24th
Dec 2016

A Tale of Four Caches

by Yoav Weiss

A Tale of Four Caches

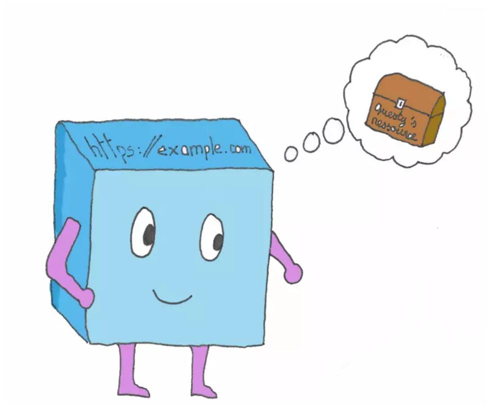
There's a lot of talk these days about browser caches in relation to preload, HTTP/2 push and Service workers, but also a lot of confusion.

So, I'd like to tell you a story about one request's journey to fulfill its destiny and find a matching resource.

> The following story is based on Chromium's terms and concepts, but other browsers are not inherently different.

Questy's Journey

Questy was a request. It was created inside the rendering engine (also called "renderer" to keep things shorter), with one burning desire: to find a resource that would make its existence complete and to live together happily ever after, at least until the current document is detached when the tab is closed.



Questy, dreaming of its resource

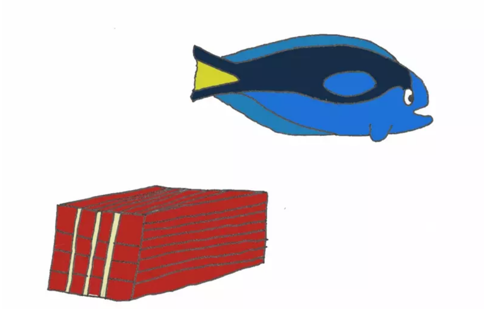
So Questy started its journey in its pursuit for happiness. But where would it find a resource that would be just the right one for it?

The closest place to look for one was at the…

Memory Cache

The Memory Cache had a large container full of resources. It contained all the resources that the renderer fetched as part of the current document and kept during the document's lifetime. That means that if the resource Questy is looking for was already fetched elsewhere in the current document, that resource will be found in the Memory Cache.

But a name like "the short term memory cache" might have been more appropriate: the memory cache keeps resources around only until the end of their navigation, and in some cases, even less then that.



The short term memory cache and its container

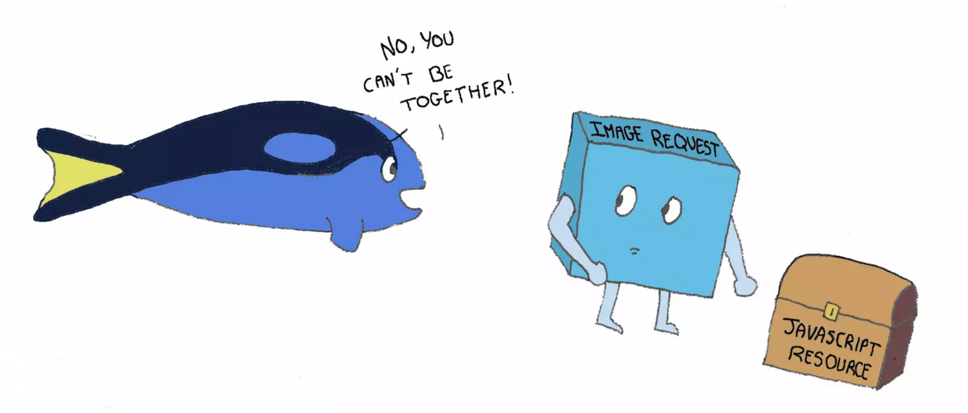
There are many potential reasons why the resource Questy is looking for was already fetched.

The preloader is probably the biggest one. If Questy was created as a result of a DOM node creation by the HTML parser, there's a good chance that the resource it needs was already fetched earlier on, during the HTML tokenization phase by the preloader.

Explicit preload directives (`<link rel=preload>`) is another big case where the preloaded resources are stored in the Memory Cache.

