# HTTP response headers: harnessing the power of Vary

**Rogier “Doc” Mulhuijzen**
Senior Professional Services Engineer, Fastly

# Best practices for HTTP response headers: Vary

**Rogier "Doc" Mulhuijzen, Senior Professional Services Engineer, Fastly**
Doc is a senior professional services engineer and Varnish wizard at Fastly, where performance tuning and troubleshooting have formed the foundation of his 20+ year career. When he's not helping customers, he sits on the Varnish Governance Board, where he helps give direction and solve issues for the Varnish open source project. Formerly an Oracle DBA and Unix admin, Rogier learned scaling while working for Ziggo, a major Dutch ISP. In his spare time, he likes to conquer all terrains by riding motorcycles, snowboarding, and sailing. Find him on Twitter as @drwilco.

Vary is one of the most powerful HTTP response headers. Used correctly, it can do wonderful things. Unfortunately, this header is frequently used incorrectly, which can lead to abysmal cache hit ratios. Worse still, if it's not used when it should be, the wrong content could be delivered.

Instead of just pointing you to the Vary specification, I'm going to explain the Vary header using the most common use case: compression. I'm also going to use Fastly's content delivery network (CDN) as a running example, but these best practices extend beyond the Fastly use case. Our CDN is powered by Varnish Configuration Language (VCL) — you'll see references to Varnish, the open source HTTP accelerator, throughout this article. If you're unfamiliar, I recommend this intro to Varnish for further reading.

If you're using Apache's mod_deflate, the correct Vary header is automatically added to your responses. The same goes for Fastly's gzip feature. But as this is a very simple real-world use of Vary, I will use it to demonstrate how Vary works.

## Anatomy of an HTTP Request

Normally, when a request comes into one of Fastly's caches, only two parts of the request are used to find an object in the cache: the path (and query string, if present), and the Host header.

This is a pretty average request for http://example.com/somepage.php:

```
GET /somepage.php HTTP/1.1
Host: example.com
Connection: keep-alive
Accept: text/html,application/xhtml+xml,application/xml;q=0.9,image/webp,*/*;q=0.8
User-Agent: Mozilla/5.0 (Windows NT 6.3; WOW64) AppleWebKit/537.36 ...
Accept-Language: en-US,en;q=0.8
Accept-Encoding: gzip,deflate,sdch
```

As shown above, the browser sends a lot of information along with the URL. The Accept header tells you what sort of content the browser prefers, User-Agent specifies which version of what browser it is, Accept-Language contains a list of languages (and dialects) that the user has configured, and `Accept-Encoding` shows which compression schemes the browser supports.

For practical purposes, we only care about `gzip`. `deflate` is ancient, and `sdch` is not used by anyone except Google.

## Compression at work

Hypothetical situation: You have a webserver without `mod_deflate`, but you've figured out how to do `gzip` compression in PHP. So when you see `gzip` in the `Accept-Encoding` header, you set the `Content-Encoding: gzip` header to tell the browser what you're doing, and you compress the response body.

Now imagine that this server is the origin for a Fastly service, and this page is one that we can cache. What would happen if a browser that doesn't understand compression is the first to ask for this page? We'd end up with the uncompressed page in our cache.

Is this a problem? Just a small one. If a browser that does understand compression requests this page, it will get the uncompressed version from our cache, and will render just fine. But the uncompressed version is bigger, so it will cost more bandwidth to transmit, resulting in slower delivery for the end user and higher costs for you.

The bigger problem is when the first request for an object comes from a browser that does do compression, and we end up with the compressed version in our cache. Now, when a browser comes along that doesn't understand compression, it gets the compressed version and has no idea what to do other than display gibberish.

## Vary to the rescue

There are two ways that you can fix this. First, you can change your cache key in your Fastly configuration, but that will cause additional problems:

1. You would have to purge both the compressed and uncompressed versions separately.
2. A mistake made here could cause all URLs to return the same single object.

Instead, you can use the Vary header and avoid both of these problems.

