



Web portal caching

# 1 Contents

|   |                                                                          |   |
|---|--------------------------------------------------------------------------|---|
| 2 | How HTTP caching works                                                   | 2 |
| 3 | Caching in WebKit                                                        | 4 |
| 4 | Client/Server implementation strategies                                  | 4 |
| 5 | Application cache . . . . .                                              | 5 |
| 6 | Custom HTTP application caching server running locally . . . . .         | 5 |
| 7 | Separately-maintained locally accessible copy of the portal contents . . | 6 |

8 The purpose of this document is to evaluate the available strategies to implement  
9 a custom, single-purpose browser restricted to a single portal website that hosts  
10 several HTML/JS applications.

11 The portal and the visited applications should be available even if no Internet  
12 connection is available.

13 If a connection to the Internet is available, the locally-stored contents should be  
14 refreshed.

15 Locally-stored copies should be used to speed up loading even when the connec-  
16 tion to the Internet is available.

17 The portal and the applications store all their runtime data using the `local-`  
18 `Storage`<sup>1</sup> or `IndexedDB`<sup>2</sup> mechanisms and how that is synchronized is out of the  
19 scope of this document, which instead focuses on how to manage static assets.

## 20 How HTTP caching works

21 Caching is a very important and complex feature in modern web engines to  
22 improve page load time and reduce bandwidth consumption. [RFC7234](#)<sup>3</sup> defines  
23 the mechanisms that control caching in the HTTP protocol regardless of its  
24 transport or serialization, which means that the same mechanisms apply to  
25 HTTPS and HTTP2 in the same way.

26 HTTP has provisions for several use cases:

- 27 • preventing highly dynamic resources from being cached
- 28 • letting clients know for how long is acceptable to use cached data
- 29 • optimizing validation of cached entries to skip the download of the body  
30 if the copy on the client still matches the one on the server
- 31 • informing clients about resources that can be safely used even if stale when  
32 no connection is available and which ones must return an error

---

<sup>1</sup><https://html.spec.whatwg.org/multipage/webstorage.html>

<sup>2</sup><https://www.w3.org/TR/IndexedDB/>

<sup>3</sup><https://tools.ietf.org/html/rfc7234>

33 Caching is generally available only for the `GET` method and is controlled by the  
34 server for every single HTTP resource by adding the `Cache-control` header to its  
35 responses: this instructs the client (the web engine) on the ways it can store the  
36 retrieved contents and re-use them to skip the download on subsequent requests.

37 One of the most important uses of the `Cache-control` header is to disable any kind  
38 of caching on highly dynamic generated resources, by specifying the `no-store`  
39 value.

40 The `public` and `private` directives instruct clients that the resource can be stored  
41 in the local cache (`public` also allows for caching in intermediate proxy servers,  
42 a feature which is progressively getting obsolete as it conflicts with the confiden-  
43 tiality requirements of HTTPS/TLS).

44 The `Expire` header and the `max-age` directive let the server instruct the client for  
45 how long it can consider the cached resource valid. The client can completely  
46 skip any network access as long as the cached resource is “fresh”, otherwise it  
47 has to validate it against the server, but this does not mean that a complete  
48 re-download is always needed: using conditional requests, that is using the `If-`  
49 `Modified-Since` or `If-None-Match` headers to pass the values of the `Last-Modified`  
50 or `ETag` headers from the previous request, the download of the body is skipped  
51 if the values match and only headers will be transferred with a `304 Not Modified`  
52 response.

53 The HTML5 specification recently introduced the concept of [application cache](#)<sup>4</sup>  
54 which caters for an additional, higher-level use case: pro-actively downloading  
55 all the resources needed by an HTML application for offline usage.

56 This works by adding a `manifest` attribute to the `<html>` element of the main  
57 application page, and from there indicate the URL of a specially formatted  
58 resource that lists all the URLs the client needs to pro-actively retrieve in order  
59 to be able to run the application correctly when offline. The caching model  
60 used by this specification is somewhat less refined than the one used by the  
61 HTTP specification and for this reason it needs some special attention on how  
62 to ensure that the application is properly refreshed when changes are made on  
63 the server.

64 The more complex and powerful [Service Workers](#)<sup>5</sup> specification is meant to re-  
65 place this, but it is not supported yet by all modern browsers (works in Firefox  
66 and Chrome, WebKit and Edge don’t support it yet). The specification has  
67 been stable for more than a year, despite not being finalized yet. The WebKit  
68 team has not yet shown a clear interest in implementing it, which may be the  
69 reason why the specification is still in the current status.

---

<sup>4</sup><https://html.spec.whatwg.org/multipage/browsers.html#offline>

<sup>5</sup><https://www.w3.org/TR/service-workers/>

## 70 Caching in WebKit

71 WebKit currently has several caches:

- 72 • a non-persistent, in-memory cache of rendered pages which is set to 2  
73 pages if the total RAM is bigger or equal to 512MB
- 74 • a non-persistent, in-memory [decoded/parsed object cache](#)<sup>6</sup>, set to 128MB  
75 if the total RAM is bigger or equal to 2GB and progressively lowered as  
76 the amount of total RAM decreases
- 77 • a persistent, on-disk [resources cache](#)<sup>7</sup> of 500MB if there are more than  
78 16GB free on the disk, progressively scaling down to 50MB if less than  
79 1GB is available.

