediately, even if there would normally be a cache hit. The content is not just deleted from the cache, but is replaced with a new version fetched from the application.

As fetching is done with the parameters of the refresh request, other variants of the same content will not be touched.

Ban Unlike purge, ban does not remove the content from the cache immediately. Instead, a reference to the content is added to a blacklist (or ban list). Every client request is checked against this blacklist. If the request happens to match blacklisted content, fresh content is fetched from the application, stored in the proxy server and returned to the client.

Bans cannot remove content from cache immediately because that would require going through all cached content, which could take a long time and reduce performance of the cache.

The ban solution may seem cumbersome, but offers more powerful cache invalidation, such as selecting content to be banned by regular expressions. This opens the way for powerful invalidation schemes, such as tagging cache entries.

CHAPTER 3

Proxy Server Configuration

You need to configure the proxy server of your choice (Varnish, NGINX or Symfony HttpCache) to work with FOS-HttpCache. These guides help you for the configuration for the features of this library. You will still need to know about the other features of the proxy server to get everything right.

Varnish Configuration

Below you will find detailed Varnish configuration recommendations for the features provided by this library. The configuration is provided for Varnish 3, 4 and 5.

Basic Varnish Configuration

To invalidate cached objects in Varnish, begin by adding an ACL (for Varnish 3 see ACL for Varnish 3) to your Varnish configuration. This ACL determines which IPs are allowed to issue invalidation requests. To use the provided configuration fragments, this ACL has to be named `invalidators`. The most simple ACL, valid for all Varnish versions from 3 onwards, looks as follows:

```
# /etc/varnish/your_varnish.vcl

acl invalidators {
    "localhost";
    # Add any other IP addresses that your application runs on and that you
    # want to allow invalidation requests from. For instance:
    # "192.168.1.0"/24;
}
```

Important: Make sure that all web servers running your application that may trigger invalidation are whitelisted here. Otherwise, lost cache invalidation requests will lead to lots of confusion.

Provided VCL Subroutines

In order to ease configuration we provide a set of VCL subroutines in the `resources/config` directory. These can be included from your main Varnish configuration file, typically `default.vcl`. Then you need to make your `VCL_*` subroutines call the `fos_*` routines.

Tip: When including one of the provided VCL, you need to call all the defined subroutines or your configuration will not be valid.

See the respective sections below on how to configure usage of each of the provided VCLs.

Purge

Purge removes a specific URL (including query strings) in all its variants (as specified by the `Vary` header).

Subroutines are provided in `resources/config/varnish-[version]/fos_purge.vcl`. To enable this feature, add the following to `your_varnish.vcl`:

- *Varnish 4 & 5*

```
include "path-to-config/varnish/fos_purge.vcl";

sub vcl_recv {
    call fos_purge_recv;
}
```

- *Varnish 3*

```
include "path-to-config/varnish-3/fos_purge.vcl";

sub vcl_recv {
    call fos_purge_recv;
}

sub vcl_hit {
    call fos_purge_hit;
}

sub vcl_miss {
    call fos_purge_miss;
}
```

Read more on handling PURGE requests in the Varnish documentation (for Varnish 3, see purging for Varnish 3).

Refresh

Refresh fetches a page from the backend even if it would still be in the cache, resulting in an updated cache without a cache miss on the next request.

Refreshing applies only to a specific URL including the query string, but *not* its variants.

Subroutines are provided in `resources/config/varnish-[version]/fos_refresh.vcl`. To enable this feature, add the following to `your_varnish.vcl`:

- *Varnish 4 & 5*

```
include "path-to-config/varnish/fos_refresh.vcl";

sub vcl_recv {
    call fos_refresh_recv;
}
```

- *Varnish 3*

```
include "path-to-config/varnish-3/fos_refresh.vcl";

sub vcl_recv {
    call fos_refresh_recv;
}
```

Read more on forcing a refresh in the Varnish documentation (for Varnish 3, see refreshing for Varnish 3).

Ban

Banning invalidates whole groups of cached entries with regular expressions.

Subroutines are provided in `resources/config/varnish-[version]/fos_ban.vcl` To enable this feature, add the following to `your_varnish.vcl`:

- *Varnish 4 & 5*

```
include "path-to-config/varnish/fos_ban.vcl";

sub vcl_recv {
    call fos_ban_recv;
}

sub vcl_backend_response {
    call fos_ban_backend_response;
}

sub vcl_deliver {
    call fos_ban_deliver;
}
```

- *Varnish 3*

```
include "path-to-config/varnish-3/fos_ban.vcl";

sub vcl_recv {
    call fos_ban_recv;
}

sub vcl_fetch {
    call fos_ban_fetch;
}

sub vcl_deliver {
    call fos_ban_deliver;
}
```

This subroutine also sets the `X-Url` and `X-Host` headers on the cache object. These headers are used by the Varnish ban lurker that crawls the content to eventually throw out banned data even when it's not requested by any client. Read

more on handling BAN requests in the Varnish documentation (for Varnish 3, see banning for Varnish 3).

Tagging

Feature: *cache tagging*

If you have included `fos_ban.vcl`, tagging will be automatically enabled with the `X-Cache-Tags` header for both marking the tags on the response and for the invalidation request to tell what tags to invalidate.

If you use a different name for *response tagging* than the default `X-Cache-Tags` or a different name for specifying which tags to invalidate in your *cache invalidator configuration* you have to write your own VCL code for tag invalidation. Your custom VCL will look like this:

- *Varnish 4 & 5*

```
/*
 * This file is part of the FOSHttpCache package.
 *
 * (c) FriendsOfSymfony <http://friendsofsymfony.github.com/>
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

sub fos_ban_recv {

    if (req.method == "BAN") {
        if (!client.ip ~ invalidators) {
            return (synth(405, "Not allowed"));
        }

        if (req.http.X-Cache-Tags) {
            ban("obj.http.X-Host ~ " + req.http.X-Host
                + " && obj.http.X-Url ~ " + req.http.X-Url
                + " && obj.http.content-type ~ " + req.http.X-Content-Type
                // the left side is the response header, the right side the␣
↪invalidation header
                + " && obj.http.X-Cache-Tags ~ " + req.http.X-Cache-Tags
            );
        } else {
            ban("obj.http.X-Host ~ " + req.http.X-Host
                + " && obj.http.X-Url ~ " + req.http.X-Url
                + " && obj.http.content-type ~ " + req.http.X-Content-Type
            );
        }

        return (synth(200, "Banned"));
    }
}

sub fos_ban_backend_response {

    # Set ban-lurker friendly custom headers
    set beresp.http.X-Url = bereq.url;
    set beresp.http.X-Host = bereq.http.host;
}

sub fos_ban_deliver {
```

```
    # Keep ban-lurker headers only if debugging is enabled
    if (!resp.http.X-Cache-Debug) {
        # Remove ban-lurker friendly custom headers when delivering to client
        unset resp.http.X-Url;
        unset resp.http.X-Host;

        # Unset the tagged cache headers
        unset resp.http.X-Cache-Tags;
    }
}
```

