FOSHttpCache Documentation

Release 1.0.0

David Buchmann, David de Boer

Jul 16, 2020

Contents

This is the documentation for the FOSHttpCache library.

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 caching proxy 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

1.1 Installation

The FOSHttpCache library is available on Packagist. You can install it using Composer:

```
$ composer require friendsofsymfony/http-cache:~1.0
```

Note: This library follows Semantic Versioning. Because constraint `~1.0` will only increment the minor and patch numbers, it will not introduce BC breaks.

1.2 Configuration

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

1. *configure your caching proxy*
2. *set up a client for your caching proxy*
3. *set up the cache invalidator*

1.3 Overview

This library mainly consists of:

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

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.

2.1 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.

Caching proxy Also reverse caching proxy. Examples: Varnish, NGINX.

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 caching proxy 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.

2.2 What is Cache Invalidation?

There are only two hard things in Computer Science: cache invalidation and naming things.

– *Phil Karlton*

2.2.1 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.

2.2.2 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.

2.2.3 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 caching proxy. 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.

2.3 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 caching proxy immediately. The next time a client requests the URL, data is fetched from the application, stored in the caching proxy, 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 caching proxy 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

Caching Proxy Configuration

You need to configure the caching proxy of your choice. 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 caching proxy to get everything right.

3.1 Varnish Configuration

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

3.1.1 Basic Varnish Configuration

To invalidate cached objects in Varnish, begin by adding an ACL to your Varnish configuration. This ACL determines which IPs are allowed to issue invalidation requests. Let's call the ACL *invalidators*. The ACL below will be used throughout the Varnish examples on this page.

```
# /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.

3.1.2 Purge

To configure Varnish for handling PURGE requests:

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

- *Varnish 4*

```
sub vcl_recv {
    if (req.method == "PURGE") {
        if (!client.ip ~ invalidators) {
            return (synth(405, "Not allowed"));
        }
        return (purge);
    }
}
```

- *Varnish 3*

```
sub vcl_recv {
    if (req.request == "PURGE") {
        if (!client.ip ~ invalidators) {
            error 405 "Not allowed";
        }
        return (lookup);
    }
}

sub vcl_hit {
    if (req.request == "PURGE") {
        purge;
        error 204 "Purged";
    }
}

# The purge in vcl_miss is necessary to purge all variants in the cases where
# you hit an object, but miss a particular variant.
sub vcl_miss {
    if (req.request == "PURGE") {
        purge;
        error 204 "Purged (Not in cache)";
    }
}
```

3.1.3 Refresh

If you want to invalidate cached objects by forcing a refresh add the following to your Varnish configuration:

Refresh invalidates a specific URL including the query string, but *not* its variants.

```
sub vcl_recv {
    if (req.http.Cache-Control ~ "no-cache" && client.ip ~ invalidators) {
        set req.hash_always_miss = true;
    }
}
```

3.1.4 Ban

To configure Varnish for handling BAN requests:

- *Varnish 4*

```
sub vcl_recv {

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

            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 vcl_backend_response {

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

sub vcl_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 resp.http.X-Cache-Tags;
    }
}
```

- *Varnish 3*

```
sub vcl_recv {

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

            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";
    }
}

```

(continues on next page)

(continued from previous page)

```
sub vcl_fetch {

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

sub vcl_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 resp.http.X-Cache-Tags;
    }
}
```

Varnish contains a ban lurker that crawls the content to eventually throw out banned data even when it's not requested by any client.

3.1.5 Tagging

Add the following to your Varnish configuration to enable *cache tagging*.

Note: The custom X-Cache-Tags header should match the tagging header *configured in the cache invalidator*.

- *Varnish 4*

```
sub vcl_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
                + " && 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 vcl_backend_response {
```

(continues on next page)

(continued from previous page)

```

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

sub vcl_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 resp.http.X-Cache-Tags;
    }
}
```

- *Varnish 3*

```
sub vcl_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
                + " && 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 vcl_fetch {

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

sub vcl_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;
```

(continues on next page)

(continued from previous page)

```
        unset resp.http.X-Cache-Tags;
    }
}
```

3.1.6 User Context

To support *user context hashing* you need to add some logic to the `recv` and the `deliver` methods:

- *Varnish 4*