80 Those sizes are computed automatically but they can be customized to fit any  
81 requirements.

82 When a new resource needs to be cached WebKit makes sure that the upper  
83 bound is respected and frees older cache entries in a LRU pattern to make  
84 enough room to accommodate the resource which is about to be downloaded.

85 Downloaded contents to be stored in the on-disk URL cache are directly saved  
86 in the filesystem, using the normal buffering that the kernel does for every  
87 application to improve performance and minimize eMMC wear. This is further  
88 minimized by the fact that only contents marked for caching by the server using  
89 the appropriate HTTP headers will be cached: highly dynamic contents like  
90 news tickers won't be marked as cacheable so they won't impact the eMMC at  
91 all.

92 The application cache is handled separately and it is unlimited by default, but  
93 this is a setting that can be changed. All the resources are stored in a SQLite  
94 database as data blobs, except for audio and video resources where the only the  
95 metadata is stored in the database and the contents are stored separately.

96 To use the application cache effectively in WebKitGTK+ some implementation  
97 work would be required to limit the maximum size as the WebKit core hooks  
98 are currently not used by the WebKitGTK+ port, and the WebKit core itself  
99 does not currently provide any expiration policy for the cached contents.

## 100 Client/Server implementation strategies

101 Multiple strategies can be used to implement the previously defined system and  
102 affect the design of the client and of the contents offered by the portal server.

---

<sup>6</sup><https://trac.webkit.org/browser/webkit/releases/WebKitGTK/webkit-2.0/Source/WebKit2/Shared/CacheModel.cpp#L83>

<sup>7</sup><https://trac.webkit.org/browser/webkit/releases/WebKitGTK/webkit-2.0/Source/WebKit2/Shared/CacheModel.cpp#L158>

## 103 **Application cache**

104 The main HTML page of the portal links to an appcache manifest that instruct  
105 the browser to pro-actively fetch all the needed resources.

106 All subsequent accesses to the portal will be served from the cached copy, re-  
107 gardless of the availability of an Internet connection.

108 If the portal is accessed when an Internet connection is available, the browser will  
109 retrieve the appcache manifest from the server in the background and check for  
110 modifications: if a new version is detected the portal resources will be refreshed  
111 in the background and will be used for subsequent accesses to the portal.

112 Each application will have its own appcache manifest, so it will be locally cached  
113 after the first visit.

114 To ensure that the portal is available on first-boot even if no Internet connection  
115 is available, during the process of generating the system image the browser will  
116 be launched using a special mode that will cause it to connect to the portal, pop-  
117 ulate the application cache and exit as soon as the `ApplicationCache::updateready`  
118 event is fired. An ad-hoc program using WebKit may be used instead of adding  
119 a special mode to the browser.

120 This is the simplest and most portable approach on the client side, as all the  
121 caching logic is provided by the portal server using standard W3C mechanisms.

## 122 **Custom HTTP application caching server running locally**

123 Alternatively, the browser can be instructed to connect to a custom HTTP proxy  
124 server running locally instead of directly to the portal server.

125 Since TLS authentication cannot work appropriately through proxy servers, it is  
126 taken care by the proxy server itself, with the browser talking to the local proxy  
127 over unencrypted HTTP and the proxy converting HTTP requests to HTTPS.

128 This means that unencrypted communications will only happen locally between  
129 trusted components, while all the network traffic will be encrypted. Just like for  
130 any other HTTP error, the proxy can return error pages to the browser in case  
131 of TLS error (for instance, if the server certificate is expired) or return cached  
132 contents if available.

133 The custom proxy is then responsible for connecting to the portal server and  
134 retrieving updated contents from there, locally caching it with any kind of expiry  
135 and refresh policy desired, and processing cached resources when needed, for  
136 instance by rewriting links from HTTPS to HTTP.

137 The browser needs to be configured to reduce its own caching to a minimum,  
138 since the smart proxy already does it.

139 During the manufacturing process the proxy cache will be preloaded with the  
140 resources hosted by the portal server.

141 This is the most flexible approach.

142 **Separately-maintained locally accessible copy of the portal**  
143 **contents**

144 Instead of having a locally running custom HTTP caching proxy, the portal  
145 contents are stored as plain files on the system. The browser will contain custom  
146 logic to load the local HTML file instead of the portal URL when no Internet  
147 connection is available.

148 A separate process will periodically compare the locally-stored HTML file and  
149 resources against the portal server and refresh the local copy.

150 This is the least flexible choice, and the locally stored copies cannot be used as  
151 cache to speed up rendering when the connection to the Internet is available.