- *Varnish 3*

```
/*
 * This file is part of the FOSHttpCache package.
 *
 * (c) FriendsOfSymfony <http://friendsofsymfony.github.com/>
 *
 * For the full copyright and license information, please view the LICENSE
 * file that was distributed with this source code.
 */

sub fos_ban_recv {

    if (req.request == "BAN") {
        if (!client.ip ~ invalidators) {
            error 405 "Not allowed.";
        }

        if (req.http.X-Cache-Tags) {
            ban("obj.http.X-Host ~ " + req.http.X-Host
                + " && obj.http.X-Url ~ " + req.http.X-Url
                + " && obj.http.content-type ~ " + req.http.X-Content-Type
                // the left side is the response header, the right side the␣
↪invalidation header
                + " && obj.http.X-Cache-Tags ~ " + req.http.X-Cache-Tags
            );
        } else {
            ban("obj.http.X-Host ~ " + req.http.X-Host
                + " && obj.http.X-Url ~ " + req.http.X-Url
                + " && obj.http.content-type ~ " + req.http.X-Content-Type
            );
        }

        error 200 "Banned";
    }
}

sub fos_ban_fetch {

    # Set ban-lurker friendly custom headers
    set beresp.http.X-Url = req.url;
    set beresp.http.X-Host = req.http.host;
}

sub fos_ban_deliver {

```

```
44      # Keep ban-lurker headers only if debugging is enabled
45      if (!resp.http.X-Cache-Debug) {
46          # Remove ban-lurker friendly custom headers when delivering to client
47          unset resp.http.X-Url;
48          unset resp.http.X-Host;
49
50          # Unset the tagged cache headers
51          unset resp.http.X-Cache-Tags;
52      }
53  }
```

Hint: The line you need to adjust from the code above is line 21. The left side is the header used to tag the response, the right side is the header used when sending invalidation requests. If you change one or the other header name, make sure to adjust the configuration accordingly.

User Context

Feature: *user context hashing*

The `fos_user_context.vcl` needs the `user_context_hash_url` subroutine that sets the URL to do the hash lookup. The default URL is `/_fos_user_context_hash` and you can simply include `resources/config/varnish-[version]/fos_user_context_url.vcl` in your configuration to provide this. If you need a different URL, write your own `user_context_hash_url` subroutine instead.

Tip: The provided VCL to fetch the user hash restarts GET/HEAD requests. It would be more efficient to do the hash lookup request with curl, using the curl Varnish plugin. If you can enable curl support, the recommended way is to implement your own VCL to do a curl request for the hash lookup instead of using the VCL provided here.

Also note that restarting a GET request leads to Varnish discarding the body of the request. If you have some special case where you have GET requests with a body, use curl.

To enable this feature, add the following to `your_varnish.vcl`:

- *Varnish 4 & 5*

```
include "path-to-config/varnish/fos_user_context.vcl";
include "path-to-config/varnish/fos_user_context_url.vcl";

sub vcl_recv {
    call fos_user_context_recv;
}

sub vcl_backend_response {
    call fos_user_context_backend_response;
}

sub vcl_deliver {
    call fos_user_context_deliver;
}
```

- *Varnish 3*

```
include "path-to-config/varnish-3/fos_user_context.vcl";
include "path-to-config/varnish/fos_user_context_url.vcl";

sub vcl_recv {
    call fos_user_context_recv;
}

sub vcl_fetch {
    call fos_user_context_fetch;
}

sub vcl_deliver {
    call fos_user_context_deliver;
}
```

Caching User Specific Content

By default, Varnish does not check for cached data as soon as the request has a `Cookie` or `Authorization` header, as per the builtin VCL (for Varnish 3, see default VCL). For the user context, you make Varnish cache even when there are credentials present.

You need to be very careful when doing this: Your application is responsible for properly specifying what may or may not be shared. If a content only depends on the hash, `Vary` on the header containing the hash and set a `Cache-Control` header to make Varnish cache the request. If the response is individual however, you need to `Vary` on the `Cookie` and/or `Authorization` header and probably want to send a header like `Cache-Control: s-maxage=0` to prevent Varnish from caching.

Your backend application needs to respond to the `application/vnd.fos.user-context-hash` request with *a proper user hash*.

Tip: The provided VCL assumes that you want the context hash to be cached, so we set the `req.url` to a fixed URL. Otherwise Varnish would cache every hash lookup separately.

However, if you have a *paywall scenario*, you need to leave the original URL unchanged. For that case, you would need to write your own VCL.

Cleaning the Cookie Header

In the examples above, an unaltered Cookie header is passed to the backend to use for determining the user context hash. However, cookies as they are sent by a browser are unreliable. For instance, when using Google Analytics, cookie values are different for each request. Because of this, the hash request would not be cached, but multiple hashes would be generated for one and the same user.

To make the hash request cacheable, you must extract a stable user session id *before* calling `fos_user_context_recv`. You can do this as explained in the Varnish documentation:

```
1 sub vcl_recv {
2     # ...
3
4     set req.http.cookie = ";" + req.http.cookie;
5     set req.http.cookie = regsuball(req.http.cookie, "; +", ";");
6     set req.http.cookie = regsuball(req.http.cookie, ";(PHPSESSID)=", "; \1=");
7     set req.http.cookie = regsuball(req.http.cookie, ";[^ ][^;]*", "");
```

```
    set req.http.cookie = regsuball(req.http.cookie, "^[; ]+|[; ]+$", "");

    # ...
}
```

Note: If your application's user authentication is based on a cookie other than PHPSESSID, change `PHPSESSID` to your cookie name.

Custom TTL

By default, the proxy server looks at the `s-maxage` instruction in the `Cache-Control` header to know for how long it should cache a page. But the `Cache-Control` header is also sent to the client. Any caches on the Internet, for example the Internet provider or from a cooperate network might look at `s-maxage` and cache the page. This can be a problem, notably when you do *explicit cache invalidation*. In that scenario, you want your proxy server to keep a page in cache for a long time, but caches outside your control must not keep the page for a long duration.

One option could be to set a high `s-maxage` for the proxy and simply rewrite the response to remove or reduce the `s-maxage`. This is not a good solution however, as you start to duplicate your caching rule definitions.

The solution to this issue provided here is to use a separate, different header called `X-Reverse-Proxy-TTL` that controls the TTL of the proxy server to keep `s-maxage` for other proxies. Because this is not a standard feature, you need to add configuration to your proxy server.

Subroutines are provided in `resources/config/varnish-[version]/fos_custom_ttl.vcl`. The configuration needs to use inline C, which is disabled by default since Varnish 4.0. To use the custom TTL feature, you need to start your Varnish with inline C enabled: `-p vcc_allow_inline_c=on`. Then add the following to `your_varnish.vcl`:

- *Varnish 4 & 5*

```
include "path-to-config/varnish/fos_custom_ttl.vcl";

sub vcl_backend_response {
    call fos_custom_ttl_backend_response;
}
```

- *Varnish 3*

```
include "path-to-config/varnish-3/fos_custom_ttl.vcl";

sub vcl_fetch {
    call fos_custom_ttl_fetch;
}
```

The custom TTL header is removed before sending the response to the client.

Debugging

Configure your Varnish to set a custom header (`X-Cache`) that shows whether a cache hit or miss occurred. This header will only be set if your application sends an `X-Cache-Debug` header:

Subroutines are provided in `fos_debug.vcl`.

To enable this feature, add the following to `your_varnish.vcl`:

- *Varnish 4 & 5*

```
include "path-to-config/varnish/fos_debug.vcl";

sub vcl_deliver {
    call fos_debug_deliver;
}
```

- *Varnish 3*

```
include "path-to-config/varnish-3/fos_debug.vcl";

sub vcl_deliver {
    call fos_debug_deliver;
}
```

NGINX Configuration

Below you will find detailed NGINX configuration recommendations for the features provided by this library. The examples are tested with NGINX version 1.4.6.

NGINX cache is a set of key/value pairs. The key is built with elements taken from the requests (URI, cookies, http headers etc) as specified by `proxy_cache_key` directive.

When we interact with the cache to purge/refresh entries we must send to NGINX a request which has the very same values, for the elements used for building the key, as the request that create the entry. In this way NGINX can build the correct key and apply the required operation to the entry.