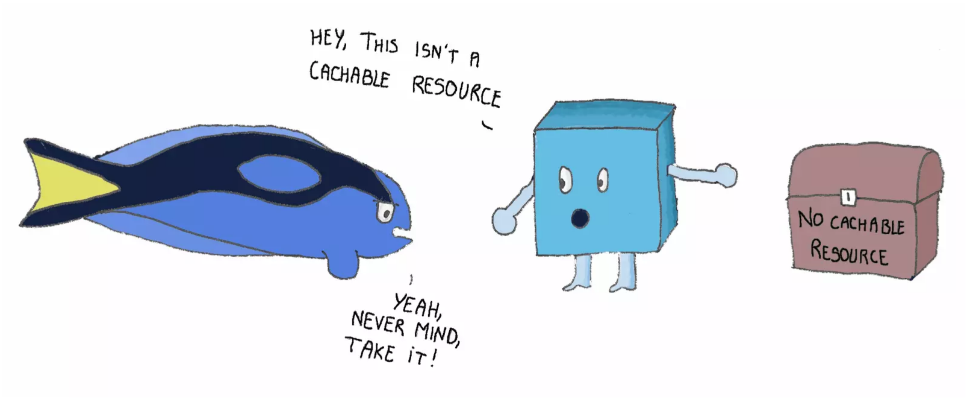
Otherwise, it's also possible that a previous DOM node or CSS rule triggered a fetch for the same resource. For example, a page can contain multiple `<img>` elements all with the same `src` attribute, which fetch only a single resource. The mechanism enabling those multiple elements to fetch only a single resource is the Memory Cache.

But, the Memory Cache would not give requests a matching resource that easily. Obviously, in order for a request and a resource to match, they must have matching URLs. But, that's not sufficient. They must also have a matching resource type (so a resource fetched as a script cannot match a request for an image), CORS mode and a few other characteristics.



The matching characteristics for requests from the Memory Cache are not well-defined in specifications, and therefore may slightly vary between browser implementations. Bleh.

One thing that Memory Cache doesn't care about is HTTP semantics. If the resource stored in it has `max-age=0` or `no-cache` `Cache-Control` headers, that's not something that Memory Cache cares about. Since it's allowing the reuse of the resource in the current navigation, HTTP semantics are not that important here.


HEY, THIS ISN'T A CACHABLE RESOURCE
YEAH, NEVER MIND, TAKE IT!
NO CACHABLE RESOURCE

The only exception to that is `no-store` directives which the memory cache does respect in certain situations (for example, when the resource is reused by a separate node).

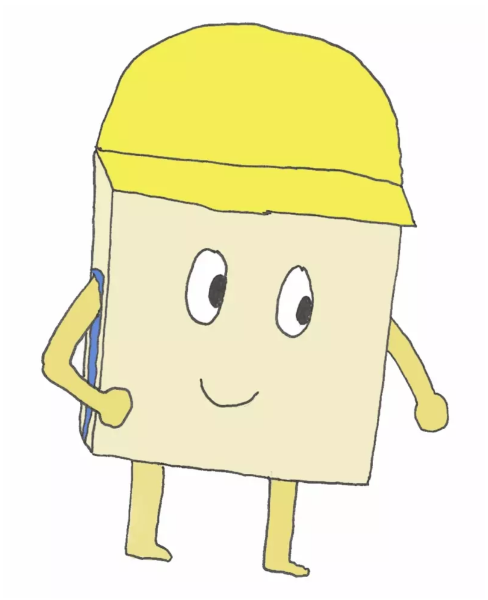
So, Questy went ahead and asked the Memory Cache for a matching resource. Alas, one was not to be found.

Questy did not give up. It got past the Resource Timing and DevTools network registration point, where it registered as a request looking for a resource (which meant it will now show up in DevTools as well as in resource timing, assuming it will find its resource eventually).

After that administrative part was done, it relentlessly continued towards the…

Service Worker Cache

Unlike the Memory Cache, the Service Worker doesn't follow any conventional rules. It is, in a way, unpredictable, only abiding to what their master, the Web developer, tells them.


A hard-working service worker

First of all, it only exists if a Service Worker was installed by the page. And since its logic is defined by the Web developer using JavaScript, rather than built into the browser, Questy had no idea if it would find a resource for it, and even if it would, would that resource be everything it dreamed of? Would it be a matching resource, stored in its cache? Or just a crafted response, created by the twisted logic of the Service Worker's master?

No one can tell. Since Service Workers are given their own logic, matching requests and potential resources, wrapped in a Response object, can be done any way they see fit.

Service Worker has a cache API, which enables it to keep resources around. One major difference between it and the Memory Cache is that it is persistent. Resources stored in that cache are kept around, even if the tab closes or the browser restarted. One case where they get evicted from the cache if the developer explicitly evicts them (using `cache.delete(resource)`). Another case happens if the browser runs out of storage space, and in that case, the *entire* Service Worker cache gets nuked, along with all other origin storage, such as indexedDB, localStorage, etc. That way, the Service Worker can know that the resources in that cache are in sync among themselves and with other origin storage.

