
FOSHttpCacheBundle Documentation

Release 2.0.0

David de Boer, David Buchmann

Jul 17, 2017

Contents

1	Contents	3
1.1	Overview	3
1.2	Features	5
1.3	Reference	17
1.4	Testing	44
1.5	Contributing	44

Note: This documentation is for the (upcoming) 2.0 of the library. For the stable 1.* version, please refer to the [stable documentation](#).

This is the documentation for the [FOSHttpCacheBundle](#). Use the FOSHttpCacheBundle to:

- Set path-based cache expiration headers via your application configuration;
- Set up an invalidation scheme without writing PHP code;
- Tag your responses and invalidate cache based on tags;
- Send invalidation requests with minimal impact on performance with the [FOSHttpCache](#) library;
- Differentiate caches based on user *type* (e.g. roles);
- Easily implement your own HTTP cache client.

Overview

Installation

This bundle is available on [Packagist](#). You can install it using Composer. Note that the `FOSHttpCache` library needs a `psr/http-message-implementation` and `php-http/client-implementation`. If your project does not contain one, composer will complain that it did not find `psr/http-message-implementation`.

To install the bundle together with Guzzle, run:

```
$ composer require friendsofsymfony/http-cache-bundle guzzlehttp/psr7 php-http/  
↳guzzle6-adapter
```

If you want to use something else than Guzzle 6, see [Packagist](#) for a list of available [client implementations](#).

Then add the bundle to your application:

```
<?php  
// app/AppKernel.php  
  
public function registerBundles()  
{  
    $bundles = array(  
        // ...  
        new FOS\HttpCacheBundle\FOSHttpCacheBundle(),  
        // ...  
    );  
}
```

For most features, you also need to [configure a caching proxy](#).

Requirements

SensioFrameworkExtraBundle

If you want to use this bundle's annotations, install the [SensioFrameworkExtraBundle](#):

```
$ composer require sensio/framework-extra-bundle
```

And include it in your project:

```
<?php
// app/AppKernel.php

public function registerBundles()
{
    $bundles = array(
        // ...
        new FOS\HttpCacheBundle\FOSHttpCacheBundle(),
        new Sensio\Bundle\FrameworkExtraBundle\SensioFrameworkExtraBundle(),
        // ...
    );
}
```

ExpressionLanguage

If you wish to use [expressions](#) in your annotations, you also need Symfony's [ExpressionLanguage](#) component. If you're not using full-stack Symfony 2.4 or later, you need to explicitly add the component:

```
$ composer require symfony/expression-language
```

Configuration

Now you can configure the bundle under the `fos_http_cache` key. The [Features](#) section introduces the bundle's features. The [Configuration](#) section lists all configuration options.

Functionality

This table shows where you can find specific functions.

Functionality	Annotations	Configuration	Manually
Set Cache-Control headers	(SensioFrameworkExtraBundle)	rules	(Symfony)
Tag and invalidate	@Tag	rules	cache manager
Invalidate routes	@InvalidateRoute	invalidators	cache manager
Invalidate paths	@InvalidatePath	invalidators	cache manager

License

This bundle is released under the MIT license.

```
Copyright (c) 2010-2015 Liip, http://www.liip.ch <contact@liip.ch>
Driebit, http://www.driebit.nl <info@driebit.nl>
```

```
Permission is hereby granted, free of charge, to any person obtaining a copy
of this software and associated documentation files (the "Software"), to deal
in the Software without restriction, including without limitation the rights
```


to use, copy, modify, merge, publish, distribute, sublicense, and/or sell copies of the Software, and to permit persons to whom the Software is furnished to do so, subject to the following conditions:

The above copyright notice and this permission notice shall be included in all copies or substantial portions of the Software.

THE SOFTWARE IS PROVIDED "AS IS", WITHOUT WARRANTY OF ANY KIND, EXPRESS OR IMPLIED, INCLUDING BUT NOT LIMITED TO THE WARRANTIES OF MERCHANTABILITY, FITNESS FOR A PARTICULAR PURPOSE AND NONINFRINGEMENT. IN NO EVENT SHALL THE AUTHORS OR COPYRIGHT HOLDERS BE LIABLE FOR ANY CLAIM, DAMAGES OR OTHER LIABILITY, WHETHER IN AN ACTION OF CONTRACT, TORT OR OTHERWISE, ARISING FROM, OUT OF OR IN CONNECTION WITH THE SOFTWARE OR THE USE OR OTHER DEALINGS IN THE SOFTWARE.

Features

This part introduces the bundle's features. Each feature section links to the corresponding reference section.

Caching Headers

Prerequisites: *None*

You can configure HTTP caching headers based on request and response properties. This configuration approach is more convenient than [manually setting cache headers](#) and an alternative to [setting caching headers through annotations](#).

Set caching headers under the `cache_control` configuration section, which consists of a set of rules. When the request matches all criteria under `match`, the headers under `headers` will be set on the response.

A Response may already have cache headers set, e.g. by the controller method. By default, the options that already exist are not overwritten, but additional headers are added. You can force to overwrite the headers globally by setting `cache_control.defaults.override: true` to `true`, or on a per rule basis with `override: true` under `headers`.

This is an example configuration. For more, see the [cache_control](#) configuration reference.

```
# app/config/config.yml
fos_http_cache:
  cache_control:
    defaults:
      overwrite: true
    rules:
      # only match login.example.com
      -
        match:
          host: ^login.example.com$
        headers:
          cache_control: { public: false, max_age: 0, s_maxage: 0 }
          etag: true
          vary: [Accept-Encoding, Accept-Language]

      # match all actions of a specific controller
      -
        match:
          attributes: { _controller: ^AcmeBundle:Default:.* }
```

```
headers:
  cache_control: { public: true, max_age: 15, s_maxage: 30 }
  last_modified: "-1 hour"
-
match:
  path: ^/$
headers:
  cache_control: { public: true, max_age: 64000, s_maxage: 64000 }
  etag: true
  vary: [Accept-Encoding, Accept-Language]

# match everything to set defaults
-
match:
  path: ^/
headers:
  overwrite: false
  cache_control: { public: true, max_age: 15, s_maxage: 30 }
  etag: true
```

Invalidation

Works with:

- Varnish
- Nginx (except regular expressions)
- *Symfony HttpCache* (except regular expressions)

Preparation:

In order to invalidate cached objects, requests are sent to your caching proxy, so first:

1. configure your proxy
2. *enable a proxy client*

By *invalidating* a piece of content, you tell your HTTP caching proxy (Varnish or Nginx) to no longer serve it to clients. When next requested, the proxy will fetch a fresh copy from the backend application and serve that instead. By *refreshing* a piece of content, a fresh copy will be fetched right away.

Tip: Invalidation can result in better performance compared to the validation caching model, but is more complex. Read the [Introduction to Cache Invalidation](#) of the FOSHttpCache documentation to learn about the differences and decide which model is right for you.

Cache Manager

To invalidate single paths, URLs and routes manually, use the `invalidatePath($path, $headers)` and `invalidateRoute($route, $params, $headers)` methods on the cache manager:

```
$cacheManager = $container->get('fos_http_cache.cache_manager');

// Invalidate a path
$cacheManager->invalidatePath('/users')->flush();
```

```
// Invalidate a URL
$cacheManager->invalidatePath('http://www.example.com/users')->flush();

// Invalidate a route
$cacheManager->invalidateRoute('user_details', array('id' => 123))->flush();

// Invalidate a route or path with headers
$cacheManager->invalidatePath('/users', array('X-Foo' => 'bar'))->flush();
$cacheManager->invalidateRoute('user_details', array('id' => 123), array('X-Foo' =>
↳ 'bar'))->flush();
```

To invalidate multiple representations matching a regular expression, call `invalidateRegex($path, $contentType, $hosts)`:

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

To refresh paths and routes, you can use `refreshPath($path, $headers)` and `refreshRoute($route, $params, $headers)` in a similar manner. See *The Cache Manager* for more information.

