Web portal caching

Contents

Web portal caching

Introduction

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

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

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

Locally-stored copies should be used to speed up loading even when the connection to the Internet is available.

The portal and the applications store all their runtime data using the `localStorage`[1] or IndexedDB[2] mechanisms and how that is synchronized is out of the scope of this document, which instead focuses on how to manage static assets.

How HTTP caching works

Caching is a very important and complex feature in modern web engines to improve page load time and reduce bandwidth consumption. RFC7234[3] defines the mechanisms that control caching in the HTTP protocol regardless of its transport or serialization, which means that the same mechanisms apply to HTTPS and HTTP2 in the same way.

HTTP has provisions for several use cases:

- preventing highly dynamic resources from being cached
- letting clients know for how long is acceptable to use cached data

[1] https://html.spec.whatwg.org/multipage/webstorage.html
[2] https://www.w3.org/TR/IndexedDB/
[3] https://tools.ietf.org/html/rfc7234

- optimizing validation of cached entries to skip the download of the bodyi if the copy on the client still matches the one on the server
- informing clients about resources that can be safely used even if stale when no connection is available and which ones must return an error

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

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

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

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

The HTML5 specification recently introduced the concept of application cache[4] which caters for an additional, higher-level use case: pro-actively downloading all the resources needed by an HTML application for offline usage.

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

The more complex and powerful Service Workers[5] specification is meant to replace this, but it is not supported yet by all modern browsers (works in Firefox and Chrome, WebKit and Edge don't support it yet). The specification has been stable for more than a year, despite not being finalized yet. The WebKit

[4] https://html.spec.whatwg.org/multipage/browsers.html#offline
[5] https://www.w3.org/TR/service-workers/

team has not yet shown a clear interest in implementing it, which may be the reason why the specification is still in the current status.

Caching in WebKit

WebKit currently has several caches:

- a non-persistent, in-memory cache of rendered pages which is set to 2 pages if the total RAM is bigger or equal to 512MB
- a non-persistent, in-memory decoded/parsed object cache[6], set to 128MB if the total RAM is bigger or equal to 2GB and progressively lowered as the amount of total RAM decreases
- a persistent, on-disk resources cache[7] of 500MB if there are more than 16GB free on the disk, progressively scaling down to 50MB if less than 1GB is available.

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

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

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

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

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

Client/Server implementation strategies

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

[6] https://trac.webkit.org/browser/trunk/Source/WebKit2/Shared/CacheModel.cpp#L83

[7] https://trac.webkit.org/browser/trunk/Source/WebKit2/Shared/CacheModel.cpp#L158

Application cache

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

All subsequent accesses to the portal will be served from the cached copy, regardless of the availability of an Internet connection.

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

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

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

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

Custom HTTP application caching server running locally

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

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

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

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

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

During the manifactuing process the proxy cache will be preloaded with the resources hosted by the portal server.

This is the most flexible approach.

Separatedly-maintained locally accessible copy of the portal contents

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

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

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