The Vary header tells any HTTP cache which parts of the request header, other than the path and the Host header, to take into account when trying to find the right object. It does this by listing the names of the relevant headers, which in this case is `Accept-Encoding`. If there are multiple headers that influence the response, they would all be listed in a single header, separated by a comma.

The response headers for a compressed response should look something like:

```
HTTP/1.1 200 OK
Content-Length: 3458
Cache-Control: max-age=86400
Content-Encoding: gzip
Vary: Accept-Encoding
```

And for an uncompressed response, the response headers look like:

```
HTTP/1.1 200 OK
Content-Length: 8365
Cache-Control: max-age=86400
Vary: Accept-Encoding
```

Note that the Vary header is present in the response regardless of whether or not compression is used.

Why is this? Let's look at what happens when it is actually there.

First, a request comes in for an object, without an `Accept-Encoding` header. The object is not in the cache, so we request it from the origin, which returns it with the Vary header. When Fastly stores the object in the cache, the Vary header is noted, and the values of the relevant headers from the request are stored as well.

So now there's an object in the cache that has a little flag on it that says "only to be used for requests that have no `Accept-Encoding` in the request."

Now, imagine a browser comes along that does understand compression and sends the request as outlined above. First, we look it up by the Host header and path. This will find the object, but the request has an `Accept-Encoding` header, set to `gzip,deflate,sdch`, and that doesn't match the flag put on this object.

So Fastly goes to your origin again, and this time we should get back a compressed version of the object. This response is then stored with a flag saying that this version should only be used for requests with `Accept-Encoding: gzip,deflate,sdch.`

If the Vary header hadn't been there in the first response, we wouldn't have known that we couldn't use the cached object for the second request.

## Normalization

You might wonder if all browsers these days send the same `Accept-Encoding` header.

Unfortunately, the answer is no.

I sampled 100,000 requests on one of our caches, and got 44 different `Accept-Encoding` headers. (For those interested in how I did that, or the numbers, see here.)

If all of those requests had been for the same URL, we would have had 44 "different" versions in our cache. But because the origin can only generate two versions, one if `gzip` is present in `Accept-Encoding` and one if it's not, that's 42 requests to the origin we'd like to avoid.

Since the origin only cares whether `gzip` is there or not, why don't we normalize the `Accept-Encoding` header to either contain `gzip` or not be present at all?

There are only ever two variations of the `Accept-Encoding` header in our requests, and therefore we will only ever have two variations of the object in our cache.

This is easily done with a little VCL:

```
# do this only once per request
 if (req.restarts == 0) {
   # normalize Accept-Encoding to reduce vary
   if (req.http.Accept-Encoding ~ "gzip") {
     set req.http.Accept-Encoding = "gzip";
   } else {
     unset req.http.Accept-Encoding;
   }
 }
```

You still might want to support some ancient HTTP clients, so let's add support for `deflate` as well, and let's make sure Internet Explorer 6 doesn't need to deal with compression (it's notoriously bad at it).

```
# do this only once per request
if (req.restarts == 0) {
  # normalize Accept-Encoding to reduce vary
  if (req.http.Accept-Encoding) {
    if (req.http.User-Agent ~ "MSIE 6") {
     unset req.http.Accept-Encoding;
    } elsif (req.http.Accept-Encoding ~ "gzip") {
      set req.http.Accept-Encoding = "gzip";
    } elsif (req.http.Accept-Encoding ~ "deflate") {
      set req.http.Accept-Encoding = "deflate";
    } else {
      unset req.http.Accept-Encoding;
    }
  }
}
```

## Bad to the bone

As you can see, just adding a simple Vary header to your response without doing some normalization of the request headers could have had a pretty disastrous impact on the amount of requests sent to the origin. The only thing preventing it is the standard normalization that Fastly does.

First of all, you don't want to Vary on a header that has a lot of variations without normalization.

Second, when normalizing, try to bring the header down to a handful of possibilities at most. One way to do this is to have the values hardcoded in your VCL instead of relying on `regsub()`. A good rule of thumb: for popular content with long expiry times, the amount of traffic to your origin scales in a linear fashion with the amount of possible values.

Here are some Vary headers I've seen over the years, why they're bad, and how to normalize them.