Tip: If you want to add a header (such as Authorization) to *all* invalidation requests, you can use a *custom HTTP client* instead.

Configuration

You can add invalidation rules to your application configuration:

```
# app/config/config.yml
fos_http_cache:
  invalidation:
    rules:
      -
        match:
          attributes:
            _route: "villain_edit|villain_delete"
        routes:
          villains_index: ~ # e.g., /villains
          villain_details: ~ # e.g., /villain/{id}
```

Now when a request to either route `villain_edit` or route `villain_delete` returns a successful response, both routes `villains_index` and `villain_details` will be purged. See the *invalidation* configuration reference.

Annotations

Set the `@InvalidatePath` and `@InvalidateRoute` annotations to trigger invalidation from your controllers:

```
use FOS\HttpCacheBundle\Configuration\InvalidatePath;

/**
 * @InvalidatePath("/articles")
 * @InvalidatePath("/articles/latest")
 * @InvalidateRoute("overview", params={"type" = "latest"})
 * @InvalidateRoute("detail", params={"id" = {"expression"="id"}})
```

```
*/  
public function editAction($id)  
{  
}
```

See the *Annotations* reference.

Console Commands

This bundle provides commands to trigger cache invalidation from the command line. You could also send invalidation requests with a command line tool like `curl` or, in the case of `varnish`, `varnishadm`. But the commands simplify the task and will automatically talk to all configured cache instances.

- `fos:httpcache:invalidate:path` accepts one or more paths and invalidates each of them. See *invalidatePath()*.
- `fos:httpcache:refresh:path` accepts one or more paths and refreshes each of them. See *refreshPath()* and *refreshRoute()*.
- `fos:httpcache:invalidate:regex` expects a regular expression and invalidates all cache entries matching that expression. To invalidate your entire cache, you can specify `.` (dot) which will match everything. See *invalidatePath()*.
- `fos:httpcache:invalidate:tag` accepts one or more tags and invalidates all cache entries matching any of those tags. See *Tagging*.

If you need more complex interaction with the cache manager, best write your own commands and use the *cache manager* to implement your specific logic.

Tagging

Works with:

- `Varnish`

If your application has many intricate relationships between cached items, which makes it complex to invalidate them by route, cache tagging will be useful. It helps you with invalidating many-to-many relationships between content items.

Cache tagging, or more precisely *Tagged Cache Invalidation*, a simpler version of *Linked Cache Invalidation (LCI)*, allows you to:

- assign tags to your applications's responses (e.g., `articles`, `article-42`)
- *invalidate the responses by tag* (e.g., invalidate all responses that are tagged `article-42`)

Basic Configuration

First *configure your proxy* for tagging. Then enable tagging in your application configuration:

```
fos_http_cache:  
  tags:  
    enabled: true
```

For more information, see *tags*.

Setting and Invalidating Tags

You can tag responses in different ways:

- From PHP code by using the response tagger to set tags and the cache manager to invalidate tags;
- Set tags from twig templates with a function;
- In project configuration or using annotations on controller actions.

You can add tags before the response object exists. The tags are automatically added to the response by a listener. The listener also detects pending tag invalidations and flushes them. As with other invalidation operations, tag invalidation requests are flushed to the caching proxy *after the response has been sent*.

Tagging from Code

To add tags to responses, inject the `ResponseTagger` (service `fos_http_cache.http.symfony_response_tagger`) and use `addTags($tags)` to add tags that will be set on the response:

```
use FOS\HttpCacheBundle\Http\SymfonyResponseTagger;

class NewsController
{
    /**
     * @var SymfonyResponseTagger
     */
    private $responseTagger;

    public function articleAction($id)
    {
        $this->responseTagger->addTags(array('news', 'news-' . $id));

        // ...
    }
}
```

To invalidate tags, inject the `CacheManager` (service `fos_http_cache.cache_manager`) and call `invalidateTags($tags)` on it:

```
use FOS\HttpCacheBundle\CacheManager;

class NewsController
{
    /**
     * @var CacheManager
     */
    private $cacheManager;

    public function editAction($id)
    {
        // ...

        $this->cacheManager->invalidateTags(array('news-' . $id));

        // ...
    }
}
```

Tagging from Twig Templates

In situations where a page is assembled in the templating layer, it can be more convenient to add tags from inside the template. This works the same way as with the response tagger and can also be mixed with the other methods:

```
{# template.html.twig #}
{{ fos_httpcache_tag('mytag') }}
{{ fos_httpcache_tag(['tag-one', 'tag-two']) }}
```

Hint: This twig function never outputs anything into the template but is only called for the side effect of adding the tag to the response header.

Note: Tag invalidation from twig would be a strange architecture and is therefore not supported.

Tagging with Configuration Rules

Alternatively, you can *configure rules* for setting and invalidating tags:

```
// app/config/config.yml
fos_http_cache:
  tags:
    rules:
      -
        match:
          path: ^/news/article
        tags: [news]
```

Now if a *safe* request matches the criteria under `match`, the response will be tagged with `news`. When an *unsafe* request matches, the tag `news` will be invalidated.

Tagging with Controller Annotations

Add the `@Tag` annotations to your controllers to set and invalidate tags:

```
use FOS\HttpCacheBundle\Configuration\Tag;

class NewsController
{
    /**
     * @Tag("news", expression="'news-~id'")
     */
    public function articleAction($id)
    {
        // Assume $id equals 123
    }
}
```

If `articleAction` handles a *safe* request, a tag `news-123` is set on the response. If a client tries to update or delete news article 123 with an *unsafe* request to `articleAction`, such as `POST` or `DELETE`, tag `news-123` is invalidated.

See the [@Tag reference](#) for full details.

User Context

Works with:

- [Varnish](#)
- [Symfony HttpCache](#)

If your application serves different content depending on the user's group or context (guest, editor, admin), you can cache that content per user context. Each user context (group) gets its own unique hash, which is then used to vary content on. The event listener responds to hash requests and sets the Vary header. This way, you can differentiate your content between user groups while not having to store caches for each individual user.

Note: Please read the [User Context](#) chapter in the FOSHttpCache documentation before continuing.

How It Works

These five steps resemble the Overview in the FOSHttpCache documentation.

1. A [client](#) requests `/foo`.
2. The [proxy server](#) receives the request and holds it. It first sends a *hash request* to the *context hash route*.
3. The [application](#) receives the hash request. An event listener (`UserContextListener`) aborts the request immediately after the Symfony firewall was applied. The application calculates the hash (`HashGenerator`) and then sends a response with the hash in a custom header (`X-User-Context-Hash` by default).
4. The caching proxy receives the hash response, copies the hash header to the client's original request for `/foo` and restarts that request.
5. If the response to `/foo` should differ per user context, the application sets a `Vary: X-User-Context-Hash` header. The appropriate user context dependent representation of `/foo` will then be returned to the client.

Note: In other words we execute a preflight request here. A preflight request is a request that is sent prior to the real request. See for example how [CORS requests](#) work. The result of the preflight request is the `X-User-Context-Hash` header that is added to the real request.

This concept can be generalized to handle more than the user context scenario. The [terminal42/header-replay-bundle](#) builds on top of the `FOSHttpCacheBundle` to add support for more complicated use cases. Have a look at the [HeaderReplayBundle documentation](#) if the user context is not flexible enough for your needs.