```
sub vcl_recv {

    # 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
        )
    ) {
        return (synth(400));
    }

    # Lookup the context hash if there are credentials on the request
    # Only do this for cacheable requests. Returning a hash lookup discards the␣
↪request body.
    # https://www.varnish-cache.org/trac/ticket/652
    if (req.restarts == 0
        && (req.http.cookie || req.http.authorization)
        && (req.method == "GET" || req.method == "HEAD")
    ) {
        # Backup accept header, if set
        if (req.http.accept) {
            set req.http.X-Fos-Original-Accept = req.http.accept;
        }
        set req.http.accept = "application/vnd.fos.user-context-hash";

        # Backup original URL
        set req.http.X-Fos-Original-Url = req.url;
        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 (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;
        unset req.http.X-Fos-Original-Url;
        if (req.http.X-Fos-Original-Accept) {
            set req.http.accept = req.http.X-Fos-Original-Accept;
            unset req.http.X-Fos-Original-Accept;
        } else {
            # If accept header was not set in original request, remove the header␣
↪here.
```

(continues on next page)

(continued from previous page)

```
            unset req.http.accept;
        }

        # 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_backend_response {
    if (bereq.http.accept ~ "application/vnd.fos.user-context-hash"
        && beresp.status >= 500
    ) {
        return (abandon);
    }
}

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-Context-Hash = resp.http.X-User-Context-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-Context-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-Context-Hash;
}
```

- *Varnish 3*

```
sub vcl_recv {

    # 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;
    }

```

(continues on next page)

(continued from previous page)

```
    # Lookup the context hash if there are credentials on the request
    # Only do this for cacheable requests. Returning a hash lookup discards the␣
↪request body.
    # https://www.varnish-cache.org/trac/ticket/652
    if (req.restarts == 0
        && (req.http.cookie || req.http.authorization)
        && (req.request == "GET" || req.request == "HEAD")
    ) {
        # Backup accept header, if set
        if (req.http.accept) {
            set req.http.X-Fos-Original-Accept = req.http.accept;
        }
        set req.http.accept = "application/vnd.fos.user-context-hash";

        # Backup original URL
        set req.http.X-Fos-Original-Url = req.url;
        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;
        unset req.http.X-Fos-Original-Url;
        if (req.http.X-Fos-Original-Accept) {
            set req.http.accept = req.http.X-Fos-Original-Accept;
            unset req.http.X-Fos-Original-Accept;
        } else {
            # If accept header was not set in original request, remove the header␣
↪here.
            unset req.http.accept;
        }

        # 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.
```

(continues on next page)

(continued from previous page)

```
    if (req.restarts == 0
        && resp.http.content-type ~ "application/vnd.fos.user-context-hash"
        && resp.status == 200
    ) {
        set req.http.X-User-Context-Hash = resp.http.X-User-Context-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-Context-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-Context-Hash;
}
```

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

Note: We do not use `X-Original-Url` here, as the header will be sent to the backend and some applications look at this header, which would lead to problems. For example, the Microsoft IIS rewriting module uses this header and Symfony has to look into that header to support IIS.

Note: If you want the context hash to be cached, you need to always set the `req.url` to the same URL, or Varnish will cache every hash lookup separately.

However, if you have a *paywall scenario*, you need to leave the original URL unchanged.

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. You can do this as explained in the Varnish documentation:

```
sub vcl_recv {
    # ...

    set req.http.cookie = ";" + req.http.cookie;
    set req.http.cookie = regsuball(req.http.cookie, "; +", ";");
    set req.http.cookie = regsuball(req.http.cookie, ";(PHPSESSID)=", "; \1=");
    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.

3.1.7 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:

- *Varnish 4*

```
sub vcl_deliver {
    # Add extra headers if debugging is enabled
    # In Varnish 4 the obj.hits counter behaviour has changed, so we use a
    # different method: if X-Varnish contains only 1 id, we have a miss, if it
    # contains more (and therefore a space), we have a hit.
    if (resp.http.X-Cache-Debug) {
        if (resp.http.X-Varnish ~ " ") {
            set resp.http.X-Cache = "HIT";
        } else {
            set resp.http.X-Cache = "MISS";
        }
    }
}
```

- *Varnish 3*

```
sub vcl_deliver {
    # Add extra headers if debugging is enabled
    if (resp.http.X-Cache-Debug) {
        if (obj.hits > 0) {
            set resp.http.X-Cache = "HIT";
        } else {
```

(continues on next page)

(continued from previous page)

```
                set resp.http.X-Cache = "MISS";
            }
        }
    }
```

3.2 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

3.2.1 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;

```

(continues on next page)

(continued from previous page)

```
        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.

