

Enterprise Programming 2

Lesson 05: 3xx Redirection, Conditional Requests and Caching

Bogdan Marculescu

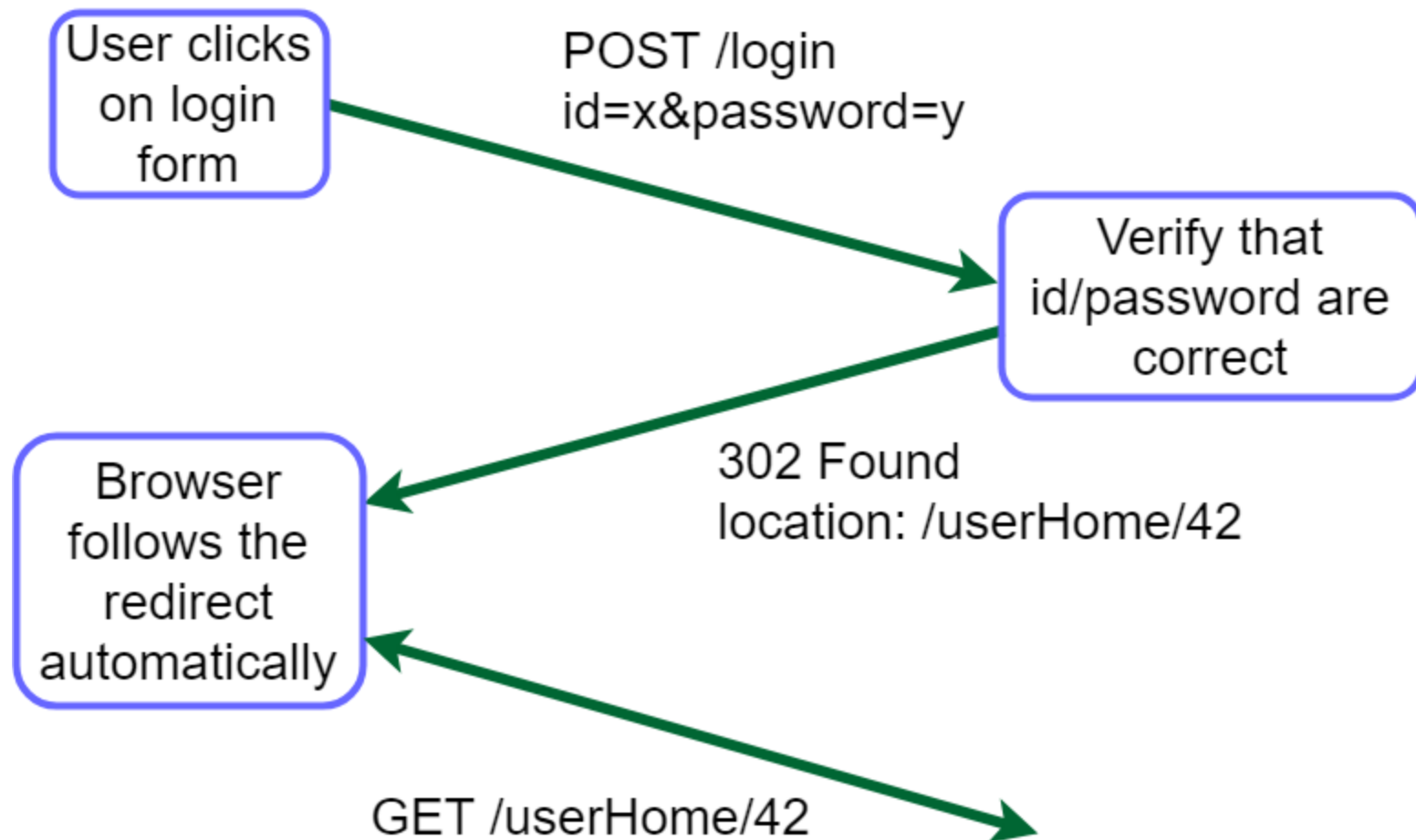
Goals

- Understand 3xx HTTP redirections
- Understand how to make *conditional requests* in HTTP to improve performance
- Understand how HTTP deals with *caches*
 - both *public* and *private* caches

3xx Redirection

3xx Status Code

- They represent *redirection*
- You ask for a resource at URI X, but then the server tells you should rather go to URI Y
 - “where” to go will be specified in the *Location* header
- ... or operation on X is completed, and result is visible at Y
- Example in a browser: how to tell the client to automatically go to homepage after a successful login on the login page?
 - assuming HTML forms, and no JavaScript



Messy Standard

- The HTTP standard is a mess when it comes to 3xx status codes
 - ie, lot of ambiguities and undefined behavior
 - eg, see <http://insanecoding.blogspot.no/2014/02/http-308-incompetence-expected.html>
- You should use redirection when needed, but keep it minds that different clients might have different, strange behaviors
- The main issue is on how HTTP methods could be changed
 - eg, in previous example, a POST was redirected into a GET

Permanent Redirection

- You ask for X, but server tells you that now it is *permanently* moved to Y
- A client, if it follows redirects automatically, will do a new request to Y
- From now on, every time you ask for X, the client would rather call for Y directly, and never use X again
 - as the redirection is *permanent*, there is no point in asking for X, you can just go directly for Y

Temporary Redirection

- You ask for X, but server tells you that now it is *temporarily* moved to Y
- A client, if it follows redirects automatically, will do a new request to Y
- Every time you ask for X, the client will still ask for X, and ignore the previously obtained Ys
 - as the redirection is *temporary*, each time you ask for X you could get a different Y'