Configuration

First [configure your caching proxy](#). Then configure Symfony for handling hash lookups. The minimal steps are described below, see the [reference](#) for more details.

You need to configure a route for the context hash. It does not specify any controller, as the request listener will abort the request right after the firewall has been applied, but the route definition must exist. Use the same path as you specified in the caching proxy and make sure that this path is allowed for anonymous users and covered by your [firewall configuration](#):

```
# app/config/routing.yml
user_context_hash:
    path: /_fos_user_context_hash
```

If your access rules limit the whole site to logged in users, make sure to handle the user context URL like the login page:

```
# app/config/security.yml
access_control:
    - { path: ^/login, roles: IS_AUTHENTICATED_ANONYMOUSLY }
    - { path: ^/_fos_user_context_hash, roles: [IS_AUTHENTICATED_ANONYMOUSLY] }
    - { path: ^/, roles: ROLE_USER }
```

Finally, enable the listener with the default settings:

```
# app/config/config.yml
fos_http_cache:
    user_context:
        enabled: true
```

Note: When using the FOSRestBundle `format_listener` configuration on all paths of your site, the hash lookup will fail with “406 Not Acceptable - No matching accepted Response format could be determined”. To avoid this problem, you can add a rule to the format listener configuration:

```
- { path: '^/_fos_user_context_hash', stop: true }
```

Generating Hashes

When a context hash request is received, the `HashGenerator` is used to build the context information. The generator does so by calling on one or more *context providers*.

The bundle includes a simple `role_provider` that determines the hash from the user’s roles. To enable it:

```
# app/config/config.yml
fos_http_cache:
    user_context:
        role_provider: true
```

Alternatively, you can create a *custom context provider*.

Caching Hash Responses

To improve User Context Caching performance, you should cache the hash responses. You can do so by configuring `hash_cache_ttl`.

Helpers

Flash Message Listener

Prerequisites: *none*

When flash messages are rendered into the content of a page, you can’t cache the page anymore. When enabled, this listener reads all flash messages into a cookie, leading to them not being there anymore when rendering the template.

This will return the page with a set-cookie header which you of course must make sure to not cache in varnish. By default, varnish will simply not cache the whole response when there is a set-cookie header. (Maybe you could do something more clever — if you do, please provide a VCL example.)

The flash message listener is automatically enabled if you configure any of the options under `flash_message`.

```
# app/config.yml
fos_http_cache:
  flash_message:
    enabled: true
```

On the client side, you need some JavaScript code that reads out the flash messages from the cookie and writes them into the DOM, then deletes the cookie to only show the flash message once. Something along these lines:

```
function getCookie(cname)
{
  var name = cname + "=";
  var ca = document.cookie.split(';');
  for(var i=0; i<ca.length; i++) {
    var c = ca[i].trim();
    if (c.indexOf(name)==0) {
      return c.substring(name.length,c.length);
    }
  }
  return false;
}

function showFlash()
{
  var cookie = getCookie("flashes"); // fos_http_cache.flash_message.name

  if (!cookie) {
    return;
  }

  var flashes = JSON.parse(decodeURIComponent(cookie));

  var html = '';
  for (var key in flashes) {
    if (key === 'length' || !flashes.hasOwnProperty(key)) {
      continue;
    }
    html = '<div class="alert alert-' + key + '>';
    html += flashes[key];
    html += '</div>';
  }
  // YOUR WORK: show flashes in your DOM...

  // remove the cookie to not show flashes again
  // path is the fos_http_cache.flash_message.path value
  document.cookie = "flashes=; expires=Thu, 01 Jan 1970 00:00:01 GMT; path=/";
}

// YOUR WORK: register showFlash on the page ready event.
```

The parts about adding the flash messages in the DOM and registering your handler depend on the JavaScript framework you use in your page.

Your VCL configuration should filter out this cookie on subsequent requests, in case the JavaScript failed to remove it.

Testing

Works with:

- Varnish
- Nginx

Preparation:

1. Configure caching proxy
2. Your application must be reachable from the caching proxy through HTTP, so you need to have a web server running. If you already have a web server installed for development, you can use that. Alternatively, on PHP 5.4 or newer, you can use PHP's built-in web server, for instance through `FOS\HttpCache\Tests\Functional\WebServerListener`.

ProxyTestCase

If you wish to test your application caching and invalidation strategies against a live Varnish or Nginx instance, extend your test classes from `ProxyTestCase`. `ProxyTestCase` is an abstract base test class that in its turn extends Symfony's `WebTestCase`. It offers some convenience methods for cache testing:

```
class YourTest extends ProxyTestCase
{
    public function testCachingHeaders()
    {
        // Retrieve an URL from your application
        $response = $this->getResponse('/your/page');

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

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

Test Client

The `getResponse()` method calls `getHttpClient()` to retrieve a test client. You can use this client yourself to customize the requests. Note that the test client must be *enabled in your configuration*. By default, it is enabled when you access your application in debug mode and you have *configured a proxy client* with `base_url`.

Controlling Your Caching Proxy

You can also use `ProxyTestCase` to control your caching proxy. First configure the proxy server:

```
// app/config/config_test.yml
fos_http_cache:
  test:
    proxy_server:
      varnish:
        binary: /usr/sbin/varnishd
        port: 8080
        config_file: /etc/varnish/your-config.vcl
```

See also:

The complete reference for the configuration for testing is in the *test configuration* section.

The custom `@clearCache` PHPUnit annotation will start the proxy server (if it was not yet running) and clear any previously cached content. This enables you to write isolated test cases:

```
use FOS\HttpCacheBundle\Test\ProxyTestCase;

class YourTest extends ProxyTestCase
{
    /**
     * @clearCache
     */
    public function testMiss()
    {
        // We can be sure this is a miss, because even if the content was
        // cached before, it has been cleared from the caching proxy.
        $this->assertMiss($this->getResponse('/your/page'));
    }
}
```

You can annotate single test methods as well as classes with `@clearCache`. An annotated test class will restart and clear the caching proxy for each test case contained in the class.

You can also manually control your caching proxy:

```
use FOS\HttpCacheBundle\Test\ProxyTestCase;

class YourTest extends ProxyTestCase
{
    public function testMiss()
    {
        // Start caching proxy
        $this->getProxy()->start();

        // Clear proxy cache
        $this->getProxy()->clear();

        $this->assertMiss($this->getResponse('/your/page'));

        // Stop caching proxy
        $this->getProxy()->stop();
    }
}
```

Symfony HttpCache

Symfony comes with a built-in reverse proxy written in PHP, known as `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 (see the [Symfony HttpCache documentation](#)).

You can use features of this library with the `Symfony HttpCache`. The basic concept is to use event listeners on the `HttpCache` class.

Warning: `Symfony HttpCache` support is currently limited to following features:

- Purge
- Refresh
- User context

Extending the correct `HttpCache`

Instead of extending `Symfony\Bundle\FrameworkBundle\HttpCache\HttpCache`, your `AppCache` should extend `FOS\HttpCacheBundle\SymfonyCache\EventDispatchingHttpCache`:

```
require_once __DIR__.'/AppKernel.php';

use FOS\HttpCacheBundle\SymfonyCache\EventDispatchingHttpCache;

class AppCache extends 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. The drawback is that you need to manually check whether you need to adjust your `AppCache` each time you update the `FOSHttpCache` library.

By default, the event dispatching cache kernel registers all event listeners it knows about. You can disable listeners, or customize how they are instantiated.