By default NGINX key is built with `$scheme$proxy_host$request_uri`. For a full list of the elements you can use in the key see this page from the official documentation.

Purge

NGINX does not support *purge* functionality out of the box but you can easily add it with ngx_cache_purge module. You just need to compile NGINX from sources adding `ngx_cache_purge` with `--add-module`.

You can check the script install-nginx.sh to get an idea about the steps you need to perform.

Then configure NGINX for purge requests:

```
worker_processes 4;

events {
    worker_connections 768;
}

http {

    log_format proxy_cache '$time_local '
        '"$upstream_cache_status | X-Refresh: $http_x_refresh" '
        '"$request" ($status) '
        '"$http_user_agent" ';

    error_log /tmp/fos_nginx_error.log debug;
    access_log /tmp/fos_nginx_access.log proxy_cache;

```

```
    proxy_cache_path /tmp/foshttpcache-nginx keys_zone=FOS_CACHE:10m;

    # Add an HTTP header with the cache status. Required for FOSHttpCache tests.
    add_header X-Cache $upstream_cache_status;

    server {

        listen 127.0.0.1:8088;

        server_name localhost
                    127.0.0.1
                    ;

        proxy_set_header   Host             $host;
        proxy_set_header   X-Real-IP        $remote_addr;
        proxy_set_header   X-Forwarded-For  $proxy_add_x_forwarded_for;

        location / {
            proxy_cache FOS_CACHE;
            proxy_pass http://localhost:8080;
            proxy_set_header Host $host;
            proxy_cache_key $uri$is_args$args;
            proxy_cache_valid 200 302 301 404 1m;

            proxy_cache_purge PURGE from 127.0.0.1;

            # For refresh
            proxy_cache_bypass $http_x_refresh;
        }

        # This must be the same as the $purgeLocation supplied
        # in the Nginx class constructor
        location ~ /purge(/.*) {
            allow 127.0.0.1;
            deny all;
            proxy_cache_purge FOS_CACHE $1$is_args$args;
        }
    }
}
```

Please refer to the ngx_cache_purge module documentation for more on configuring NGINX to support purge requests.

Refresh

If you want to invalidate cached objects by forcing a *refresh* you have to use the built-in proxy_cache_bypass directive. This directive defines conditions under which the response will not be taken from a cache. This library uses a custom HTTP header named `X-Refresh`, so add a line like the following to your config:

```
            proxy_cache_bypass $http_x_refresh;
```

Debugging

Configure your Nginx to set a custom header (`X-Cache`) that shows whether a cache hit or miss occurred:

```
add_header X-Cache $upstream_cache_status;
```

Symfony HttpCache Configuration

Symfony's HttpKernel component provides a reverse proxy implemented completely in PHP, called HttpCache. While it is certainly less efficient than using Varnish or NGINX, it can still provide considerable performance gains over an installation that is not cached at all. It can be useful for running an application on shared hosting for instance.

You can use features of this library with the help of event listeners that act on events of the `HttpCache`. The Symfony `HttpCache` does not have an event system, for this you need to use the trait `EventDispatchingHttpCache` provided by this library. The event listeners handle the requests from the *cache invalidator*.

Note: Symfony `HttpCache` does not currently provide support for banning.

Using the trait

Note: The trait is available since version 2.0.0. Version 1.* of this library instead provided a base `HttpCache` class to extend.

Your `AppCache` needs to implement `CacheInvalidation` and use the trait `FOS\HttpCache\SymfonyCache\EventDispatchingHttpCache`:

```
use FOS\HttpCache\SymfonyCache\CacheInvalidation;
use FOS\HttpCache\SymfonyCache\EventDispatchingHttpCache;
use Symfony\Component\HttpFoundation\Request;
use Symfony\Component\HttpKernel\HttpCache\HttpCache;

class AppCache extends HttpCache implements CacheInvalidation
{
    use EventDispatchingHttpCache;

    /**
     * Made public to allow event listeners to do refresh operations.
     *
     * {@inheritDoc}
     */
    public function fetch(Request $request, $catch = false)
    {
        return parent::fetch($request, $catch);
    }
}
```

The trait adds the `addSubscriber` and `addListener` methods as defined in the `EventDispatcherInterface` to your cache kernel. In addition, it triggers events before and/or after kernel methods to let the listeners interact. If you need to overwrite core `HttpCache` functionality in your kernel, you can provide your own event listeners. If you need to implement functionality directly on the methods, be careful to always call the trait methods rather than going directly to the parent, or events will not be triggered anymore. You might also need to copy a method from the trait and add your own logic between the events to not be too early or too late for the event.

When starting to extend your `AppCache`, it is recommended to use the `EventDispatchingHttpCacheTestCase` to run tests with your kernel to be sure all events are triggered as expected.

Note: If you use `HttpKernel::loadClassCache` from the console, you will need to add `class_exists('FOS\\HttpCache\\SymfonyCache\\CacheEvent');` right after the inclusion of `bootstrap.php.cache` in `app/console`. For web requests, this is done automatically by the trait. If you miss to do so, you will get the following error:

```
Fatal error: Cannot redeclare class Symfony\Component\EventDispatcher\Event in app/
↪cache/dev/classes.php on line ...
```

Cache event listeners

Now that you have an event dispatching kernel, you can make it register the listeners you need. While you could do that from your bootstrap code, this is not the recommended way. You would need to adjust every place you instantiate the cache. Instead, overwrite the constructor of your `AppCache` and register the listeners you need there:

```
use FOS\HttpCache\SymfonyCache\DebugListener();
use FOS\HttpCache\SymfonyCache\CustomTtlListener();
use FOS\HttpCache\SymfonyCache\PurgeListener;
use FOS\HttpCache\SymfonyCache\RefreshListener;
use FOS\HttpCache\SymfonyCache\UserContextListener;

// ...

/**
 * Overwrite constructor to register event listeners for FOSHttpCache.
 */
public function __construct(
    HttpKernelInterface $kernel,
    StoreInterface $store,
    SurrogateInterface $surrogate = null,
    array $options = []
) {
    parent::__construct($kernel, $store, $surrogate, $options);

    $this->addSubscriber(new CustomTtlListener());
    $this->addSubscriber(new PurgeListener());
    $this->addSubscriber(new RefreshListener());
    $this->addSubscriber(new UserContextListener());
    if (isset($options['debug']) && $options['debug']) {
        $this->addSubscriber(new DebugListener());
    }
}
```

The event listeners can be tweaked by passing options to the constructor. The Symfony configuration system does not work here because things in the cache happen before the configuration is loaded.

Purge

To support *cache invalidation*, register the `PurgeListener`. If the default settings are right for you, you don't need to do anything more.

Purging is only allowed from the same machine by default. To purge data from other hosts, provide the IPs of the machines allowed to purge, or provide a RequestMatcher that checks for an Authorization header or similar. *Only set one of ''client_ips'' or ''client_matcher''.*

- **client_ips**: String with IP or array of IPs that are allowed to purge the cache.

  default: `127.0.0.1`

- **client_matcher**: RequestMatcherInterface that only matches requests that are allowed to purge.

  default: `null`

- **purge_method**: HTTP Method used with purge requests.

  default: `PURGE`

Refresh

To support *cache refresh*, register the `RefreshListener`. You can pass the constructor an option to specify what clients are allowed to refresh cache entries. Refreshing is only allowed from the same machine by default. To refresh from other hosts, provide the IPs of the machines allowed to refresh, or provide a RequestMatcher that checks for an Authorization header or similar. *Only set one of ''client_ips'' or ''client_matcher''.*

The refresh listener needs to access the `HttpCache::fetch` method which is protected on the base HttpCache class. The `EventDispatchingHttpCache` exposes the method as public, but if you implement your own kernel, you need to overwrite the method to make it public.