3xx Codes for RESTful APIs

- 301: Permanent redirection, but use it *only* for GET
 - unless you like random surprises, like clients transforming a PUT into a GET
- 302: change from POST to GET
 - important for HTML forms, but arguably no need in a REST API
- 304: For cache control
 - eg no need to retrieve resource, as the one in cache is still valid
- 307: Temporary redirection
- 308: Permanent redirection, for methods other than GET
 - note: many client libraries will not follow such redirect automatically, so do not rely too much on it

Conditional Requests

GET The Same Data

- It is not uncommon to do GET on the same endpoints, several times
- Eg, think of home pages of sites you visit often
 - *google.com, facebook.com*, etc.
- You still need to make a GET, but then could save on *bandwidth* if server says previous response is still *valid*
 - and so not provide payload in HTTP response body
- However, need to save previous response somewhere, eg a *cache*

Response Validity

- 2 ways to specify validity, using HTTP Headers
- *Last-modified*: tells when the resource was last modified
 - the clock is based on the server, NOT the client
- *ETag*: a unique string identifier representing the current status of the resource
 - if the state of the resource changes, then the ETag should change as well
 - could be computed as a hash of the response
 - if based on a JPA entity, could use its *@Version* (if any)

Last-Modified

- Usually easy to compute
- But need to be stored somewhere on the server
 - eg, an extra column in the database tables
- Issue: HTTP Date resolution is based on *seconds*
 - if several updates in the same second, might lose the most recent ones

ETag

- As being a unique identifier, it is more precise than Last-Modified
- But not always easy to define what to use as unique identifier for a resource
- A *hash* can be used (eg MD5), but there is always the risk of a potential collision
 - albeit probability should be very low

Conditional GET Requests

- HTTP Headers *If-None-Match* and *If-Modified-Since*
- ***If-None-Match***: send the previously obtained ETag. Should get new payload only if ETag on server has changed
- ***If-Modified-Since***: send the previously obtained Last-Modified timestamp. Should get new payload only if new update on server has happened
- If server sends no payload because resource has not changed, then status code is **304**

Conditional Changes

- You might want to do a POST/PUT/PATCH only if the state on server has not changed
- A GET followed by a PUT would be two different requests, NOT done *atomically*
 - someone else might have modified the state between the GET and PUT
- If the PUT is based on data read by GET, and you want to abort the PUT if someone else changed the state, you can have a conditional request

Cont. Conditional POST/PUT/PATCH

- ***If-Match***: do the change operation only if the ETag does match. Will use/send ETag from a previous GET
- ***If-Unmodified-Since***: do the change operation only if the timestamp was not changed. Will use/send *Last-modified* value from a previous GET
- If on the server those checks fail, the server will send a **412** *Precondition Failed*, and the operation is NOT executed

HTTP Caches

Caching

- With conditional GET requests, we might avoid re-downloading a resource if not changed on server
- Still need to save such resource locally, in a so called *cache*
- Can see a cache like a glorified *Map* data-structure
 - eg, key being the ETag, and value being the downloaded resource
- But, even if having a cache, still need to pay a round-trip of HTTP request
 - ie, even if getting a **304** with no body payload, still have to do a GET request