If you do not need all listeners, or need to register some yourself to customize their behavior, overwrite `getOptions` and return the right bitmap in `fos_default_subscribers`. Use the constants provided by the `EventDispatchingHttpCache`:

```
public function getOptions()
{
    return array(
        'fos_default_subscribers' => self::SUBSCRIBER_NONE,
    );
}
```

To register event listeners that you want to instantiate yourself in the cache kernel, overwrite `getDefaultSubscribers`:

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

// ...
```

```

public function getDefaultSubscribers()
{
    // get enabled listeners with default settings
    $subscribers = parent::getDefaultSubscribers();

    $subscribers[] = new UserContextListener(array(
        'session_name_prefix' => 'eZSESSID',
    ));

    $subscribers[] = new CustomListener();

    return $subscribers;
}

```

You can also register event listeners from outside the kernel, e.g. in your `app.php` with the `addListener` and `addSubscriber` methods.

Warning: Since Symfony 2.8, the class cache (`classes.php`) is compiled even in console mode by an optional warmer (`ClassCacheCacheWarmer`). This can produce conflicting results with the regular web entry points, because the class cache may contain definitions (such as the subscribers above) that are loaded before the class cache itself; leading to redeclaration fatal errors.

There are two workarounds:

- Disable class cache warming in console mode with e.g. a compiler pass:

```

$container->getDefinition('kernel.class_cache.cache_warmer')->clearTag('kernel.
↪cache_warmer');

```

- Force loading of all classes and interfaces used by the `HttpCache` in `app/console` to make the class cache omit those classes. The simplest way to achieve this is to call `class_exists` resp. `interface_exists` with each of them.

Event Listeners

Each cache feature has its own event listener. The listeners are provided by the `FOSHttpCache` library. You can find the documentation for those listeners in the [FOSHttpCache Symfony Cache documentation section](#).

Reference

This part is a full description of all available configuration options, annotations and public methods.

Configuration

The configuration reference describes all `app/config/config.yml` options for the bundle.

proxy_client

The proxy client sends invalidation requests to your caching proxy. The *Cache Manager* wraps the proxy client and is the usual entry point for application interaction with the caching proxy.

You need to configure a client or define your own service for the cache manager to work.

The proxy client is also directly available as a service (`fos_http_cache.proxy_client.default` and `fos_http_cache.proxy_client.varnish`, `fos_http_cache.proxy_client.nginx` or `fos_http_cache.proxy_client.symfony`) that you can use directly.

If you need to adjust the proxy client, you can also configure the `CacheManager` with a *custom proxy client* that you defined as a service. In that case, you do not need to configure anything in the `proxy_client` configuration section.

varnish

```
# app/config/config.yml
fos_http_cache:
  proxy_client:
    varnish:
      tags_header: My-Cache-Tags
      header_length: 1234
      default_ban_headers:
        Foo: Bar
      http:
        servers:
          - 123.123.123.1:6060
          - 123.123.123.2
      base_url: yourwebsite.com
```

tags_header

type: string **default:** X-Cache-Tags

Header for sending tag invalidation requests to Varnish.

header_length

type: integer **default:** 7500

Maximum header length when invalidating tags. If there are more tags to invalidate than fit into the header, the invalidation request is split into multiple requests.

default_ban_headers

type: array

Map of header name header value that have to be set on each ban request. This list is merged with the built-in headers for bans.

http.servers

type: array

Comma-separated list of IP addresses or host names of your caching proxy servers. The port those servers will be contacted defaults to 80; you can specify a different port with `:<port>`.

When using a multi-server setup, make sure to include **all** proxy servers in this list. Invalidation must happen on all systems or you will end up with inconsistent caches.

http.base_url

type: string

The hostname (or base URL) where users access your web application. The base URL may contain a path. If you access your web application on a port other than 80, include that port:

```
# app/config/config.yml
fos_http_cache:
  proxy_client:
    varnish:
      http:
        base_url: yourwebsite.com:8000
```

Warning: Double-check `base_url`, for if it is mistyped, no content will be invalidated.

See the [FOSHttpCache library docs](#) on how to configure Varnish.

nginx

```
# app/config/config.yml
fos_http_cache:
  proxy_client:
    nginx:
      purge_location: /purge
      http:
        servers:
          - 123.123.123.1:6060
          - 123.123.123.2
        base_url: yourwebsite.com
```

For servers and `base_url`, see above.

purge_location

type: string

Separate location that purge requests will be sent to.

See the [FOSHttpCache library docs](#) on how to configure Nginx.

symfony

```
# app/config/config.yml
fos_http_cache:
  proxy_client:
    symfony:
      http:
```

```
servers:
  - 123.123.123.1:6060
  - 123.123.123.2
base_url: yourwebsite.com
```

For servers and base_url, see above.

noop

```
# app/config/config.yml
fos_http_cache:
  proxy_client:
    noop: ~
```

default

type: enum **options:** varnish, nginx, symfony

```
# app/config/config.yml
fos_http_cache:
  proxy_client:
    default: varnish
```

If there is only one proxy client, it is automatically the default. Only configure this if you configured more than one proxy client.

The default proxy client that will be used by the cache manager. You can *configure Nginx, Varnish and Symfony proxy clients in parallel*. There is however only one cache manager and it will only use the default client.

Custom HTTP Client

The proxy client uses a `Http\Client\Utils\HttpMethodsClient` wrapping a `Http\Client\HttpClient` instance. If you need to customize the requests, for example to send a basic authentication header with each request, you can configure a service for the `HttpClient` and specify that in the `http_client` option of any of the cache proxy clients.

Caching Proxy Configuration

You need to configure your caching proxy (Varnish or Nginx) to work with this bundle. Please refer to the [FOSHttpCache library's documentation](#) for more information.

cache_manager

The cache manager is the primary interface to invalidate caches. It is enabled by default if a *Proxy Client* is configured or when you specify the `custom_proxy_client` field.

```
# app/config/config.yml
fos_http_cache:
  cache_manager:
    enabled: true
```



```
custom_proxy_client: ~
generate_url_type: true
```

enabled**type:** enum **options:** auto, true, false

Whether the cache manager service should be enabled. By default, it is enabled if a proxy client is configured. It can not be enabled without a proxy client.

custom_proxy_client**type:** string

Instead of configuring a *Proxy Client*, you can define your own service that implements `FOS\HttpCache\ProxyClient`.

```
# app/config/config.yml
fos_http_cache:
  cache_manager:
    custom_proxy_client: acme.caching.proxy_client
```

When you specify a custom proxy client, the bundle does not know about the capabilities of the client. The `generate_url_type` defaults to true and *tag support* is only active if explicitly enabled.

generate_url_type**type:** enum **Symfony 2 options:** auto or one of the constants in `UrlGeneratorInterface`

The `$referenceType` to be used when generating URLs in the `invalidateRoute()` and `refreshRoute()` calls. If you use `ABSOLUTE_PATH` to only generate paths, you need to configure the `base_url` on the proxy client. When set to `auto`, the value is determined based on whether `base_url` is set on the default proxy client.

cache_control

The configuration contains a number of *rules*. When a request matches the parameters described in the `match` section, the headers as defined under `headers` will be set on the response, if they are not already set. Rules are checked in the order specified, where the first match wins.

A global setting and a per rule `overwrite` option allow to overwrite the cache headers even if they are already set:

```
# app/config/config.yml
fos_http_cache:
  cache_control:
    defaults:
      overwrite: false
    rules:
      # only match login.example.com
      -
        match:
          host: ^login.example.com$
        headers:
          overwrite: true
```

```
        cache_control:
          public: false
          max_age: 0
          s_maxage: 0
        etag: true
        vary: [Accept-Encoding, Accept-Language]