- **client_ips**: String with IP or array of IPs that are allowed to refresh the cache.

  default: `127.0.0.1`

- **client_matcher**: RequestMatcher that only matches requests that are allowed to refresh.

  default: `null`

User Context

To support *user context hashing* you need to register the `UserContextListener`. The user context is then automatically recognized based on session cookies or authorization headers. If the default settings are right for you, you don't need to do anything more. You can customize a number of options through the constructor:

- **anonymous_hash**: Hard-coded hash to use for anonymous users. This is a performance optimization to not do a backend request for users that are not logged in. If you specify a non-empty value for this field, that is used as context hash header instead of doing a hash lookup for anonymous users.
- **user_hash_accept_header**: Accept header value to be used to request the user hash to the backend application. Must match the setup of the backend application.

  default: `application/vnd.fos.user-context-hash`

- **user_hash_header**: Name of the header the user context hash will be stored into. Must match the setup for the Vary header in the backend application.

  default: `X-User-Context-Hash`

- **user_hash_uri**: Target URI used in the request for user context hash generation.

  default: `/_fos_user_context_hash`

- **user_hash_method**: HTTP Method used with the hash lookup request for user context hash generation.

  default: `GET`

- **session_name_prefix**: Prefix for session cookies. Must match your PHP session configuration.

  default: `PHPSESSID`

> **Warning:** If you have a customized session name, it is **very important** that this constant matches it. Session IDs are indeed used as keys to cache the generated use context hash.
>
> Wrong session name will lead to unexpected results such as having the same user context hash for every users, or not having it cached at all, which hurts performance.

Note: To use authorization headers for user context, you might have to add some server configuration to make these headers available to PHP.

With Apache, you can do this for example in a `.htaccess` file:

```
RewriteEngine On
RewriteRule .* - [E=HTTP_AUTHORIZATION:%{HTTP:Authorization}]
```

Cleaning the Cookie Header

By default, the UserContextListener only sets the session cookie (according to the `session_name_prefix` option) in the requests to the backend. If you need a different behavior, overwrite `UserContextListener::cleanupHashLookupRequest` with your own logic.

Custom TTL

By default, the proxy server looks at the `s-maxage` instruction in the `Cache-Control` header to know for how long it should cache a page. But the `Cache-Control` header is also sent to the client. Any caches on the Internet, for example the Internet provider or from a cooperate network might look at `s-maxage` and cache the page. This can be a problem, notably when you do *explicit cache invalidation*. In that scenario, you want your proxy server to keep a page in cache for a long time, but caches outside your control must not keep the page for a long duration.

One option could be to set a high `s-maxage` for the proxy and simply rewrite the response to remove or reduce the `s-maxage`. This is not a good solution however, as you start to duplicate your caching rule definitions.

The solution to this issue provided here is to use a separate, different header called `X-Reverse-Proxy-TTL` that controls the TTL of the proxy server to keep `s-maxage` for other proxies. Because this is not a standard feature, you need to add configuration to your proxy server.

The `CustomTtlListener` looks at a specific header to determine the TTL, preferring that over `s-maxage`. The default header is `X-Reverse-Proxy-TTL` but you can customize that in the listener constructor:

```
new CustomTtlListener('My-TTL-Header');
```

The custom header is removed before sending the response to the client.

Debugging

For the `assertHit` and `assertMiss` assertions to work, you need to add debug information in your AppCache. When running the tests, create the cache kernel with the option `'debug' => true` and add the `DebugListener`.

The `UNDETERMINED` state should never happen. If it does, it means that something went really wrong in the kernel. Have a look at `X-Symfony-Cache` and at the HTML body of the response.

CHAPTER 4

Proxy Client Setup

This library ships with clients for the Varnish and NGINX caching servers and the Symfony built-in HTTP cache.

A Multiplexer client that forwards calls to multiple proxy clients is available, mainly for transition scenarios of your applications. A Noop client that implements the interfaces but does nothing at all is provided for local development and testing purposes.

The recommended usage is to have your application interact with the *cache invalidator* which you set up with the proxy client suitable for the proxy server you use.

Supported invalidation methods

Not all clients support all *invalidation methods*. This table provides of methods supported by each proxy client:

Client	Purge	Refresh	Ban	Tagging
Varnish	✓	✓	✓	✓
NGINX	✓	✓		
Symfony Cache	✓	✓		
Noop	✓	✓	✓	✓
Multiplexer	✓	✓	✓	✓

Of course, you can also implement your own client for other needs. Have a look at the interfaces in namespace `FOS\HttpCache\ProxyClient\Invalidation`.

Setup

Most proxy clients use the `HttpDispatcher` to send requests to the proxy server. The `HttpDispatcher` is built on top of the HTTPlug abstraction to be independent of specific HTTP client implementations.

Basic HTTP setup with HttpDispatcher

The dispatcher needs to know the IP addresses or hostnames of your proxy servers. If your proxy servers do not run on the default port (80 for HTTP, 443 for HTTPS), you need to specify the port with the server name. Make sure to provide the direct access to the web server without any other proxies that might block invalidation requests.

The server IPs are sufficient for invalidating absolute URLs. If you want to use relative paths in invalidation requests, supply the hostname and possibly a base path to your website with the `$baseUri` parameter:

```
use FOS\HttpCache\ProxyClient\HttpDispatcher;

$servers = ['10.0.0.1', '10.0.0.2:6081']; // Port 80 assumed for 10.0.0.1
$baseUri = 'my-cool-app.com';
$httpDispatcher = new HttpDispatcher($servers, $baseUri);
```

If your web application is accessed on a port other than the default port, make sure to include that port in the base URL:

```
$baseUri = 'my-cool-app.com:8080';
```

You can additionally specify the HTTP client and URI factory that should be used. If you specify a custom HTTP client, you need to configure the client to convert HTTP error status into exceptions. This can either be done in a client specific way or with the HTTPlug `PluginClient` and the `ErrorPlugin`. If client and/or URI factory are not specified, the dispatcher uses HTTPlug discovery to find available implementations.

Learn more about available HTTP clients in the HTTPlug documentation. To customize the behavior of the HTTP client, you can use HTTPlug plugins.

Varnish Client

The Varnish client sends HTTP requests with the `HttpDispatcher`. Create the dispatcher as explained above and pass it to the Varnish client:

```
use FOS\HttpCache\ProxyClient\Varnish;

$varnish = new Varnish($httpDispatcher);
```

Note: To make invalidation work, you need to *configure Varnish* accordingly.

You can also pass some options to the Varnish client:

- `tags_header` (default: `X-Cache-Tags`): The HTTP header used to specify which tags to invalidate when sending invalidation requests to the caching proxy. Make sure that your *Varnish configuration* corresponds to the header used here;
- `header_length` (default: 7500): Control the maximum header length when invalidating tags. If there are more tags to invalidate than fit into the header, the invalidation request is split into several requests;
- `default_ban_headers` (default: []): Map of headers that are set on each ban request, merged with the built-in headers.

Additionally, you can specify the request factory used to build the invalidation HTTP requests. If not specified, auto discovery is used – which usually is fine.