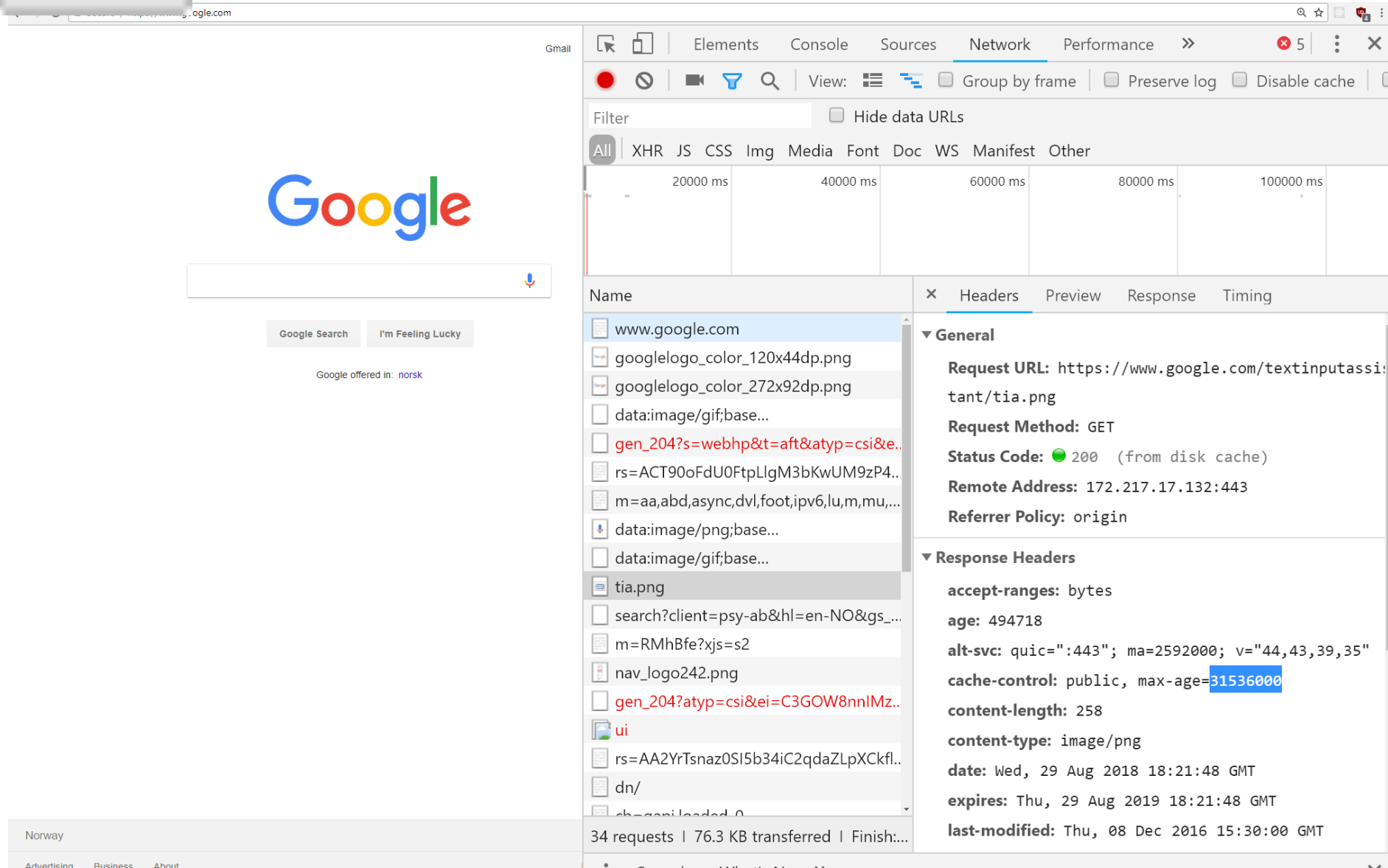
Freshness

- What about using cache WITHOUT doing a conditional GET on server?
- How can we know that the resource is still *fresh* and was not changed on server?
- Server can explicitly tell us for how long a resource is fresh, using the ***Cache-Control*** HTTP header
- ***Max-age***: number of seconds that the client is safe to reuse a downloaded resource without a new conditional GET
 - eg, ***Cache-Control: max-age=30***

How To Set Max-age?

- It depends on the context...
- Eg. forecast application: maybe computing forecast every hour, so *Max-age* till the next update
- Eg. static files like HTML/JS/CSS/IMG/etc.: if you deploy new version of your app no more than once a day, then can have something like *max-age=86400*
 - there are 86400 seconds in 1 day
- etc.

Ex.:  *tia.png* cached for 1 year



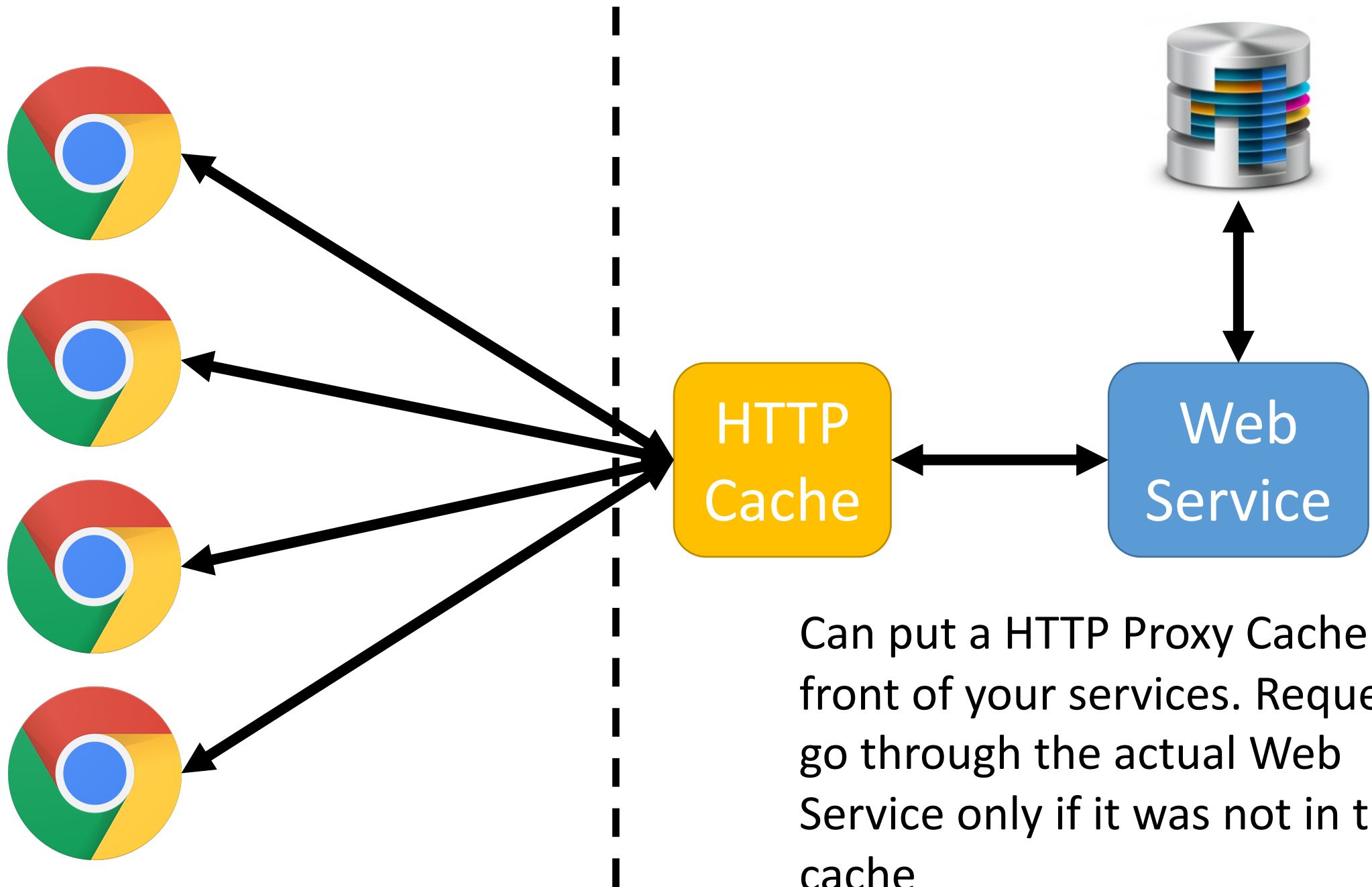
The screenshot displays the Google homepage on the left and the Chrome DevTools Network tab on the right. The Network tab shows a list of requests, with 'tia.png' selected. The details for 'tia.png' are expanded, showing the following information:

- Name:** tia.png
- General:**
 - Request URL:** https://www.google.com/textinputassitant/tia.png
 - Request Method:** GET
 - Status Code:** 200 (from disk cache)
 - Remote Address:** 172.217.17.132:443
 - Referrer Policy:** origin
- Response Headers:**
 - accept-ranges:** bytes
 - age:** 494718
 - alt-svc:** quic=":443"; ma=2592000; v="44,43,39,35"
 - cache-control:** public, max-age=31536000
 - content-length:** 258
 - content-type:** image/png
 - date:** Wed, 29 Aug 2018 18:21:48 GMT
 - expires:** Thu, 29 Aug 2019 18:21:48 GMT
 - last-modified:** Thu, 08 Dec 2016 15:30:00 GMT

Cache Invalidation

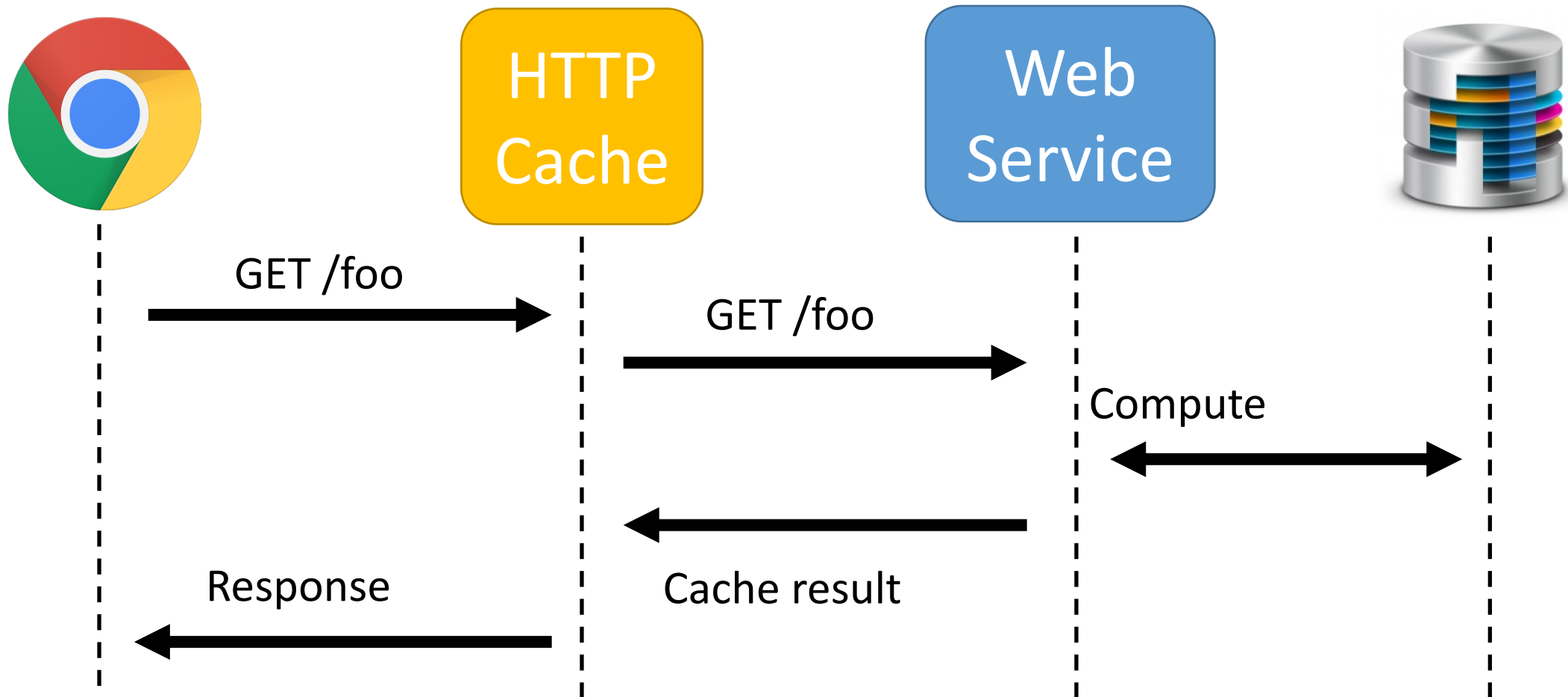
- Assume you give a *Max-age* of 1 week
- Resources will be cached on each client for 1 week
- What if, within that week, you want to make an update?
 - eg, you find out there was a serious bug which led to create invalid resources
- *In HTTP, there is no way to tell client to invalidate its cache* 😞
- If the invalidation for resource X is really critical, only thing you can do is to put X in a different URL, and have all links pointing to the new URL
 - So, old cached URL would not be used any longer
 - Note: this might become quite expensive, as need to update all existing links in your whole app. So, not something to do often...