    # match all actions of a specific controller
    -
      match:
        attributes: { _controller: ^
↪Acme\\TestBundle\\Controller\\DefaultController::.* }
      headers:
        cache_control:
          public: true
          max_age: 15
          s_maxage: 30
          last_modified: "-1 hour"

    -
      match:
        path: ^/$
      headers:
        cache_control:
          public: true
          max_age: 64000
          s_maxage: 64000
        etag: true
        vary: [Accept-Encoding, Accept-Language]

    # match everything to set defaults
    -
      match:
        path: ^/
        additional_response_status:
          - 500
      headers:
        cache_control:
          public: true
          max_age: 15
          s_maxage: 30
        etag: true
```

rules

type: array

A set of cache control rules consisting of *match* criteria and *header* instructions.

match

type: array

A match definition that when met, will execute the rule effect. See *match*.

The cache control rules additionally allow to overwrite the global *cacheable* configuration for a specific rule. Put the `match_response` (semantics of `response.expression`) or `additional_response_status` setting under `match` with the same semantics as explained in *cacheable*.

Be aware that `additional_response_status` completely replaces `cacheable.response.additional_status`. There is *no* merge taking place.

headers

type: array

YAML alias for same headers for different matches

If you have many rules that should end up with the same headers, you can use YAML “aliases” *within the same configuration file* to avoid redundant configuration. The `&alias` notation creates an alias, the `<< : *alias` notation inserts the aliased configuration. You can then still overwrite parts of the aliased configuration. An example would be:

```
rules:
  -
    match:
      path: ^/products.*
    headers: &public
    cache_control:
      public: true
      max_age: 600
      s_maxage: 300
      reverse_proxy_ttl: 3600
  -
    match:
      path: ^/brands.*
    headers:
      << : *public
    cache_control:
      max_age: 1800
```

In the `headers` section, you define what headers to set on the response if the request was matched.

Headers are **merged**. If the response already has certain cache directives set, they are not overwritten. The configuration can thus specify defaults that may be changed by controllers or services that handle the response, or `@Cache` annotations.

The listener that applies the rules is triggered at priority 10, which makes it handle before the `@Cache` annotations from the `SensioFrameworkExtraBundle` are evaluated. Those annotations unconditionally overwrite cache directives.

The only exception is responses that *only* have the `no-cache` directive. This is the default value for the cache control and there is no way to determine if it was manually set. If the full header is only `no-cache`, the whole cache control is overwritten.

You can prevent the cache control on specific requests by injecting the service `fos_http_cache.event_listener.cache_control` and calling `setSkip()` on it. If this method is called, no cache rules are applied.

cache_control

type: array

The map under `cache_control` is set in a call to `Response::setCache()`. The names are specified with underscores in yaml, but translated to `-` for the `Cache-Control` header.

You can use the standard cache control directives:

- `max_age` time in seconds;
- `s_maxage` time in seconds for proxy caches (also public caches);
- `private` true or false;
- `public` true or false;
- `no_cache` true or false (use exclusively to support HTTP 1.0);
- `no_store`: true or false.

```
# app/config/config.yml
fos_http_cache:
  cache_control:
    rules:
      -
        headers:
          cache_control:
            public: true
            max_age: 64000
            s_maxage: 64000
```

If you use `no_cache`, you should *not set any other options*. This will make Symfony properly handle HTTP 1.0, setting the `Pragma: no-cache` and `Expires: -1` headers. If you add other `cache_control` options, Symfony will not do this handling. Note that Varnish 3 does not respect `no-cache` by default. If you want it respected, add your own logic to `vcl_fetch`.

Note: The cache-control headers are described in detail in [RFC 2616#section-14.9](#) and further clarified in [RFC 7234#section-5.2](#).

Extra Cache Control Directives

You can also set headers that Symfony considers non-standard, some coming from RFCs extending [RFC 2616](#) HTTP/1.1. The following options are supported:

- `must_revalidate` ([RFC 7234#section-5.2.2.1](#))
- `proxy_revalidate` ([RFC 7234#section-5.2.2.7](#))
- `no_transform` ([RFC 7234#section-5.2.2.4](#))
- `stale_if_error`: seconds ([RFC 5861#section-4](#))
- `stale_while_revalidate`: seconds ([RFC 5861#section-3](#))

The *stale* directives need a parameter specifying the time in seconds how long a cache is allowed to continue serving stale content if needed. The other directives are flags that are included when set to true:

```
# app/config/config.yml
fos_http_cache:
  cache_control:
    rules:
      -
        path: ^/$
        headers:
          cache_control:
            stale_while_revalidate: 9000
            stale_if_error: 3000
            must_revalidate: true
            proxy_revalidate: true
            no_transform: true
```

etag

type: boolean

This enables a simplistic ETag calculated as md5 hash of the response body:

```
# app/config/config.yml
fos_http_cache:
  cache_control:
    rules:
      -
        headers:
          etag: true
```

Tip: This simplistic ETag handler will not help you to prevent unnecessary work on your web server, but allows a caching proxy to use the ETag cache validation method to preserve bandwidth. The presence of an ETag tells clients that they can send a `If-None-Match` header with the ETag their current version of the content has. If the caching proxy still has the same ETag, it responds with a “304 Not Modified” status.

You can get additional performance if you write your own ETag handler that can read an ETag from your content and decide very early in the request whether the ETag changed or not. It can then terminate the request early with an empty “304 Not Modified” response. This avoids rendering the whole page. If the page depends on permissions, make sure to make the ETag differ based on those permissions (e.g. by appending the *user context hash*).

last_modified

type: string

The input to the `last_modified` is used for the `Last-Modified` header. This value must be a valid input to `DateTime`:

```
# app/config/config.yml
fos_http_cache:
  cache_control:
    rules:
      -
        headers:
          last_modified: "-1 hour"
```

Note: Setting an arbitrary last modified time allows clients to send `If-Modified-Since` requests. Varnish can handle these to serve data from the cache if it was not invalidated since the client requested it.

Note that the default system will generate an arbitrary last modified date. You can get additional performance if you write your own last modified handler that can compare this date with information about the content of your page and decide early in the request whether anything changed. It can then terminate the request early with an empty “304 Not Modified” response. Using content meta data increases the probability for a 304 response and avoids rendering the whole page.

See also [RFC 7232#section-2.2.1](#) for further consideration on how to generate the last modified date.

Note: You may configure both ETag and last modified on the same response. See [RFC 7232#section-2.4](#) for more details.

vary

type: string

You can set the `vary` option to an array that defines the contents of the `Vary` header when matching the request. This adds to existing `Vary` headers, keeping previously set `Vary` options:

```
# app/config/config.yml
fos_http_cache:
  cache_control:
    rules:
      -
        headers:
          vary: My-Custom-Header
```

reverse_proxy_ttl

type: integer

Set a `X-Reverse-Proxy-TTL` header for reverse proxy time-outs not driven by `s-maxage`. This keeps your `s-maxage` free for use with reverse proxies not under your control.

Warning: This is a custom header. You need to set up your caching proxy to respect this header. See the [FOSHttpCache documentation for Varnish](#) or [for the Symfony HttpCache](#).

To use the custom TTL, specify the option `reverse_proxy_ttl` in the headers section:

```
# app/config/config.yml
fos_http_cache:
  cache_control:
    rules:
      -
        headers:
          reverse_proxy_ttl: 3600
          cache_control:
```

```
public: true
s_maxage: 60
```

This example adds the header `X-Reverse-Proxy-TTL: 3600` to your responses.

invalidation

Configure invalidation to invalidate routes when some other routes are requested.