3.2.2 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;
```

3.2.3 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;
```

3.3 Symfony HttpCache Configuration

The `symfony/http-kernel` 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 the `EventDispatchingHttpCache` provided here. The basic concept is to use event subscribers on the HttpCache class.

Warning: If you are using the full stack Symfony framework, have a look at the HttpCache provided by the FOSHttpCacheBundle instead.

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

3.3.1 Extending the Correct HttpCache Class

Instead of extending `Symfony\Component\HttpKernel\HttpCache\HttpCache`, your `AppCache` should extend `FOS\HttpCache\SymfonyCache\EventDispatchingHttpCache`.

Tip: If your class already needs to extend a different class, simply copy the event handling code from the EventDispatchingHttpCache into your `AppCache` class and make it implement `CacheInvalidationInterface`. The drawback is that you need to manually check whether you need to adjust your `AppCache` each time you update the FOSHttpCache library.

Now that you have an event dispatching kernel, you can make it register the subscribers 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 AppCache and register the subscribers there. A simple cache will look like this:

```
use FOS\HttpCache\SymfonyCache\EventDispatchingHttpCache;
use FOS\HttpCache\SymfonyCache\UserContextSubscriber;

class AppCache extends EventDispatchingHttpCache
{
    /**
     * Overwrite constructor to register event subscribers for FOSHttpCache.
     */
    public function __construct(HttpKernelInterface $kernel, $cacheDir = null)
    {
        parent::__construct($kernel, $cacheDir);

        $this->addSubscriber(new UserContextSubscriber());
        $this->addSubscriber(new PurgeSubscriber());
        $this->addSubscriber(new RefreshSubscriber());
    }
}
```

3.3.2 Purge

To support *cache purging*, register the `PurgeSubscriber`. 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 purge_client_ips or purge_client_matcher*.

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

  default: `127.0.0.1`

- **purge_client_matcher**: RequestMatcher that only matches requests that are allowed to purge.

  default: `null`

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

  default: `PURGE`

3.3.3 Refresh

To support *cache refresh*, register the `RefreshSubscriber`. 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 refresh_client_ips or refresh_client_matcher*.

The refresh subscriber 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.

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

  default: `127.0.0.1`

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

  default: `null`

3.3.4 User Context

To support *user context hashing* you need to register the `UserContextSubscriber`. 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**: Hash used for anonymous user. This is a performance optimization to not do a backend request for users that are not logged in.

- **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`

- **user_identifier_headers**: List of request headers that authenticate a non-anonymous request.

  default: `['Authorization', 'HTTP_AUTHORIZATION', 'PHP_AUTH_USER']`

- **session_name_prefix**: Prefix for session cookies. Must match your PHP session configuration. If cookies are not relevant in your application, you can set this to `false` to ignore any cookies. (**Only set this to ‘‘false‘‘ if you do not use sessions at all.**)

  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 UserContextSubscriber 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 `UserContextSubscriber::cleanupHashLookupRequest` with your own logic.

CHAPTER 4

Caching Proxy Clients

This library ships with clients for the Varnish, NGINX and Symfony built-in caching proxies. You can use the clients either wrapped by the *cache invalidator* (recommended), or directly for low-level access to invalidation functionality.

4.1 Setup

4.1.1 Varnish Client

At minimum, supply an array containing IPs or hostnames of the Varnish servers that you want to send invalidation requests to. Make sure to include the port Varnish runs on if it is not port 80:

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

$servers = array('10.0.0.1', '10.0.0.2:6081'); // Port 80 assumed for 10.0.0.1
$varnish = new Varnish($servers);
```

This is sufficient for invalidating absolute URLs. If you also wish to invalidate relative paths, supply the hostname (or base URL) where your website is available as the second parameter:

```
$varnish = new Varnish($servers, 'my-cool-app.com');
```

Again, if you access your web application on a port other than 80, make sure to include that port in the base URL:

```
$varnish = new Varnish($servers, 'my-cool-app.com:8080');
```

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

4.1.2 NGINX Client

At minimum, supply an array containing IPs or hostnames of the NGINX servers that you want to send invalidation requests to. Make sure to include the port NGINX runs on if it is not the default:

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

$servers = array('10.0.0.1', '10.0.0.2:8088'); // Port 80 assumed for 10.0.0.1
$nginx = new Nginx($servers);
```

This is sufficient for invalidating absolute URLs. If you also wish to invalidate relative paths, supply the hostname (or base URL) where your website is available as the second parameter:

```
$nginx = new Nginx($servers, 'my-cool-app.com');
```

If you have configured NGINX to support purge requests at a separate location, supply that location to the class as the third parameter:

```
$nginx = new Nginx($servers, 'my-cool-app.com', '/purge');
```

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

4.1.3 Symfony Client

At minimum, supply an array containing IPs or hostnames of your web servers running Symfony. Provide the direct access to the web server without any other proxies that might block invalidation requests. Make sure to include the port the web server runs on if it is not the default:

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

$servers = array('10.0.0.1', '10.0.0.2:8088'); // Port 80 assumed for 10.0.0.1
$client = new Symfony($servers);
```

This is sufficient for invalidating absolute URLs. If you also wish to invalidate relative paths, supply the hostname (or base URL) where your website is available as the second parameter:

```
$client = new Symfony($servers, 'my-cool-app.com');
```

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

4.2 Using the Clients

Each client is an implementation of ProxyClientInterface. All other interfaces, `PurgeInterface`, `RefreshInterface` and `BanInterface` extend this `ProxyClientInterface`. So each client implements at least one of the three *invalidation methods* depending on the caching proxy's abilities.

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

4.2.1 Supported invalidation methods

Client	Purge	Refresh	Ban
Varnish	✓	✓	✓
NGINX	✓	✓	
Symfony Cache	✓	✓	

4.2.2 Purge

If the caching proxy understands *purge* requests, its client should implement `PurgeInterface`. Use the `purge($url)` method to purge one specific URL. The URL can be either an absolute URL or a relative path:

```
$client
    ->purge('http://my-app.com/some/path')
    ->purge('/other/path')
    ->flush()
;
```

You can specify HTTP headers as the second argument to `purge()`. For instance:

```
$client
    ->purge('/some/path', array('X-Foo' => 'bar'))
    ->flush()
;
```

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

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

4.2.3 Refresh

If the caching proxy understands *refresh* requests, its client should implement `RefreshInterface`. Use `refresh()` to refresh one specific URL. The URL can be either an absolute URL or a relative path:

```
$client
    ->refresh('http://my-app.com/some/path')
    ->refresh('other/path')
    ->flush()
;
```

You can specify HTTP headers as the second argument to `refresh()`. For instance, to only refresh the JSON representation of an URL:

```
$client
    ->refresh('/some/path', array('Accept' => 'application/json')
    ->flush()
;
```

4.2.4 Ban

If the caching proxy understands *ban* requests, its client should implement `BanInterface`.

You can invalidate all URLs matching a regular expression by using the `banPath($path, $contentType, $hosts)` method. It accepts a regular expression for the path to invalidate and an optional content type regular expression and list of application hostnames.

For instance, to ban all `.png` files on all application hosts:

```
$client->banPath('.*png$');
```

To ban all HTML URLs that begin with `/articles/`:

```
$client->banPath('/articles/.*', 'text/html');
```

By default, URLs will be banned on all application hosts. You can limit this by specifying a host header:

```
$client->banPath('*.png$', null, '^www.example.com$');
```

If you want to go beyond banning combinations of path, content type and hostname, use the `ban(array $headers)` method. This method allows you to specify any combination of headers that should be banned. For instance, when using the Varnish client:

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

$varnish->ban(array(
    Varnish::HTTP_HEADER_URL   => '.*\.png$',
    Varnish::HTTP_HEADER_HOST  => '.*example\.com',
    Varnish::HTTP_HEADER_CACHE => 'my-tag',
));
```

Make sure to add any headers that you want to ban on to your *proxy configuration*.