Public HTTP Caches



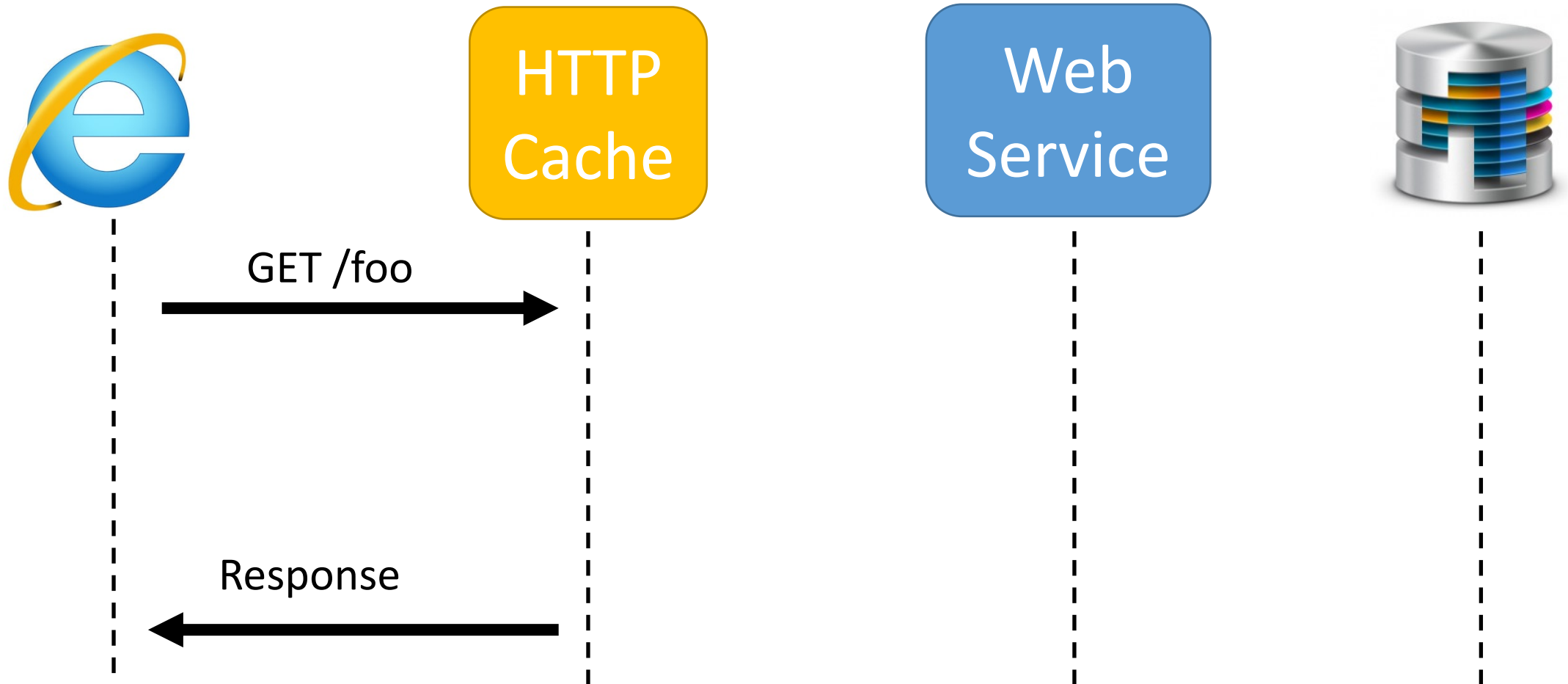
Why?

First client will trigger whole computation, but all followings will access directly from public cache on first request