## Vary: User-Agent

There are literally thousands of different User-Agent strings. In a sample of 100,000 requests, I found just shy of 8,000 different ones.

In one scenario, you want to present different formatting to mobile users. Note that with this example, the User-Agent header is replaced with a simple string that looks nothing like the regular values for this header. If you use this, make sure your origin knows how to deal with it.

You could use VCL like this:

```
if (req.http.User-Agent ~ "(Mobile|Android|iPhone|iPad)") {
  set req.http.User-Agent = "mobile";
} else {
  set req.http.User-Agent = "desktop";
}
```

## Vary: Referer

If your content is very popular, it will get linked to by many other sites, and every search query on Google has a unique URL. Both of these things will lead to a large number of unique Referer values.

Say you want to show some sort of welcome pop-up to people who arrived on one of your pages from a page that is not part of your site, but you don't want this to happen for people navigating within your site.

The VCL needed would look like:

```
if (req.http.Referer ~ "^https?://www.example.com/") {
  set req.http.Referer = "http://www.example.com/";
} else {
  unset req.http.Referer;
}
```

## Vary: Cookie

`Cookie` is probably one of the most unique request headers, and is therefore very bad. Cookies often carry authentication details, in which case you're better off not trying to cache pages, but just passing them through. If you're interested in caching with tracking cookies, read more here.

However, sometimes cookies are used for A/B testing purposes, in which case it's a good idea to Vary on a custom header and leave the Cookie header intact. This avoids a lot of additional logic to make sure the Cookie header is left for URLs that need it (and are probably not cacheable).

```
  sub vcl_recv {
   # set the custom header
   if (req.http.Cookie ~ "ABtesting=B") {
     set req.http.X-ABtesting = "B";
   } else {
     set req.http.X-ABtesting = "A";
   }
...
}

...


sub vcl_fetch {
  # vary on the custom header
  if (beresp.http.Vary) {
    set beresp.http.Vary = beresp.http.Vary ", X-ABtesting";
  } else {
    set beresp.http.Vary = "X-ABtesting";
  }
  ...
}

...
sub vcl_deliver {
  # Hide the existence of the header from downstream
  if (resp.http.Vary) {
    set resp.http.Vary = regsub(resp.http.Vary, "X-ABtesting", "Cookie");
  }
  # Set the ABtesting cookie if not present in the request
  if (req.http.Cookie !~ "ABtesting=") {
    # 75% A, 25% B
    if (randombool(3, 4)) {
      add resp.http.Set-Cookie = "ABtesting=A; expires=" now + 180d "; path=/;";
    } else {
      add resp.http.Set-Cookie = "ABtesting=B; expires=" now + 180d "; path=/;";
    }
  }

  ...
}
```

In `vcl_recv`, which is before the lookup in the cache, you use VCL to add a custom header, based on the Cookie. This assumes that the cookie value is either `B` or `A`, and if the cookie is missing, it defaults to `A`.

Then in `vcl_fetch` you add your custom header to the Vary. And finally in `vcl_deliver`, you replace your custom header in the Vary with `Cookie`. Not only will this hide the existence of the custom header from the outside world, but if there happens to be a shared cache between Fastly and the end users, they will still get the correct variation. And now you have A/B testing, where all the browsers see is just another cookie, and your origin only needs to do something different based on a very simple header. `(X-ABtesting)`

## Vary: *

Don’t use this, period.

The HTTP RFC says that if a Vary header contains the special header name `*`, each request for said URL is supposed to be treated as a unique (and uncacheable) request.

This is much better indicated by using **`Cache-Control: private`**, which is clearer to anyone reading the response headers. It also signifies to any intermediary cache that the object shouldn’t ever be stored, which is much more secure.

## Vary: Accept-Encoding, User-Agent, Cookie, Referer

I’m not kidding. I have seen this. Without any normalization. As you might have guessed, there’s maybe one in a googolplex chance that this will ever see a cache hit.

## The next level

So far I’ve discussed Vary for compression, introduced some (simple) logic for device detection, and set up A/B testing logic at the edge. If you’d like to learn more about web performance, take a look at the Fastly blog.