4.3 Custom Guzzle Client

By default, the proxy clients instantiate a Guzzle client to communicate with the caching proxy. If you need to customize the requests, for example to send a basic authentication header, you can inject a custom Guzzle client:

```
use FOS\HttpCache\ProxyClient\Varnish;
use Guzzle\Http\Client;

$client = new Client();
$client->setDefaultOption('auth', array('username', 'password', 'Digest'));

$servers = array('10.0.0.1');
$varnish = new Varnish($servers, '/baseUrl', $client);
```

The Symfony client accepts a guzzle client as the 3rd parameter as well, NGINX accepts it as 4th parameter.

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.

5.1 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(...);

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

Note: See *proxy client setup* for more on constructing a client.

5.2 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',
    array('Cookie' => 'foo=bar; fizz=bang')
)->flush();
```

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

5.3 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',
    array('Cookie' => 'foo=bar; fizz=bang')
)->flush();
```

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

5.4 Invalidating With a Regular Expression

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

5.4.1 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 caching proxy:

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

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

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

5.4.2 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.

Cache 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 `X-My-Header`, you would do:

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

5.5 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.

5.6 Error handling

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

These exception are of two types:

- `\FOS\HttpCache\ProxyUnreachableException` when the client cannot connect to the caching proxy

- `\FOS\HttpCache\ProxyResponseException` when the caching proxy 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);
    }
}
```

5.6.1 Logging errors

You can log any exceptions in the following way. 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 subscriber to the cache invalidator:

```
use FOS\HttpCache\EventListener\LogSubscriber;

$subscriber = new LogSubscriber($monolog);
$cacheInvalidator->getEventDispatcher()->addSubscriber($subscriber);
```

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

Extra Invalidation Handlers

This library provides decorators that build on top of the `CacheInvalidator` to simplify common operations.

6.1 Tag Handler

New in version 1.3: The tag handler was added in FOSHttpCache 1.3. If you are using an older version of the library and can not update, you need to use `CacheInvalidator::invalidateTags`.

The tag handler helps you to mark responses with tags that you can later use to invalidate all cache entries with that tag. Tag invalidation works only with a `CacheInvalidator` that supports `CacheInvalidator::INVALIDATE`.

6.1.1 Setup

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

The tag handler is a decorator around the `CacheInvalidator`. After *creating the invalidator* with a proxy client that implements the `BanInterface`, instantiate the `TagHandler`:

```
use FOS\HttpCache\Handler\TagHandler;

// $cacheInvalidator already created as instance of FOS\HttpCache\CacheInvalidator
$tagHandler = new TagHandler($cacheInvalidator);
```

6.1.2 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 handler using:

```
$tagHandler->addTags(array('tag-two', 'group-a'));
```

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

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

Tip: If you are using Symfony with the FOSHttpCacheBundle, the tag header is set 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(array('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.

6.1.3 Custom Tags Header

Tagging uses a custom HTTP header to identify tags. You can change the default header `X-Cache-Tags` in the constructor:

```
use FOS\HttpCache\Handler\TagHandler;

// $cacheInvalidator already created as instance of FOS\HttpCache\CacheInvalidator
$tagHandler = new TagHandler($cacheInvalidator, 'My-Cache-Header');
```

Make sure to reflect this change in your *caching proxy configuration*.

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. Both approaches integrate with the concepts presented in this chapter.

7.1 Overview

Caching on user context works as follows:

1. A *client* requests `/foo.php` (the *original request*).
2. The *caching proxy* 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 caching proxy 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.

7.2 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*.

7.3 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.

7.3.1 Calculating the User Context Hash

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

```
use FOS\HttpCache\UserContext\HashGenerator;

$hashGenerator = new HashGenerator(array(
    new IsAuthenticatedProvider(),
    new RoleProvider(),
));
```

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

7.3.2 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\ContextProviderInterface;
use FOS\HttpCache\UserContext\UserContext;

class IsAuthenticatedProvider implements ContextProviderInterface
{
    protected $userService;

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

(continues on next page)

(continued from previous page)

```
    }

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

7.3.3 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.

7.3.4 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.

7.4 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*:

```php
// /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";
}
```

7.5 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.

The FOSHttpCache library provides base test case classes to help you write functional tests. This is helpful to test the way your application sets caching headers and invalidates cached content.