```
# app/config/config.yml
fos_http_cache:
  invalidation:
    enabled: true      # Defaults to 'auto'
    rules:
      -
        match:
          attributes:
            _route: "villain_edit|villain_delete"
        routes:
          villains_index: ~      # e.g., /villains
          villain_details:      # e.g., /villain/{id}
          ignore_extra_params: false # Defaults to true
```

enabled

type: enum, **default:** auto, **options:** true, false, auto

Enabled by default if you have configured the cache manager with *a proxy client*.

expression_language

type: string

If your application is using a [custom expression language](#) which is extended from Symfony's [expression language component](#), you can define it as a [service](#) and include it in the configuration.

Your custom expression functions can then be used in the `@InvalidateRoute` [annotations](#).

```
# app/config/config.yml
fos_http_cache:
  invalidation:
    expression_language: app.expression_language
```

rules

type: array

A set of invalidation rules. Each rule consists of a match definition and one or more routes that will be invalidated. Rules are checked in the order specified, where the first match wins. The routes are invalidated when:

1. the HTTP request matches *all* criteria defined under `match`
2. the HTTP response is successful.

match

type: array

A match definition that when met, will execute the rule effect. See [match](#).

routes

type: array

A list of route names that will be invalidated.

ignore_extra_params

type: boolean **default:** true

Parameters from the request are mapped by name onto the route to be invalidated. By default, any request parameters that are not part of the invalidated route are ignored. Set `ignore_extra_params` to `false` to set those parameters anyway.

A more detailed explanation: assume route `villain_edit` resolves to `/villain/{id}/edit`. When a client successfully edits the details for villain with id 123 (at `/villain/123/edit`), the index of villains (at `/villains`) can be invalidated (purged) without trouble. But which villain details page should we purge? The current request parameters are automatically matched against invalidate route parameters of the same name. In the request to `/villain/123/edit`, the value of the `id` parameter is 123. This value is then used as the value for the `id` parameter of the `villain_details` route. In the end, the page `villain/123` will be purged.

tags

Create tag rules in your application configuration to set tags on responses and invalidate them. See the [tagging feature chapter](#) for an introduction.

enabled

type: enum, **default:** auto, **options:** true, false, auto

Enabled by default if you have configured the cache manager with [a proxy client](#).

If you use a [proxy client that does not support banning](#), cache tagging is not possible. If you leave `enabled` on `auto`, tagging will be deactivated.

Enables tag annotations and rules. If you want to use tagging, it is recommended that you set this to `true` so you are notified of missing dependencies and incompatible proxies:

```
# app/config/config.yml
fos_http_cache:
  tags:
    enabled: true
```


header

type: string **default:** X-Cache-Tags

Custom HTTP header that tags are stored in.

expression_language

type: string

If your application is using a [custom expression language](#) which is extended from Symfony's [expression language component](#), you can [define it as a service](#) and include it in the configuration.

Your custom expression functions can then be used in both the `tag_expressions` section of the tag configuration and `@tag annotations`.

```
# app/config/config.yml
fos_http_cache:
  tags:
    expression_language: app.expression_language
```

strict

type: boolean **default:** false

Set this to `true` to throw an exception when an empty or null tag is added.

```
# app/config/config.yml
fos_http_cache:
  tags:
    strict: true
```

rules

type: array

Write your tagging rules by combining a `match` definition with a `tags` array. Rules are checked in the order specified, where the first match wins. These tags will be set on the response when all of the following are true:

1. the HTTP request matches *all* criteria defined under `match`
2. the HTTP request is *safe* (GET or HEAD)
3. the HTTP response is considered *cacheable*

When the definition matches an unsafe request (so 2 is false), the tags will be invalidated instead.

match

type: array

A match definition that when met, will execute the rule effect. See [match](#).

tags

type: array

Tags that should be set on responses to safe requests; or invalidated for unsafe requests.

```
# app/config/config.yml
fos_http_cache:
  tags:
    rules:
      -
        match:
          path: ^/news
        tags: [news-section]
```

tag_expressions

type: array

You can dynamically refer to request attributes using *expressions*. Assume a route `/articles/{id}`. A request to path `/articles/123` will set/invalidate tag `articles-123` with the following configuration:

```
# app/config/config.yml
fos_http_cache:
  tags:
    rules:
      -
        match:
          path: ^/articles
        tags: [articles]
        tag_expressions: ["'article-~id'"]
```

The expression has access to all request attributes and the request itself under the name `request`.

You can combine `tags` and `tag_expression` in one rule.

user_context

This chapter describes how to configure user context caching. See the *User Context Feature chapter* for an introduction to the subject.

Configuration

Caching Proxy Configuration

Varnish

Set up Varnish caching proxy as explained in the [user context documentation](#).

Symfony reverse proxy

Set up Symfony reverse proxy as explained in the [Symfony HttpCache documentation](#).

Context Hash Route

Then add the route you specified in the hash lookup request to the Symfony routing configuration, so that the user context event subscriber can get triggered:

```
# app/config/routing.yml
user_context_hash:
    path: /_fos_user_context_hash
```

Important: If you are using [Symfony security](#) for the hash generation, make sure that this route is inside the firewall for which you are doing the cache groups.

Note: This route is never actually used, as the context event subscriber will act before a controller would be called. But the user context is handled only after security happened. Security in turn only happens after the routing. If the routing does not find a route, the request is aborted with a 'not found' error and the listener is never triggered.

The event subscriber has priority 7 which makes it act right after the security listener which has priority 8. The reason to use a listener here rather than a controller is that many expensive operations happen later in the handling of the request. Having this listener avoids those.

enabled

type: enum **default:** auto **options:** true, false, auto

Set to `true` to explicitly enable the subscriber. The subscriber is automatically enabled if you configure any of the `user_context` options.

```
# app/config/config.yml
fos_http_cache:
    user_context:
        enabled: true
```

hash_header

type: string **default:** X-User-Context-Hash

The name of the HTTP header that the event subscriber will store the context hash in when responding to hash requests. Every other response will vary on this header.

match

accept

type: string **default:** application/vnd.fos.user-context-hash

HTTP Accept header that hash requests use to get the context hash. This must correspond to your caching proxy configuration.

method

type: string

HTTP method used by context hash requests, most probably either GET or HEAD. This must correspond to your caching proxy configuration.

matcher_service

type: string **default:** fos_http_cache.user_context.request_matcher

Id of a service that determines whether a request is a context hash request. The service must implement `Symfony\Component\HttpFoundation\RequestMatcherInterface`. If set, `accept` and `method` will be ignored.

hash_cache_ttl

type: integer **default:** 0

Time in seconds that context hash responses will be cached. Value 0 means caching is disabled. For performance reasons, it makes sense to cache the hash generation response; after all, each content request may trigger a hash request. However, when you decide to cache hash responses, you must invalidate them when the user context changes, particularly when the user logs in or out. This bundle provides a logout handler that takes care of this for you.

always_vary_on_context_hash

type: boolean **default:** true

This bundle automatically adds the Vary header for the user context hash, so you don't need to do this yourself or *configure it as header*. If the hash header is missing from a request for some reason, the response is set to vary on the user identifier headers to avoid problems.

If not all your pages depend on the hash, you can set `always_vary_on_context_hash` to `false` and handle the Vary yourself. When doing that, you should be careful to set the Vary header whenever needed, or you will end up with mixed up caches.

logout_handler

The logout handler will invalidate any cached user hashes when the user logs out.

For the handler to work:

- your caching proxy should be [configured for BANs](#)
- Symfony's default behavior of regenerating the session id when users log in and out must be enabled (`invalidate_session`).

Add the handler to your firewall configuration:

```
# app/config/security.yml
security:
  firewalls:
    secured_area:
      logout:
        invalidate_session: true
```

```
handlers:
  - fos_http_cache.user_context.logout_handler
```

enabled

type: enum **default:** auto **options:** true, false, auto

Defaults to `auto`, which enables the logout handler service if a *proxy client* is configured. Set to `true` to explicitly enable the logout handler. This will throw an exception if no proxy client is configured.

user_identifier_headers

type: array **default:** ['Cookie', 'Authorization']

Determines which HTTP request headers the context hash responses will vary on.

If the hash only depends on the `Authorization` header and should be cached for 15 minutes, configure:

```
# app/config/config.yml
fos_http_cache:
  user_context:
    user_identifier_headers:
      - Authorization
    hash_cache_ttl: 900
```

role_provider

type: boolean **default:** false

One of the most common scenarios is to differentiate the content based on the roles of the user. Set `role_provider` to `true` to determine the hash from the user's roles. If there is a security context that can provide the roles, all roles are added to the hash:

```
# app/config/config.yml
fos_http_cache
  user_context:
    role_provider: true
```

Custom Context Providers

Custom providers need to:

- implement the `FOS\HttpCache\UserContext\ContextProvider` interface
- be tagged with `fos_http_cache.user_context_provider`.

If you need context providers to run in a specific order, you can specify the optional `priority` parameter for the tag. The higher the priority, the earlier a context provider is executed. The build-in provider has a priority of 0.

The `updateUserContext (UserContext $context)` method of the context provider is called when the hash is generated.

```
acme.demo_bundle.my_service:
  class: "%acme.demo_bundle.my_service.class%"
  tags:
    - { name: fos_http_cache.user_context_provider, priority: 10 }
```

```
<service id="acme.demo_bundle.my_service" class="%acme.demo_bundle.my_service.class%">
  <tag name="fos_http_cache.user_context_provider" priority="10" />
</service>
```

```
$container
->register('acme.demo_bundle.my_service', '%acme.demo_bundle.my_service.class%')
->addTag('fos_http_cache.user_context_provider', array('priority' => 10))
;
```

Flash Message Configuration

The *flash message listener* is a tool to avoid rendering the flash message into the content of a page. It is another building brick for caching pages for logged in users.

```
# app/config/config.yml
fos_http_cache:
  flash_message:
    enabled: true
    name: flashes
    path: /
    host: null
    secure: false
```

enabled

type: boolean **default:** false

This event subscriber is disabled by default. You can set enabled to true if the default values for all options are good for you. When you configure any of the options, the subscriber is automatically enabled.

name

type: string **default:** flashes Set the name of the cookie.

path

type: string **default:** /

The cookie path to use.

host

type: string

Set the host for the cookie, e.g. to share among subdomains.

secure**type:** boolean **default:** false

Whether the cookie may only be passed through HTTPS.

debug

Enable the `debug` parameter to set a custom header (`X-Cache-Debug`) header on each response. You can then configure your caching proxy to add debug information when that header is present:

```
# app/config/config.yml
fos_http_cache:
  debug:
    enabled: true
    header: Please-Send-Debug-Infos
```

enabled**type:** enum **default:** auto **options:** true, false, autoThe default value is `%kernel.debug%`, triggering the header when you are in dev mode but not in prod mode.**header****type:** string **default:** X-Cache-Debug

Custom HTTP header that triggers the caching proxy to set debugging information on the response.

cacheable**response**

Configure which responses are considered *cacheable*. This bundle will only set Cache-Control headers, including tags etc., on cacheable responses.

You can only set one of `expression` or `additional_status`.**additional_status****type:** array

Following [RFC 7231](#), by default responses are considered *cacheable* if they have status code 200, 203, 204, 206, 300, 301, 404, 405, 410, 414 or 501. You can add status codes to this list by setting `additional_status`:

```
# app/config/config.yml
fos_http_cache:
  cacheable:
    response:
      additional_status:
        - 100
        - 500
```

expression

type: string

An ExpressionLanguage expression to decide whether the response is considered cacheable. The expression can access the Response object with the response variable:

```
# app/config/config.yml
fos_http_cache:
  cacheable:
    response:
      expression: "response.getStatusCode() >= 300"
```

When you configure an expression, *only* the expression is used to decide whether the response is cacheable. The default status codes from RFC 7231 are ignored when specifying an expression.

match

The *cache*, *invalidation* and *tag rule* configurations all use `match` sections to limit the configuration to specific requests and responses.

Each `match` section contains one or more match criteria for requests. All criteria are regular expressions. For instance:

```
match:
  host: ^login.example.com$
  path: ^/$
```

host

type: string

A regular expression to limit the caching rules to specific hosts, when you serve more than one host from your Symfony application.

Tip: To simplify caching of a site that offers front-end editing, put the editing on a separate (sub-)domain. Then define a first rule matching that domain with `host` and set `max-age: 0` to make sure your caching proxy never caches the editing domain.

path

type: string

For example, `path: ^/` will match every request. To only match the home page, use `path: ^/$`.

methods

type: array

Can be used to limit caching rules to specific HTTP methods like GET requests. Note that the rule effect is not applied to *unsafe* methods, not even when you set the methods here:


```
match:
  methods: [PUT, DELETE]
```

ips**type:** array

An array that can be used to limit the rules to a specified set of request client IP addresses.

Note: If you use a caching proxy and want specific IPs to see different headers, you need to forward the client IP to the backend. Otherwise, the backend only sees the caching proxy IP. See [Trusting Proxies](#) in the Symfony documentation.

attributes**type:** array

An array of request attributes to match against. Each attribute is interpreted as a regular expression.

_controller**type:** string

Controller name regular expression. Note that this is the controller name used in the route, so it depends on your route configuration whether you need `Acme\TestBundle\Controller\NameController::hello` or `acme_test.controller.name:helloAction` for [controllers as services](#).

Warning: Symfony always expands the short notation in route definitions. Even if you define your route as `AcmeTestBundle::Name::hello` you still need to use the long form here. If you use a service however, the compiled route still uses the service name and you need to match on that. If you mixed both, you can do a regular expression like `^(Acme\TestBundle|acme_test.controller)`.

_route**type:** string

Route name regular expression. To match a single route:

```
match:
  attributes:
    _route: ^articles_index$
```

To match multiple routes:

```
match:
  attributes:
    _route: ^articles.*|news$
```

Note that even for the request attributes, your criteria are interpreted as regular expressions.

```
match:
  attributes: { _controller: ^AcmeBundle:Default:.* }
```

test

Configures a proxy server and test client that can be used when *testing your application against a caching proxy*.