A full example could look like this:

```
$options = [
    'tags_header' => 'X-Custom-Tags-Header',
    'header_length' => 4000,
    'default_ban_headers' => [
        'EXTRA-HEADER' => 'header-value',
    ]
];
$requestFactory = new MyRequestFactory();

$varnish = new Varnish($httpDispatcher, $options, $requestFactory);
```

NGINX Client

The NGINX client sends HTTP requests with the `HttpDispatcher`. Create the dispatcher as explained above and pass it to the NGINX client:

```
use FOS\HttpCache\ProxyClient\Nginx;

$nginx = new Nginx($httpDispatcher);
```

If you have configured NGINX to support purge requests at a separate location, call *setPurgeLocation()*:

```
use FOS\HttpCache\ProxyClient\Nginx;

$nginx = new Nginx($servers, $baseUri);
$nginx->setPurgeLocation('/purge');
```

Note: To use the client, you need to *configure NGINX* accordingly.

Symfony Client

The Symfony client sends HTTP requests with the `HttpDispatcher`. Create the dispatcher as explained above and pass it to the Symfony client:

```
use FOS\HttpCache\ProxyClient\Symfony;

$symfony = new Symfony($httpDispatcher);
```

Note: To make invalidation work, you need to *use the EventDispatchingHttpCache*.

Noop Client

The Noop (no operation) client implements the interfaces for invalidation, but does nothing. It is useful for developing your application or on a testing environment that does not have a proxy server set up. Rather than making the cache invalidator optional in your code, you can (based on the environment) determine whether to inject the real client or the Noop client. The rest of your application then does not need to worry about the environment.

Multiplexer Client

The `MultiplexerClient` allows to send invalidation requests to multiple proxy clients.

It is useful when multiple caches exist in the environment and they need to be handled at the same time; the Multiplexer proxy client will forward the cache invalidation calls to all proxy clients supporting the operation in question:

```
use FOS\HttpCache\ProxyClient\MultiplexerClient;
use FOS\HttpCache\ProxyClient\Nginx;
use FOS\HttpCache\ProxyClient\Symfony;

$nginxClient = new Nginx($servers);
$symfonyClient = new Symfony([...]);
// Expects an array of ProxyClient in the constructor
$client = new MultiplexerClient([$nginxClient, $symfonyClient]);
```

Invalidation calls on `MultiplexerClient` will be forwarded to all proxy clients that support the *invalidation method* and be ignored if none do. Calling `getTagsHeaderValue` and `getTagsHeaderName` will throw an `UnsupportedProxyOperationException` if none of the proxy clients support tagging (i.e., implement `TagCapable`).

Note: Having multiple layers of HTTP caches in place is not a good idea in general. The `MultiplexerClient` is provided for special situations, for example during a transition phase of an application where an old and a new system run in parallel.

Note: When using the multiplexer, code relying on `instanceof` checks on the client and also the `CacheInvalidator::supports` method will not work, as the `MultiplexerClient` implements all interfaces, but the attached clients might not. Make sure that none of the code you use relies on such checks - or write your own multiplexer that only implements the interfaces supported by the clients you use.

Using the Proxy Client

The recommended usage of the proxy client is to create an instance of `CacheInvalidator` with the correct client for your setup. See *The Cache Invalidator* for more information.

Implementation Notes

Each client is an implementation of ProxyClient. All other interfaces, `PurgeCapable`, `RefreshCapable`, `BanCapable` and `TagCapable`, extend this `ProxyClient`. So each client implements at least one of the three *invalidation methods* depending on the proxy server's abilities. To interact with a proxy client directly, refer to the doc comments on the interfaces.

The `ProxyClient` has one method: `flush()`. After collecting invalidation requests, `flush()` needs to be called to actually send the requests to the proxy server. This is on purpose: this way, we can send all requests together, reducing the performance impact of sending invalidation requests.

CHAPTER 5

The Cache Invalidator

Use the cache invalidator to invalidate or refresh paths, URLs and headers. It is the invalidator that you will probably use most when interacting with the library.

Setup

Create the cache invalidator by passing a proxy client as adapter:

```
use FOS\HttpCache\CacheInvalidator;
use FOS\HttpCache\ProxyClient;

$client = new ProxyClient\Varnish(...);
// or
$client = new ProxyClient\Nginx(...);
// or
$client = new ProxyClient\Symfony(...);
// or, for local development
$client = new ProxyClient\Noop();

$cacheInvalidator = new CacheInvalidator($client);
```

Depending on the capabilities of the proxy client, some invalidation methods may not work. If you try to call an invalidation method that is not supported, an `UnsupportedProxyOperationException` is thrown. You can check for support by calling `CacheInvalidator::support` with the constant of the operation you need.

See *proxy clients* for the details on setting up the proxy client and an overview of the supported operations of each client.

Invalidating Paths and URLs

Note: Make sure to *configure your proxy* for purging first.

Invalidate a path:

```
$cacheInvalidator->invalidatePath('/users')->flush();
```

See below for the *flush()* method.

Invalidate a URL:

```
$cacheInvalidator->invalidatePath('http://www.example.com/users')->flush();
```

Invalidate a URL with added header(s):

```
$cacheInvalidator->invalidatePath(
    'http://www.example.com/users',
    ['Cookie' => 'foo=bar; fizz=bang']
)->flush();
```

This allows you to pass headers that are different between invalidation requests. If you want to add a header to all requests, such as `Authorization`, *configure the HTTP client* to use a custom HTTP client instead.

Please note that purge will invalidate all variants, so you do not need to send any headers that you vary on, such as `Accept`.

Refreshing Paths and URLs

Note: Make sure to *configure your proxy* for refreshing first.

```
$cacheInvalidator->refreshPath('/users')->flush();
```

Refresh a URL:

```
$cacheInvalidator->refreshPath('http://www.example.com/users')->flush();
```

Refresh a URL with added header(s):

```
$cacheInvalidator->refreshPath(
    'http://www.example.com/users',
    ['Cookie' => 'foo=bar; fizz=bang']
)->flush();
```

This allows you to pass headers that are different between invalidation requests. If you want to add a header to all requests, such as `Authorization`, *configure the HTTP client* to use a custom HTTP client instead.

Invalidating by Tags

Note: Make sure to *configure your proxy* for tagging first, in the case of Varnish this is powered by banning.

When you are using *response tagging*, you can invalidate all responses that where tagged with a specific label.

Invalidate a tag:

```
$cacheInvalidator->invalidateTags(['blog-post-44'])->flush();
```

See below for the *flush()* method.

Invalidate several tags:

```
$cacheInvalidator
    ->invalidateTags(['type-65', 'location-3'])
    ->flush()
;
```

Invalidating With a Regular Expression

Note: Make sure to *configure your proxy* for banning first.

URL, Content Type and Hostname

You can invalidate all URLs matching a regular expression by using the `invalidateRegex` method. You can further limit the cache entries to invalidate with a regular expression for the content type and/or the application hostname.

For instance, to invalidate all .css files for all hostnames handled by this proxy server:

```
$cacheInvalidator->invalidateRegex('.*css$')->flush();
```

To invalidate all `.png` files on host example.com:

```
$cacheInvalidator
    ->invalidateRegex('.*', 'image/png', ['example.com'])
    ->flush()
;
```

Any Header

You can also invalidate the cache based on any headers.

Note: If you use non-default headers, make sure to *configure your proxy* to have them taken into account.

Proxy client implementations should fill up the headers to at least have the default headers always present to simplify the cache configuration rules.

To invalidate on a custom header `My-Header`, you would do:

```
$cacheInvalidator->invalidate(['My-Header' => 'my-value'])->flush();
```

Flushing

The CacheInvalidator internally queues the invalidation requests and only sends them out to your HTTP proxy when you call `flush()`:

```
$cacheInvalidator
    ->invalidateRoute(...)
    ->invalidatePath(...)
    ->flush()
;
```

Try delaying flush until after the response has been sent to the client's browser. This keeps the performance impact of sending invalidation requests to a minimum.

When using the FOSHttpCacheBundle, you don't have to call `flush()`, as the bundle flushes the invalidator for you after the response has been sent.