By having your test classes extend one of the test case classes, you get:

- independent tests: all previously cached content is removed in the tests `setUp` method. The way this is done depends on which reverse proxy you use;
- an instance of this library's client that is configured to talk to your reverse proxy server. See reverse proxy specific sections for details;
- convenience methods for executing HTTP requests to your application: `$this->getHttpClient()` and `$this->getResponse()`;
- custom assertions `assertHit` and `assertMiss` for validating a cache hit/miss.

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

You will need to run a web server to provide the PHP application you want to test. The test cases only handle running the caching proxy. With PHP 5.4 or newer, the easiest is to use the PHP built in web server. See the `WebServerListener` class in `tests/Functional` and how it is registered in `phpunit.xml.dist`.

8.1 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" />
```

(continues on next page)

(continued from previous page)

```
        <const name="WEB_SERVER_DOCROOT" value="./tests/Functional/Fixtures/web" />
    </php>
</phpunit>
```

8.2 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;
    }
}
```

8.2.1 VarnishTestCase

Configuration

By default, the `VarnishTestCase` starts and stops a Varnish server for you. Make sure `symfony/process` is available in your project:

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

Then set your Varnish configuration (VCL) file. All available configuration parameters are shown below.

Constant	Getter	Default	Description
`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
`VARNISH_VERSION`	`getVarnishVersion()`	`3`	installed varnish application version
`WEB_SERVER_HOSTNAME`	`getHostName()`		hostname your application can be reached at

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.

8.2.2 NginxTestCase

Configuration

By default, the `NginxTestCase` starts and stops the NGINX server for you and deletes all cached contents. Make sure `symfony/process` is available in your project:

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

You have to set your NGINX configuration file. All available configuration parameters are shown below.

Constant	Getter	Default	Description
`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.
`WEB_SERVER_HOSTNAME`	`getHostName()`		hostname your application can be reached at

Enable Assertions

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

8.2.3 SymfonyTestCase

This test case helps to test invalidation requests with a symfony application running the Symfony HttpCache and invalidating its cache folder to get reliable tests.

The `SymfonyTestCase` does automatically start a web server. It is assumed that the web server you run for the application has the HttpCache integrated.

Configuration

Constant	Getter	Default	Description
`WEB_SERVER_HOSTNAME`	`getHostName()`		hostname your application can be reached at
`WEB_SERVER_PORT`	`getConfigFile()`		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.

Enable Assertions

For the *assertHit* and *assertMiss* assertions to work, you need to add debug information in your AppCache. Create the cache kernel with the option `'debug' => true` and add the following to your `AppCache`:

```php
public function handle(Request $request, $type = HttpKernelInterface::MASTER_REQUEST,
↪$catch = true)
{
    $response = parent::handle($request, $type, $catch);

    if ($response->headers->has('X-Symfony-Cache')) {
        if (false !== strpos($response->headers->get('X-Symfony-Cache'), 'miss')) {
            $state = 'MISS';
        } elseif (false !== strpos($response->headers->get('X-Symfony-Cache'), 'fresh
↪')) {
            $state = 'HIT';
        } else {
            $state = 'UNDETERMINED';
        }
        $response->headers->set('X-Cache', $state);
    }

    return $response;
}
```

The UNDETERMINED state should never happen. If it does, it means that your HttpCache is not correctly set into debug mode.

8.2.4 Usage

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

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

class YourFunctionalTest extends VarnishTestCase
{
    public function testCachingHeaders()
    {
        // Varnish is restarted, so you don’t have to worry about previously
        // cached content. Before continuing, the VarnishTestCase waits for
        // Varnish to become available.

        // Retrieve an 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 Varnish):
        $response = $this->getResponse('/your/resource');
        $this->assertHit($response);
    }
}
```

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

```php
public function testCachePurge()
{
```

(continues on next page)

(continued from previous page)

```
    // Again, Varnish 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->varnish->purge('/blog/articles/1');

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

Tests for Nginx look the same but extend the NginxTestCase. For more ideas, see this library's functional tests in the tests/Functional/ directory.

CHAPTER 9

Contributing

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

9.1 Testing the Library

This chapter describes how to run the tests that are included with this library.

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
```

9.1.1 Unit Tests

To run the unit tests separately:

```
$ phpunit tests/Unit
```

9.1.2 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 PHP 5.3, you need to configure a web server listening on localhost:8080 that points to the folder `tests/Functional/Fixtures/web`.

If you want to run the tests on HHVM, you need to configure a web server 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*.

9.2 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