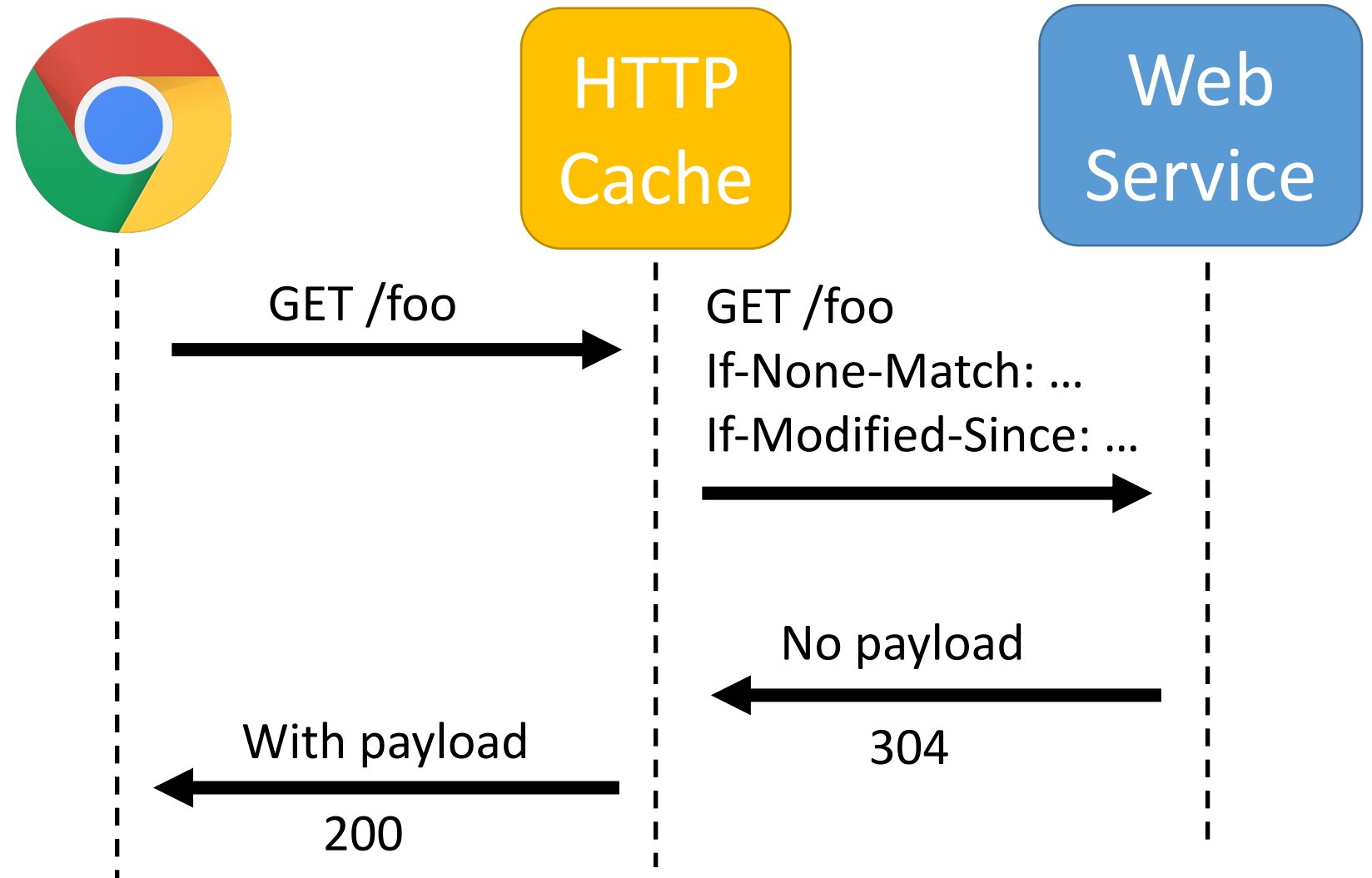
Second and Following Requests

No computation on Web Service, as results were cached in the Public Proxy HTTP Cache



Adding Cache Control Headers

Even if client does not use any cache control, the Public Cache can still add those when communicating with the web service