As `flush()` empties the invalidation queue, you can safely call the method multiple times. If there are no requests to be sent, flush will simply do nothing.

Error handling

If an error occurs during `flush()`, the method throws an ExceptionCollection that contains an exception for each failed request to the proxy server.

These exception are of two types:

- `\FOS\HttpCache\ProxyUnreachableException` when the client cannot connect to the proxy server
- `\FOS\HttpCache\ProxyResponseException` when the proxy server returns an error response, such as 403 Forbidden.

So, to catch exceptions:

```
use FOS\HttpCache\Exception\ExceptionCollection;

$cacheInvalidator
    ->invalidatePath('/users');

try {
    $cacheInvalidator->flush();
} catch (ExceptionCollection $exceptions) {
    // The first exception that occurred
    var_dump($exceptions->getFirst());

    // Iterate over the exception collection
    foreach ($exceptions as $exception) {
        var_dump($exception);
    }
}
```

Logging errors

You can log any exceptions with the help of the `LogListener` provided in this library. First construct a logger that implements `\Psr\Log\LoggerInterface`. For instance, when using Monolog:

```
use Monolog\Logger;

$monolog = new Logger(...);
$monolog->pushHandler(...);
```

Then add the logger as a listener to the cache invalidator:

```
use FOS\HttpCache\EventListener\LogListener;

$logListener = new LogListener($monolog);
$cacheInvalidator->getEventDispatcher()->addSubscriber($logListener);
```

Now, if you flush the invalidator, errors will be logged:

```
use FOS\HttpCache\Exception\ExceptionCollection;

$cacheInvalidator->invalidatePath(...)
    ->invalidatePath(...);

try {
    $cacheInvalidator->flush();
} catch (ExceptionCollection $exceptions) {
    // At least one failed request, check your logs!
}
```

CHAPTER 6

Response Tagging

The `ResponseTagger` helps you keep track of tags for a response. It can add the tags as a response header that you can later use to invalidate all cache entries with that tag.

Setup

Note: Make sure to *configure your proxy* for tagging first.

The response tagger uses an instance of `TagHeaderFormatter` to know the header name used to mark tags on the content and to format the tags into the correct header value. This library ships with a `CommaSeparatedTagHeaderFormatter` that formats an array of tags into a comma-separated list. This format is expected for invalidation with the Varnish reverse proxy. When using the default settings, everything is created automatically and the `X-Cache-Tags` header will be used:

```
use FOS\HttpCache\ResponseTagger;

$responseTagger = new ResponseTagger();
```

If you need a different behavior, you can provide your own implementation of the `TagHeaderFormatter` interface. But be aware that your *Varnish configuration* has to match with the tag on the response. For example, to use a different header name:

```
use FOS\HttpCache\ResponseTagger;
use FOS\HttpCache\TagHeaderFormatter;

$formatter = new CommaSeparatedTagHeaderFormatter('Custom-Header-Name');
$responseTagger = new ResponseTagger(['header_formatter' => $formatter]);
```

The response tagger validates tags that you set. By default, it simply ignores empty strings and does not add them to the list of tags. You can set the response tagger to strict mode to have it throw an `InvalidTagException` on empty tags:

```
$responseTagger = new ResponseTagger(['strict' => true]);
```

Usage

With tags you can group related representations so it becomes easier to invalidate them. You will have to make sure your web application adds the correct tags on all responses. You can add tags to the response using:

```
$responseTagger->addTags(['tag-two', 'group-a']);
```

Before any content is sent out, you need to send the tag header:

```
header(sprintf('%s: %s',
    $responseTagger->getTagsHeaderName(),
    $responseTagger->getTagsHeaderValue()
));
```

Tip: If you are using Symfony with the FOSHttpCacheBundle, the tags added to `ResponseTagger` are added to the response automatically. You also have additional methods of defining tags with annotations and on URL patterns.

Assume you sent four responses:

Response:	`X-Cache-Tags` header:
`/one`	`tag-one`
`/two`	`tag-two, group-a`
`/three`	`tag-three, group-a`
`/four`	`tag-four, group-b`

You can now invalidate some URLs using tags:

```
$tagHandler->invalidateTags(['group-a', 'tag-four'])->flush();
```

This will ban all requests having either the tag `group-a` /or/ `tag-four`. In the above example, this will invalidate `/two`, `/three` and `/four`. Only `/one` will stay in the cache.

Note: For further reading on tag invalidation see *cache-invalidator page*. For changing the cache header, *configure your proxy*.

CHAPTER 7

Cache on User Context

Some applications differentiate the content between types of users. For instance, on one and the same URL a guest sees a 'Log in' message; an editor sees an 'Edit' button and the administrator a link to the admin backend.

The FOSHttpCache library includes a solution to cache responses per user context (whether the user is authenticated, groups the user is in, or other information), rather than individually.

If every user has their own hash, you probably don't want to cache at all. Or if you found out its worth it, vary on the credentials and don't use the context hash mechanism.

Caution: Whenever you share caches, make sure to **not output any individual content** like the user name. If you have individual parts of a page, you can load those parts over AJAX requests or look into ESI and make the ESI sub response vary on the cookie or completely non-cached. Both approaches integrate with the concepts presented in this chapter.

You do **not want intermediary proxies to cache responses** that depend on the context. If the client will not see a difference when his context changes (e.g. is removed from or added to groups on server side), you also do not want the clients to cache pages. Because `VARY` is used for the control of the proxy server, it is not available to control clients. Often, the best solution is to disable intermediary caches by setting the cache control header `s-maxage` to 0 and using the *custom TTL* mechanism (see the documentation for *Varnish* or the *Symfony HttpCache*). If you want to use the `private` cache control instruction instead, you need to adjust your proxy server configuration to cache content with a `private` instruction.

Overview

Caching on user context works as follows:

1. A *client* requests `/foo.php` (the *original request*).
2. The *proxy server* receives the request. It sends a request (the *hash request*) with a special accept header (`application/vnd.fos.user-context-hash`) to a specific URL, e.g., `/_fos_user_context_hash`.

3. The *application* receives the hash request. The application knows the client's user context (roles, permissions, etc.) and generates a hash based on that information. The application then returns a response containing that hash in a custom header (`X-User-Context-Hash`) and with `Content-Type application/vnd.fos.user-context-hash`.
4. The proxy server receives the hash response, copies the hash header to the client's original request for `/foo.php` and restarts that request.
5. If the response to this request should differ per user context, the application specifies so by setting a `Vary: X-User-Context-Hash` header. The appropriate user role dependent representation of `/foo.php` will then be returned to the client.

After this happened the first time, the hash can be cached by Varnish for this client, moving step 2-4 into the cache. After the page is in cache, subsequent requests from clients that got the same hash can be served from the cache as well.

Note: If your application starts sessions for anonymous users, you will get one hash lookup request for each of those users. Your application can return the same hash for authenticated users with no special privileges as for anonymous users with a session cookie.

If there is no cookie and no authentication information, the hash lookup is skipped and no hash header added to the request. However, we can not avoid the initial hash lookup request per different cookie, as the caching proxy can not know which session cookies indicate a logged in user and which an anonymous session.

Proxy Client Configuration

Currently, user context caching is only supported by Varnish and by the Symfony HttpCache. See the *Varnish Configuration* or *Symfony HttpCache Configuration*.

User Context Hash from Your Application

It is your application's responsibility to determine the hash for a user. Only your application can know what is relevant for the hash. You can use the path or the accept header to detect that a hash was requested.

> **Warning:** Treat the hash lookup path like the login path so that anonymous users also can get a hash. That means that your cache can access the hash lookup even with no user provided credential and that the hash lookup never redirects to a login page.