The Service Worker is responsible for a certain scope, which at most, is limited to a single host. Service Workers can therefore only serve responses to requests requested from a document inside that scope.

Questy went up to the Service Worker and asked it if it has a resource for it. But the Service Worker had never seen that resource coming from that scope before and therefore had no corresponding resource to give Questy. So Service Worker sent Questy to carry on (using a `fetch()` call), and continue searching for a resource in the treacherous lands of the network stack.

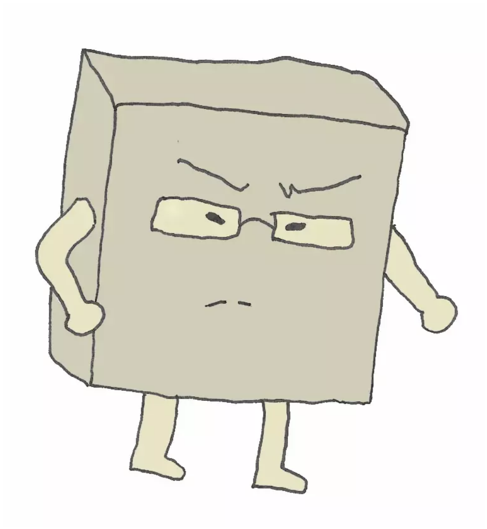
And once in the network stack, the best place to look for a resource was the…

HTTP Cache

The HTTP cache (also sometimes called "Disk cache" among its friends) is quite different from the caches Questy seen before it.

On the one hand, it is persistent, allowing resources to be reused between sessions and even across sites. If a resource was cached by one site, there's no problem for the HTTP cache to allow its reuse by other sites.

At the same time, the HTTP cache abides to HTTP semantics (the name kinda gives that part away). It will happily serve resources that it considers "fresh" (based on caching lifetime, indicated by their response's caching headers), revalidate resources that need revalidation, and refuse to store resources that it shouldn't store.



An overly strict HTTP cache

Since it's a persistent cache, it also needs to evict resources, but unlike the Service Worker cache, resources can be evicted one by one, whenever the cache feels like it needs the space to store more important or more popular resources.

The HTTP cache has a memory based component, where resource matching is being done for requests coming in. But if it actually finds a matching resource, it needs to fetch the resource contents from disk, which can be an expensive operation.

> We mentioned before that the HTTP Cache respects HTTP semantics. That's almost entirely true. There is one exception to that, when the HTTP cache stores resources for a limited amount of time. Browsers have the ability to prefetch resources for the next navigation. That can be done with explicit hints (`<link rel=prefetch>` or with the browser's internal heuristics. Those prefetched resources need to be kept around until next navigation, even if they are not cacheable. So when such a prefetched resource arrives at the HTTP cache, it is cached (and served without revalidation) for a period of 5 minutes.

The HTTP cache seemed rather strict, but Questy built up the courage to ask it if it has a matching resource for it. The response was negative :/

It will have to continue on towards the network. The journey over the network is scary and unpredictable, but Questy knew that it must find its resource no matter what. So it carried on. It found a corresponding HTTP/2 session, and was well on its way to be sent over the network, when suddenly it saw the…

Push "Cache"

The Push cache (better described as the “unclaimed push streams container”, but that’s less catchy as names go) is where HTTP/2 push resources are stored. They are stored as part of an HTTP/2 session, which has several implications.

The unclaimed push stream container AKA the push cache

The container is in no-way persistent. If the session is terminated, all the resources which weren’t claimed (i.e. were not matched with a request for them) are gone. If a resource is fetched using a different HTTP/2 session, it won’t get matched. On top of that, resources are kept around in the push cache container only for a limited amount of time. (~5 minutes in Chromium-based browsers)

The push cache matches a request to a resource according to its URL, as well as its various request headers, but it does not apply strict HTTP semantics.