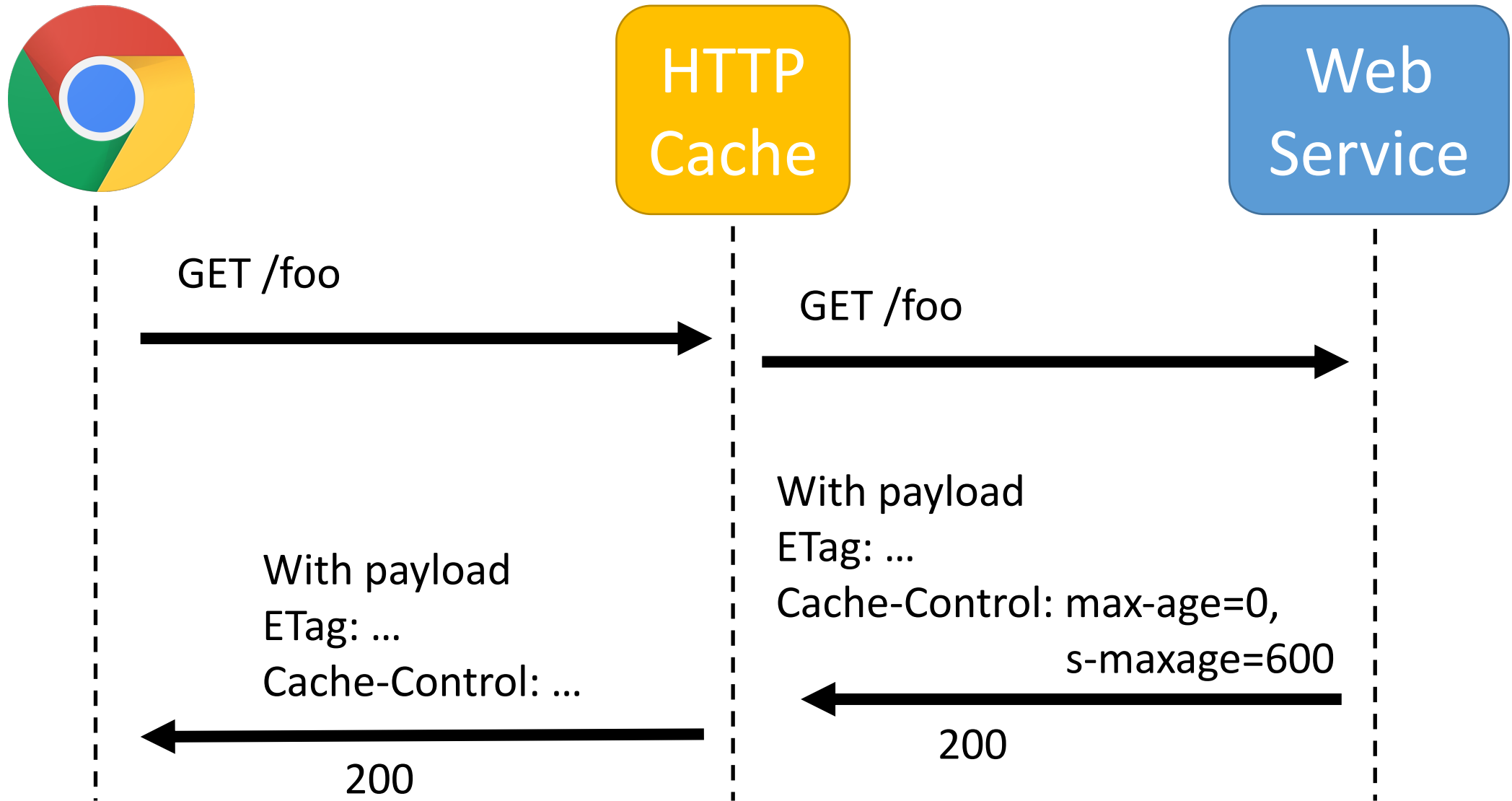
Controlling Public Caches

- ***s-maxage***: for how long a resource stored in public cache can be considered *fresh*
- Example: ***Cache-Control: max-age=60, s-maxage=300***
 - 1 minute for private cache (eg in browser) and 5 minutes for public cache
- Why having different values for public and private caches?
 - depends on the context...

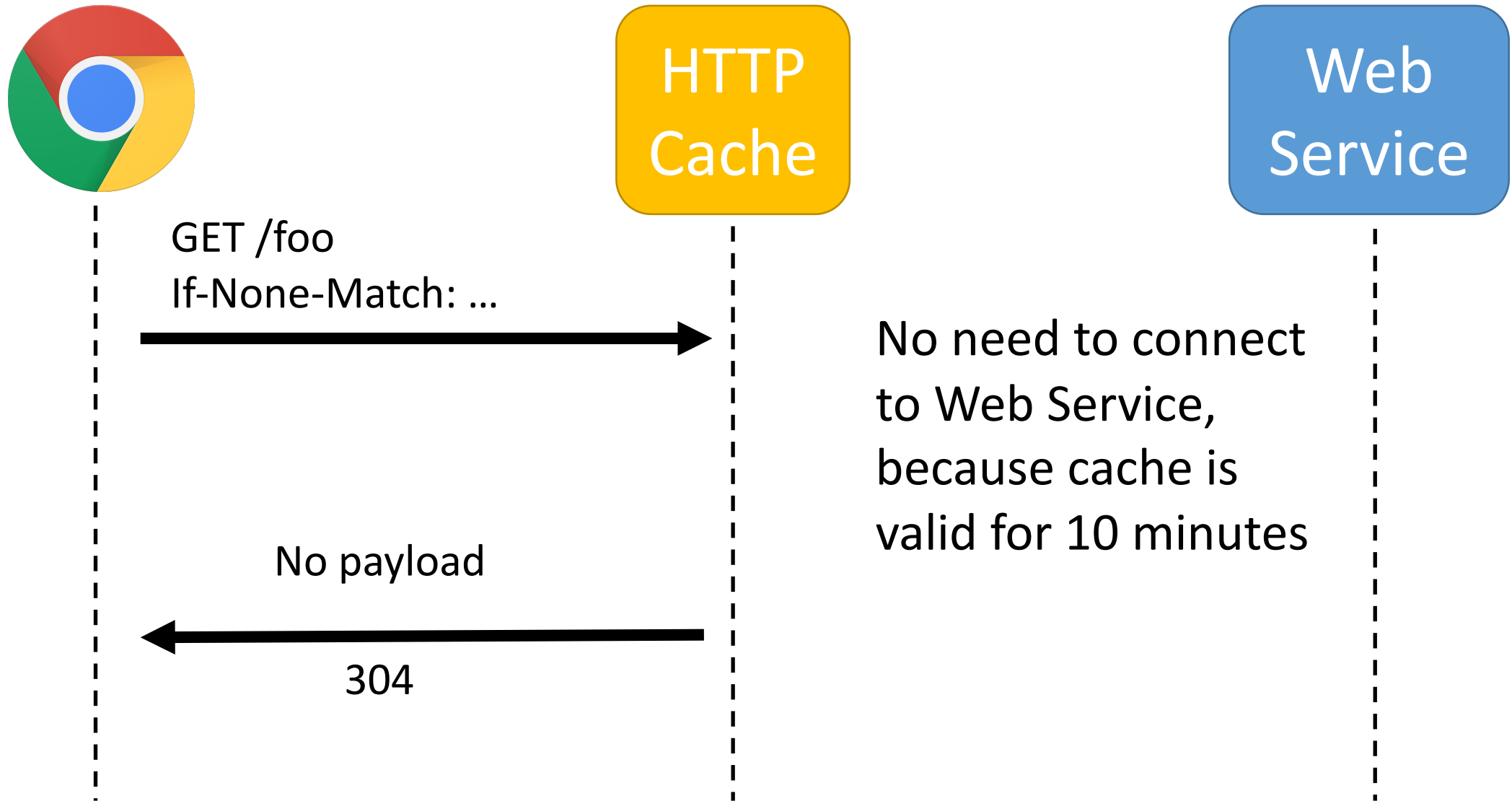
Ex. *Cache-Control: max-age=0, s-maxage=600*

- This means a response will never be considered *fresh* in the private cache, but will be fresh for 10 minutes on public cache
- Each client will always have to do a conditional GET request
- ***max-age=0*** does NOT mean that it cannot be cached, just that we need to validate each time with a conditional GET
 - we do not save the GET request, but could save on no-payload if **304**
- On public cache, we save for 10 minutes, so we avoid computing response and doing conditional GETs for 10 minutes but for first request