```
// app/config/config_test.yml
fos_http_cache:
  test:
    proxy_server:
      varnish:
        config_file: /etc/varnish/your-config.vcl
        port: 8080
        binary: /usr/sbin/varnish
    client:
      varnish:
        enabled: true
      nginx:
        enabled: false
```

proxy_server

Configures a service that can be used to start, stop and clear your caching proxy from PHP. This service is meant to be used in integration tests; don't use it in production mode.

varnish

config_file

type: string **required**

Path to a VCL file. For example Varnish configurations, see [Proxy Server Configuration](#).

binary

type: string **default:** varnishd

Path to the proxy binary (if the binary is named differently or not available in your PATH).

port

type: integer **default:** 6181

Port the caching proxy server listens on.

ip**type:** string **default:** 127.0.0.1

IP the caching proxy server runs on.

nginx**config_file****type:** string **required**Path to an Nginx configuration file. For an example Nginx configuration, see [Proxy Server Configuration](#).**binary****type:** string **default:** nginx

Path to the proxy binary.

port**type:** integer **default:** 8080

Port the caching proxy server listens on.

ip**type:** string **default:** 127.0.0.1

IP the caching proxy server runs on.

clientConfigures the *proxy test client* for Varnish and/or Nginx.**type:** array**enabled****type:** enum **default:** auto **options:** true, false, autoThe default value is `%kernel.debug%`, enabling the client when you are in `test` or `dev` mode but not in `prod` mode.**cache_header****type:** string **default:** X-Cache

HTTP header that shows whether the response was a cache hit (HIT) or a miss (MISS). This header must be set by your caching proxy for the test assertions to work.

Annotations

Annotate your controller actions to invalidate routes and paths when those actions are executed.

Note: Annotations need the SensioFrameworkExtraBundle including registering the Doctrine AnnotationsRegistry. Some features also need the ExpressionLanguage. Make sure to *install the dependencies first*.

@InvalidatePath

Invalidate a path:

```
use FOS\HttpCacheBundle\Configuration\InvalidatePath;

/**
 * @InvalidatePath("/articles")
 * @InvalidatePath("/articles/latest")
 */
public function editAction()
{
}
```

See *Invalidation* for more information.

@InvalidateRoute

Invalidate a route with parameters:

```
use FOS\HttpCacheBundle\Configuration\InvalidateRoute;

/**
 * @InvalidateRoute("articles")
 * @InvalidateRoute("articles", params={"type" = "latest"})
 */
public function editAction()
{
}
```

You can also use *expressions* in the route parameter values. This obviously *requires the ExpressionLanguage component*. To invalidate route articles with the number parameter set to 123, do:

```
/**
 * @InvalidateRoute("articles", params={"number" = {"expression"="id"}})
 */
public function editAction(Request $request, $id)
{
    // Assume $request->attributes->get('id') returns 123
}
```

The expression has access to all request attributes and the request itself under the name `request`.

See *Invalidation* for more information.

@Tag

You can make this bundle tag your response automatically using the @Tag annotation. *Safe* operations like GET that produce a successful response will lead to that response being tagged; modifying operations like POST, PUT, or DELETE will lead to the tags being invalidated.

When `indexAction()` returns a successful response for a safe (GET or HEAD) request, the response will get the tag `news`. The tag is set in a custom HTTP header (`X-Cache-Tags`, by default).

Any non-safe request to the `editAction` that returns a successful response will trigger invalidation of both the `news` and the `news-123` tags.

Set/invalidate a tag:

```
/**
 * @Tag("news-article")
 */
public function showAction()
{
    // ...
}
```

GET `/news/show` will

Multiple tags are possible:

```
/**
 * @Tag("news")
 * @Tag("news-list")
 */
public function indexAction()
{
    // ...
}
```

If you prefer, you can combine tags in one annotation:

```
/**
 * @Tag({"news", "news-list"})
 */
```

You can also use *expressions* in tags. This obviously *requires the ExpressionLanguage component*. The following example sets the tag `news-123` on the Response:

```
/**
 * @Tag(expression="'news-~id'")
 */
public function showAction($id)
{
    // Assume request parameter $id equals 123
}
```

Or, using a param converter:

```
/**
 * @Tag(expression="'news-~article.getId() ")
 */
public function showAction(Article $article)
{
```

```
// Assume $article->getId() returns 123
}
```

See *Tagging* for an introduction to tagging. If you wish to change the HTTP header used for storing tags, see *tags*.

The Cache Manager

Use the CacheManager to explicitly invalidate or refresh paths, URLs, routes, tags or responses with specific headers.

By *invalidating* a piece of content, you tell your caching proxy to no longer serve it to clients. When next requested, the proxy will fetch a fresh copy from the backend application and serve that instead.

By *refreshing* a piece of content, a fresh copy will be fetched right away.

Note: These terms are explained in more detail in [An Introduction to Cache Invalidation](#).

`invalidatePath()`

Important: Make sure to [configure your proxy](#) for purging first.

Invalidate a path:

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

Note: The `flush()` method is explained *below*.

Invalidate a URL:

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

Invalidate a route:

```
$cacheManager = $container->get('fos_http_cache.cache_manager');
$cacheManager->invalidateRoute('user_details', array('id' => 123));
```

Invalidate a regular expression:

```
$cacheManager = $container->get('fos_http_cache.cache_manager');
$cacheManager->invalidateRegex('.*', 'image/png', array('example.com'));
```

The cache manager offers a fluent interface:

```
$cacheManager
->invalidateRoute('villains_index')
->invalidatePath('/bad/guys')
->invalidateRoute('villain_details', array('name' => 'Jaws'))
->invalidateRoute('villain_details', array('name' => 'Goldfinger'))
->invalidateRoute('villain_details', array('name' => 'Dr. No'))
;
```

refreshPath() and refreshRoute()

Note: Make sure to [configure your proxy](#) for purging first.

Refresh a path:

```
$cacheManager = $container->get('fos_http_cache.cache_manager');
$cacheManager->refreshPath('/users');
```

Refresh a URL:

```
$cacheManager = $container->get('fos_http_cache.cache_manager');
$cacheManager->refreshPath('http://www.example.com/users');
```

Refresh a Route:

```
$cacheManager = $container->get('fos_http_cache.cache_manager');
$cacheManager->refreshRoute('user_details', array('id' => 123));
```

invalidateTags()

Invalidate cache tags:

```
$cacheManager->invalidateTags(array('some-tag', 'other-tag'));
```

Note: Marking a response with tags can be done through the *ResponseTagger*.

flush()

Internally, the invalidation requests are queued and only sent out to your HTTP proxy when the manager is flushed. The manager is flushed automatically at the right moment:

- when handling a HTTP request, after the response has been sent to the client (Symfony's `kernel.terminate` event)
- when running a console command, after the command has finished (Symfony's `console.terminate` event).

You can also flush the cache manager manually:

```
$cacheManager->flush();
```

Glossary

Cacheable According to [RFC 7231](#), a *response* is considered cacheable when its status code is one of 200, 203, 204, 206, 300, 301, 404, 405, 410, 414 or 501.

In FOSHttpCacheBundle, this can be changed by configuring the *cacheable.response* section.

Safe A *request* is safe if its HTTP method is GET or HEAD. Safe methods only retrieve data and do not change the application state, and therefore can be served with a response from the cache.

Testing

Testing the Bundle

To run this bundle's tests, clone the repository, install vendors, and invoke PHPUnit:

```
$ git clone https://github.com/FriendsOfSymfony/FOSHttpCacheBundle.git
$ cd FOSHttpCacheBundle
$ composer install
$ vendor/bin/phpunit
```

Tip: See the [FOSHttpCache library's docs](#) on how to write integration tests that validate your caching code and configuration against a caching proxy.

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. If your idea is specific to the Symfony framework, it belongs into the `FOSHttpCacheBundle`, otherwise it should go into the `FOSHttpCache` library.

When you change code, you can run the tests as described in [Testing](#).

Building the Documentation

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

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

To build the docs:

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


C

Cacheable, [43](#)

R

RFC

RFC 2616, [24](#)

RFC 2616#section-14.9, [24](#)

RFC 5861#section-3, [24](#)

RFC 5861#section-4, [24](#)

RFC 7232#section-2.2.1, [26](#)

RFC 7232#section-2.4, [26](#)

RFC 7234#section-5.2, [24](#)

RFC 7234#section-5.2.2.1, [24](#)

RFC 7234#section-5.2.2.4, [24](#)

RFC 7234#section-5.2.2.7, [24](#)

S

Safe, [43](#)