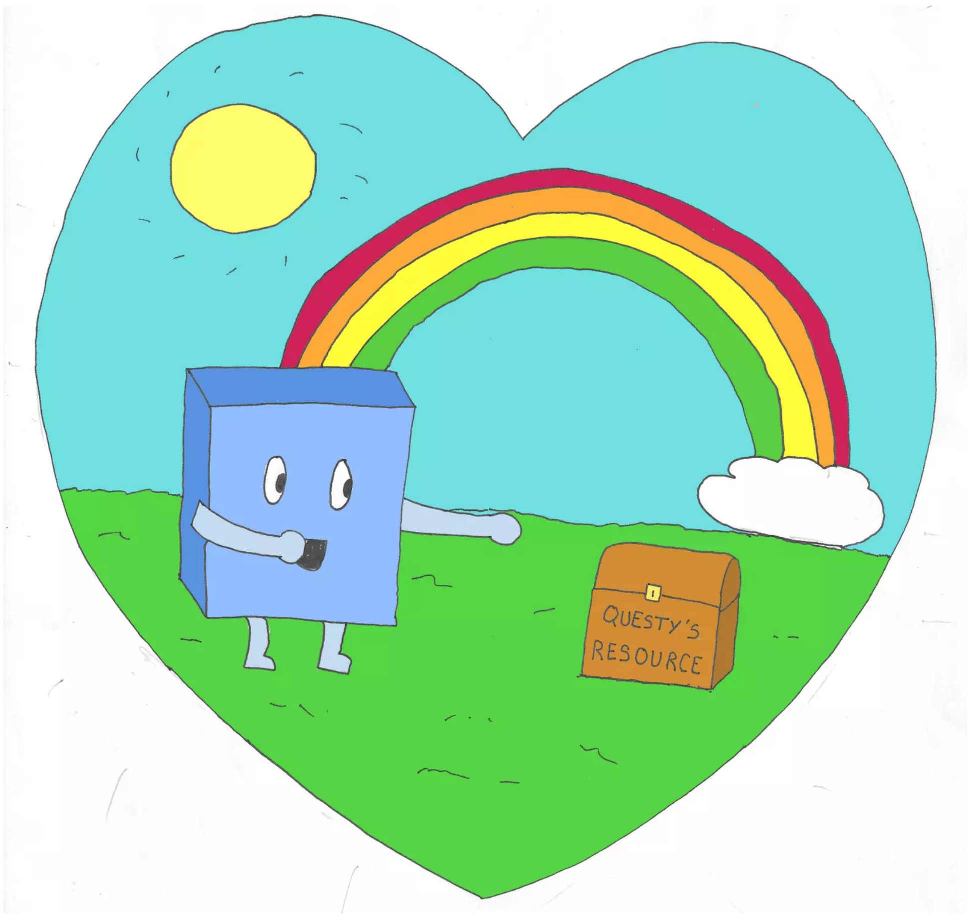
> The push cache is also not well-defined in specs and implementations may vary between browsers, operating systems and other HTTP/2 clients.

Questy had little faith, but still it asked the push cache if it has a matching resource. And to its surprise, it did!! Questy adopted the resource (which meant it removed the HTTP/2 stream from the unclaimed container) and was happy as a clam. Now it can start making its way back to the renderer with its resource.

On their way back, they went across the HTTP cache, which stopped them along the way to take a copy of the resource and store it in case future requests would need it.

Once they made it out of the net stack and back in Service Worker land, the Service Worker also stored a copy of the resource in its cache, before sending both back to the renderer.

And finally, once they got back to the renderer, Memory Cache kept a reference of the resource (rather than a copy), that it can use to assign the same resource to future requests in that same navigation session that may need it.



And they lived happily ever after, until the document got detached and both got to meet the Garbage Collector.

But that’s a story for another day.

Takeaways

So, what can we learn from Questy’s journey?

- Different requests can get matched by resources in different caches of the browser.
- The cache from which the request got matched can have an impact on the way this request is represented in DevTools and Resource Timing.
- Pushed resources are not stored persistently unless their stream got adopted by a request.
- Non-cacheable preloaded resources won’t be around for the next navigation. That’s one of the major differences between preload and prefetch.
- There are many underspecified areas here where observable behavior may differ between browser implementations. We need to fix that.

All in all, if you’re using preload, H2 push, Service Worker or other advanced techniques when trying to speed up your site, you may notice a few cases where the internal cache implementation is showing. Being aware of these internal caches and how they operate might help you to better understand what is going on and hopefully help to avoid unnecessary frustrations.

Thanks to Tim Kadlec and Jake Archibald for reviewing an early version of this article. And huge thanks to Yaara Weiss for this article’s illustrations and for being an awesome kid in general.

ABOUT THE AUTHOR

Yoav Weiss (@yoavweiss) has been working on mobile Web performance for longer than he cares to admit. He takes image bloat on the Web as a personal insult, which is why he joined the Responsive Image Community Group and implemented the various responsive images features in Blink and WebKit.

He is now working at Akamai as a Principal Architect, focused on making the Web platform faster by adding performance related features to browsers as well as to Akamai's CDN. You can follow his rants on Twitter or take a peek at his latest prototypes on Github.

When he's not writing code, he's probably slapping his bass, mowing the lawn in the French country-side or playing board games with his family.