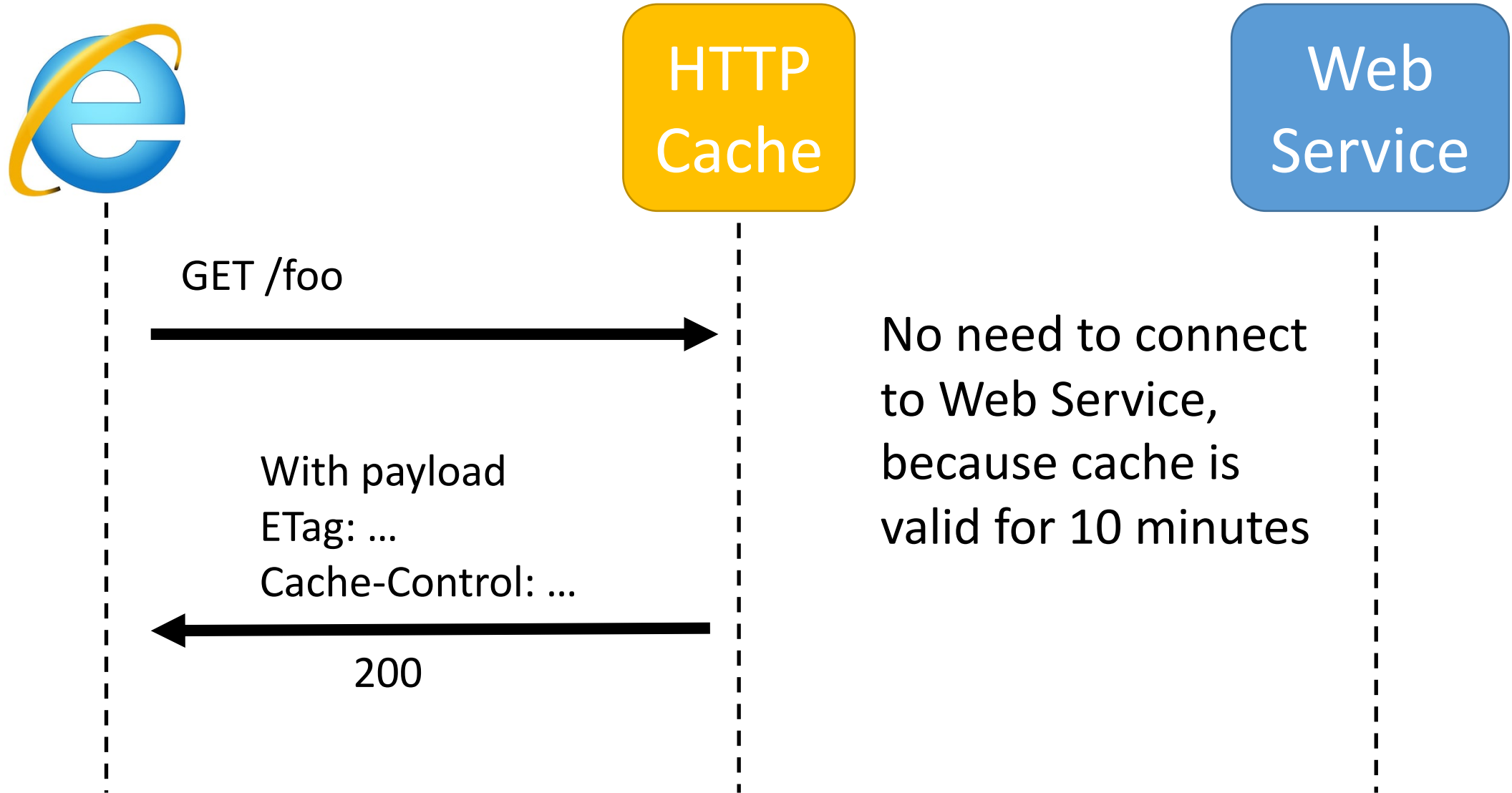
First Request



Second Request from Same Client



Third Request from Different Client



So???

- ***Cache-Control: max-age=0, s-maxage=600***
- *Good*: avoid re-computing responses
 - eg, business logic and access to database done only once for first request
- *Bad*: clients still need to do conditional GET requests, and cannot use directly the local cache without validating with server first
- *Good*: cache invalidation, we can manually reset Public Cache whenever we want, as we have full control on it

Other Cache Control Settings

- ***public***: response can be cached
- ***private***: can be cached only in a private cache, not public
- ***no-cache***: can be cached, but each time ask for validation
 - e.g., ***no-cache*** and ***max-age=0*** would be equivalent
- ***no-store***: never ever cache the response
 - note: some systems wrongly treat ***no-cache*** as it was a ***no-store***, and that is the reason why often you see ***max-age=0*** instead of ***no-cache***
- ***must-revalidate***: under certain conditions, caches “might” return stale, non-fresh values. Make sure to avoid those special cases
- ***proxy-revalidate***: same as above, but for public caches

Git Repository Modules

- *NOTE: most of the explanations will be directly in the code as comments, and not here in the slides*
- **advanced/rest/redirect**
- **advanced/rest/conditional-get**
- **advanced/rest/conditional-change**
- **advanced/rest/cache**
- Study relevant sections in RFC-7230, RFC-7231, RFC-7232 and RFC-7234