Calculating the User Context Hash

The user context hash calculation (step 3 above) is managed by a HashGenerator. Because the calculation itself will be different per application, you need to implement at least one ContextProvider and register that with the DefaultHashGenerator:

```
use FOS\HttpCache\UserContext\DefaultHashGenerator;

$hashGenerator = new DefaultHashGenerator([
    new IsAuthenticatedProvider(),
```

```
    new RoleProvider(),
]);
```

Once all providers are registered, call `generateHash()` to get the hash for the current user context.

Note: If you need custom logic in the hash generator you can create your own class implementing the HashGenerator interface.

Context Providers

Each provider is passed the UserContext and updates that with parameters which influence the varied response.

A provider that looks at whether the user is authenticated could look like this:

```
use FOS\HttpCache\UserContext\ContextProvider;
use FOS\HttpCache\UserContext\UserContext;

class IsAuthenticatedProvider implements ContextProvider
{
    protected $userService;

    public function __construct(YourUserService $userService)
    {
        $this->userService = $userService;
    }

    public function updateUserContext(UserContext $userContext)
    {
        $userContext->addParameter('authenticated', $this->userService->
↪isAuthenticated());
    }
}
```

Returning the User Context Hash

It is up to you to return the user context hash in response to the hash request (`/_fos_user_context_hash` in step 3 above):

```
// <web-root>/_fos_user_context_hash/index.php

$hash = $hashGenerator->generateHash();

if ('application/vnd.fos.user-context-hash' == strtolower($_SERVER['HTTP_ACCEPT'])) {
    header(sprintf('X-User-Context-Hash: %s', $hash));
    header('Content-Type: application/vnd.fos.user-context-hash');
    exit;
}

// 406 Not acceptable in case of an incorrect accept header
header('HTTP/1.1 406');
```

If you use Symfony, the FOSHttpCacheBundle will set the correct response headers for you.

Caching the Hash Response

To optimize user context hashing performance, you should cache the hash response. By varying on the Cookie and Authorization header, the application will return the correct hash for each user. This way, subsequent hash requests (step 3 above) will be served from cache instead of requiring a roundtrip to the application.

```
// The application listens for hash request (by checking the accept header)
// and creates an X-User-Context-Hash based on parameters in the request.
// In this case it's based on Cookie.
if ('application/vnd.fos.user-context-hash' === strtolower($_SERVER['HTTP_ACCEPT'])) {
    header(sprintf('X-User-Context-Hash: %s', $_COOKIE[0]));
    header('Content-Type: application/vnd.fos.user-context-hash');
    header('Cache-Control: max-age=3600');
    header('Vary: cookie, authorization');

    exit;
}
```

Here we say that the hash is valid for one hour. Keep in mind, however, that you need to invalidate the hash response when the parameters that determine the context change for a user, for instance, when the user logs in or out, or is granted extra permissions by an administrator.

Note: If you base the user hash on the Cookie header, you should *clean up that header* to make the hash request properly cacheable.

The Original Request

After following the steps above, the following code renders a homepage differently depending on whether the user is logged in or not, using the *credentials of the particular user*:

```
// /index.php file
header('Cache-Control: max-age=3600');
header('Vary: X-User-Context-Hash');

$authenticationService = new AuthenticationService();

if ($authenticationService->isAuthenticated()) {
    echo "You are authenticated";
} else {
    echo "You are anonymous";
}
```

Alternative for Paywalls: Authorization Request

If you can't efficiently determine a general user hash for the whole application (e.g. you have a paywall where individual users are limited to individual content), you can follow a slightly different approach:

- Instead of doing a hash lookup request to a specific authentication URL, you keep the request URL unchanged, but send a HEAD request with a specific `Accept` header.
- In your application, you intercept such requests after the access decision has taken place but before expensive operations like loading the actual data have taken place and return early with a 200 or 403 status.

- If the status was 200, you restart the request in Varnish, and cache the response even though a `Cookie` or `Authorization` header is present, so that further requests on the same URL by other authorized users can be served from cache. On status 403 you return an error page or redirect to the URL where the content can be bought.

CHAPTER 8

Testing Your Application

This chapter describes how to test your application against your reverse proxy. By running your tests against a live instance of your proxy server, you can validate the caching headers that your application sets, and the invalidation rules that it defines.

The FOSHttpCache library provides traits and base test classes to help you write functional tests with PHPUnit 5.7 and 6.*. Using the traits, you can extend your own (or your framework's) base test classes. For convenience, you can also extend the FOSHttpCache base test class suitable for your proxy server, which includes a sensible set of traits.

New in version 2.1: The testing has been updated to support PHPUnit 6 in version 2.1. If you an older version of FOSHttpCache and want to use the features described in this chapter, you need to use PHPUnit 5 to run the tests.

By using the traits, you get:

- independent tests: all previously cached content is removed in the test's `setUp()` method;
- an instance of this library's proxy client that is configured to talk to your proxy server for invalidation requests;
- a convenience method for executing HTTP requests to your proxy server: `$this->getResponse()`;
- custom assertions `assertHit()` and `assertMiss()` for validating a cache hit/miss.

Configuration

The recommended way to configure the test case is by setting constants in your `phpunit.xml`. Alternatively, you can override the getter methods.

Web Server

You will need to run a web server to provide the PHP application you want to test. The test cases only handle running the proxy server. It's easiest to use PHP's built in web server. Include the WebServerListener in your `phpunit.xml`:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit ...>
    <listeners>
        <listener class="\FOS\HttpCache\Test\WebServerListener" />
    </listeners>
</phpunit>
```

Then set the `webserver` group on your test to start PHP's web server before it runs:

```
class YourTest extends \PHPUnit_Framework_TestCase
{
    /**
     * @group webserver
     */
    public function testYourApp()
    {
        // The web server will be started before this test code runs and
        // shut down again after it finishes.
    }
}
```

Setting Constants

Compare this library's configuration to see how the constants are set:

```
<?xml version="1.0" encoding="UTF-8"?>
<phpunit ...>
        <const name="NGINX_FILE" value="./tests/Functional/Fixtures/nginx/fos.conf" />
        <const name="WEB_SERVER_HOSTNAME" value="localhost" />
        <const name="WEB_SERVER_PORT" value="8080" />
        <const name="WEB_SERVER_DOCROOT" value="./tests/Functional/Fixtures/web" />
    </php>
</phpunit>
```

Overriding Getters

You can override getters in your test class in the following way:

```
use FOS\HttpCache\Test\VarnishTestCase;

class YourFunctionalTest extends VarnishTestCase
{
    protected function getVarnishPort()
    {
        return 8000;
    }
}
```

Traits

Proxy Server Traits

FOSHttpCache provides three proxy server traits that:

- if necessary, start your proxy server before running the tests;
- clear any cached content between tests to guarantee test isolation;
- if necessary, stop the proxy server after the tests have finished;
- provide `getProxyClient()`, which returns the right *proxy client* for your proxy server.

You only need to include one of these traits in your test classes. Which one you need (`VarnishTest`, `NginxTest` or `SymfonyTest`) depends on the proxy server that you use.

VarnishTest Trait

Requires Symfony's Process component, so make sure to include that in your project:

```
$ composer require symfony/process
```

Then configure the following parameters. The web server hostname and path to your VCL file are required.

Then set your Varnish configuration (VCL) file. Configuration is handled either by overwriting the getter or by defining a PHP constant. You can set the constants in your `phpunit.xml` or in the bootstrap file. Available configuration parameters are:

Constant	Getter	Default	Description
`WEB_SERVER_HOSTNAME`	`getHostName()`		hostname your application can be reached at
`VARNISH_FILE`	`getConfigFile()`		your Varnish configuration (VCL) file
`VARNISH_BINARY`	`getBinary()`	`varnishd`	your Varnish binary
`VARNISH_PORT`	`getCachingProxyPort()`	`6181`	port Varnish listens on
`VARNISH_MGMT_PORT`	`getVarnishMgmtPort()`	`6182`	Varnish management port
`VARNISH_CACHE_DIR`	`getCacheDir()`	`sys_get_temp_dir()` + `/foshttpcache-varnish`	directory to use for cache
`WEB_SERVER_HOSTNAME`	`getHostName()`		hostname your application can be reached at

The Varnish version is controlled by an environment variable (in case you want to test both Varnish 3 and 4 on a continuous integration system). See the `.travis.yml` of the FOSHttpCache git repository for an example.

Environment Variable	Getter	Default	Description
`VARNISH_VERSION`	`getVarnishVersion()`	`4`	version of Varnish application that is used

See `tests/bootstrap.php` for an example how this repository uses the version information to set the right `VARNISH_FILE` constant.

Enable Assertions

For the *assertHit* and *assertMiss* assertions to work, you need to add a *custom X-Cache header* to responses served by your Varnish.

NginxTest Trait

Requires Symfony's Process component, so make sure to include that in your project:

```
$ composer require symfony/process
```

Then configure the following parameters. The web server hostname and path to your NGINX configuration file are required.

Constant	Getter	Default	Description
WEB_SERVER_HOSTNAME	getHostName()		hostname your application can be reached at
NGINX_FILE	getConfigFile()		your NGINX configuration file
NGINX_BINARY	getBinary()	nginx	your NGINX binary
NGINX_PORT	getCachingProxyPort()	8088	port NGINX listens on
NGINX_CACHE_PATH	getCacheDir()	sys_get_temp_dir() + /foshttpcache-nginx	directory to use for cache Must match *proxy_cache_path* directive in your configuration file.

SymfonyTest Trait

It is assumed that the web server you run for the application has the HttpCache integrated.

Constant	Getter	Default	Description
WEB_SERVER_HOSTNAME	getHostName()		Hostname your application can be reached at
WEB_SERVER_PORT	getCachingProxyPort()		The port on which the web server runs
SYMFONY_CACHE_DIR	getCacheDir()	sys_get_temp_dir() + / foshttpcache-nginx	directory to use for cache Must match the configuration of your HttpCache and must be writable by the user running PHPUnit.

HttpCaller Trait

Provides your tests with a `getResponse` method, which retrieves a URI from your application through a real HTTP call that goes through the HTTP proxy server:

```
use FOS\HttpCache\Test\HttpCaller;
use PHPUnit\Framework\TestCase;

class YourTest extends TestCase
{
    use HttpCaller;

    public function testCachingHeaders()
    {
        // Get some response from your application
        $response = $this->getResponse('/path');

        // Optionally with request headers and a custom  method
        $response = $this->getResponse('/path', ['Accept' => 'text/json'], 'PUT');
    }
}
```

This trait requires the methods `getHostName()` and `getCachingProxyPort()` to exist. When using one of the proxy server traits, these will be provided by the trait, otherwise you have to implement them.

CacheAssertions Trait

Provides cache hit/miss assertions to your tests. To enable the these `assertHit` and `assertMiss` assertions, you need to configure your proxy server to set an `X-Cache` header with the cache status:

- *Varnish*
- *NGINX*
- *Symfony HttpCache*

Then use the assertions as follows:

```
use FOS\HttpCache\Test\CacheAssertions;
use PHPUnit\Framework\TestCase;

class YourTest extends TestCase
{
    public function testCacheHitOrMiss()
    {
        // Assert the application response is a cache miss
        $this->assertMiss($response);

        // Or assert it is a hit
        $this->assertHit($response);
    }
}
```

Base Classes for Convenience

If you prefer, you can extend your test classes from `VarnishTestCase`, `NginxTestCase` or `SymfonyTestCase`. The appropriate traits will then automatically be included.

Usage

This example shows how you can test whether the caching headers your application sets influence your proxy server as you expect them to:

```
use FOS\HttpCache\Test\CacheAssertions;
use FOS\HttpCache\Test\HttpCaller;
use FOS\HttpCache\Test\VarnishTest;
// or FOS\HttpCache\Test\NginxTest;
// or FOS\HttpCache\Test\SymfonyTest;
use PHPUnit\Framework\TestCase;

class YourTest extends TestCase
{
    public function testCachingHeaders()
    {
        // The proxy server is (re)started, so you don't have to worry
        // about  previously cached content. Before continuing, the
        // VarnishTest/ NginxTest trait waits for the proxy server to
        // become available.

        // Retrieve a URL from your application
```

```
        $response = $this->getResponse('/your/resource');

        // Assert the response was a cache miss (came from the backend
        // application)
        $this->assertMiss($response);

        // Assume the URL /your/resource sets caching headers. If we
        // retrieve it again, we should have a cache hit (response delivered
        // by the proxy server):
        $response = $this->getResponse('/your/resource');
        $this->assertHit($response);
    }
}
```

This example shows how you can test whether your application purges content correctly:

```
use FOS\HttpCache\Test\CacheAssertions;
use FOS\HttpCache\Test\HttpCaller;
use FOS\HttpCache\Test\VarnishTest;
// or FOS\HttpCache\Test\NginxTest;
// or FOS\HttpCache\Test\SymfonyTest;
use PHPUnit\Framework\TestCase;

class YourTest extends TestCase
{
    public function testCachePurge()
    {
        // Again, the proxy server is restarted, so your test is independent
        // from other tests

        $url = '/blog/articles/1';

        // First request must be a cache miss
        $this->assertMiss($this->getResponse($url));

        // Next requests must be a hit
        $this->assertHit($this->getResponse($url));

        // Purge
        $this->getProxyClient()->purge('/blog/articles/1');

        // First request after must again be a miss
        $this->assertMiss($this->getResponse($url));
    }
}
```

For more ideas, see this library's functional tests in the tests/Functional/ directory.

CHAPTER 9

Contributing

We warmly welcome contributions to FOSHttpCache. Before you invest a lot of time however, please open an issue on GitHub to discuss your idea. Then we can coordinate efforts if somebody is already working on the same thing.

Testing the Library

This chapter describes how to run the tests that are included with this library. Make sure that you have PHPUnit 5.7 or higher installed.

First clone the repository, install the vendors, then run the tests:

```
$ git clone https://github.com/FriendsOfSymfony/FOSHttpCache.git
$ cd FOSHttpCache
$ composer install --dev
$ phpunit
```

Unit Tests

To run the unit tests separately:

```
$ phpunit tests/Unit
```

Functional Tests

The library also includes functional tests against a Varnish and NGINX instance. The functional test suite by default uses PHP's built-in web server. If you have PHP 5.4 or newer, simply run with the default configuration.

If you want to run the tests on HHVM, you need to configure a web server listening on localhost:8080 that points to the folder `tests/Functional/Fixtures/web` and start a HHVM FastCGI server.

To run the functional tests:

```
$ phpunit tests/Functional
```

Tests are organized in groups: one for each reverse proxy supported. At the moment groups are: *varnish* and *nginx*.

To run only the *varnish* functional tests:

```
$ phpunit --group=varnish
```

For more information about testing, see *Testing Your Application*.

Building the Documentation

First install Sphinx and install enchant (e.g. `sudo apt-get install enchant`), then download the requirements:

```
$ pip install -r doc/requirements.txt
```

To build the docs:

```
$ cd doc
$ make html
$ make spelling
```

Index