Background Fetch Storage Design

John Mellor, Aug 2017

The Background Fetch API is similar to the Fetch and XMLHttpRequest APIs, except that requests are fetched in the background and automatically resumed across browser restarts, with the response/error eventually being delivered as a Service Worker event. This enables use cases like webapps that up/download video.

Hence, unlike Fetch and XHR, Background Fetch needs to persist serialized requests and responses (sometimes partial) to disk. The actual body of the response will be stored separately by the DownloadService (it's TBD where the request body gets stored).

The API groups together a list of requests into a Background Fetch *registration*, each of which has a BackgroundFetchOptions containing title, icons, etc for the notification that allows the user to control that registration.

Each ServiceWorkerRegistration can have many Background Fetch registrations, each of which can have many requests.

It makes sense to store this in the Service Worker database (SWDB) since all data is "owned" by a ServiceWorkerRegistration, and ideally deleting a SW would atomically delete its Background Fetch metadata in the same transaction, hence avoiding getting out of sync.

It should be possible to use the existing string-based user data mechanism by serializing custom protobufs to/from a string.

Lifecycle

Each request of a Background Fetch registration progresses through several states:

1. **Active (pending)**: not yet started.
2. **Active (before initial response)**: started, but the DownloadService hasn't yet responded with the download GUID, initial response headers etc ("initial" is just to distinguish this response from any subsequent responses if the download is interrupted then resumed, since response headers are discarded when resuming).
3. **Active (after initial response)**: the DownloadService is fetching this request.
4. **Completed**: the request has finished being fetched (successfully or with an error), and we have the full response.

A registration is completed – and an event delivered to the SW – once all of its requests have completed. If one request fails, the remaining ones will still be attempted, as partial results are often still useful.

Uniqueness of IDs

Developers pass in an DOMString “id” when creating or getting a Background Fetch registration, henceforth referred to as **developer_id**.

Unfortunately, these IDs can’t easily be relied on as primary keys, since it’s possible for JS to abort a BackgroundFetchRegistration (or wait for it to complete) then create another one with the same id, whilst holding onto a reference to the previous BackgroundFetchRegistration (or ActiveFetches/SettledFetches) objects. The methods on all of these objects must continue to work, i.e. stale data must not be deleted until the last object that refers to it is garbage collected, and aborting an old BackgroundFetchRegistration must not abort a newer one with the same developer_id, etc.

So instead, we will generate a **unique_id** for every Background Fetch registration, and use that as our primary key throughout the Background Fetch code (from the Blink BackgroundFetchRegistration objects which will internally store their unique_id to disambiguate stale vs active ones, all the way to the Delegate that talks to the DownloadService and will group notifications according to their unique_id).

This will ultimately make the code a lot simpler, as consistently using these unique_ids also removes various race conditions between Background Fetch registrations completing/aborting and new ones being created with the same developer_id.

(And if we ever moved to global storage instead of or in addition to per-ServiceWorkerRegistration storage in the SWDB, unique_ids would have the added benefit of being globally unique, so we wouldn’t need to concatenate them with the profile ID and service_worker_registration_id in order to use them as keys in a global map).

Important operations

Key: *sw_reg_id* is an int64_t service_worker_registration_id; see above for *developer_id* and *unique_id*.

Called once per startup (if there are any fetches)

// Loads the options (title, icons, etc) of each Background Fetch registration that has active
// requests. Additionally loads a list of active download GUIDs of all active requests for each
// of those Background Fetch registrations. Together this is used to show/update the
// downloading notifications, properly grouped by origin/registration, and for two-way sync
// against the DownloadService’s list of GUIDs, i.e. abort any that only the DownloadService
// persisted, and start any that only Background Fetch persisted.
1. GetOptionsAndDownloadGUIDsForActiveRegistrations(); NOT DONE

Called once per created registration

// Creates a new Background Fetch registration given id, options (title, icons, etc) and the list
// of requests.
2. CreateRegistration(sw_reg_id, developer_id, options, request_info_vector); DONEISH

Called on demand by API

// List registration IDs, and get options (title, icons, etc) by id. Both exclude registrations
// for which MarkRegistrationForDeletion has already been called.
3. GetIds() & GetRegistration(sw_reg_id, developer_id) NOT DONE

Called once per downloading/completed request

// Picks and loads the next BackgroundFetchRequestInfo object marked pending, in
// global FIFO order (TODO: smarter scheduling).
4. GetNextPendingRequestInfoAndMarkItActive(); INFLIGHT CL 2

// Stores response headers, and response code/text for an active request.
5. WriteActiveDownloadMetadata(sw_reg_id, unique_id, request_index, metadata); NOT DONE

// Stores file path/size, url chain, etc for a completed request. Marks it as no longer active
// by deleting the download GUID.
6. WriteCompletedDownloadResponse(sw_reg_id, unique_id, request_index, response); NOT DONE

Called once per finished registration

// Loads completed data for a Background Fetch registration so it can be
// delivered in a Service Worker event.
7. ReadCompletedRegistration(sw_reg_id, unique_id); NOT DONE

// Marks that the backgroundfetched/backgroundfetchfail/backgroundfetchabort event
// has been dispatched. It would be nice to just call DeleteRegistration at this point, but
// unfortunately if JavaScript holds a reference to a BackgroundFetchRegistration
// object we need to keep the corresponding data around until the last such reference
// is released (or until shutdown) – see Uniqueness of IDs. And we can't just move the
// Background Fetch registration's data to RAM as it might consume too
// much memory. So instead this step disassociates the developer_id from the unique_id, so
// that existing JS objects with a reference to unique_id can still access the data, but it can
// no longer be reached using GetIds or GetRegistration.
8. MarkRegistrationForDeletion(sw_reg_id, developer_id, unique_id); DONE

// Deletes all data for a Background Fetch registration. Called when the last JS reference to

// a BackgroundFetchRegistration (or ActiveFetches/SettledFetches) for unique_id is garbage
// collected. MarkRegistrationForDeletion must already have been called for this unique_id.
9. DeleteRegistration(sw_reg_id, unique_id); DONE

// Deletes inactive registrations marked for deletion. Called when the browser restarts.
10. GarbageCollect(); DONE

Proposed Schema

(This is all per ServiceWorkerRegistration, as usual for the SWDB user data mechanism.)

Each Background Fetch registration has a BackgroundFetchOptions metadata, a developer-chosen developer_id, and a list of requests/responses.

Background Fetch registrations and their requests are stored side by side rather than nesting the requests within their registration – this avoids having to load all of a Background Fetch registration's requests/responses into memory at once, and enables efficiently querying only the active requests, or the next pending request.

Key:
Done: no symbol
Not done: ❌
In Progress: 🐙 (by John) 🐙 (by Dan)

user data key	**type (all protobufs serialized to/from strings)**	**written by**	**deleted by**	**read by**
"bgfetch_active_unique_id_" + developer_id	{unique_id}	2	8	2, 3, 8, 10
"bgfetch_registration_" + unique_id	{unique_id, developer_id, origin, BackgroundFetchOptions}	2	9	❌1, 3 🐙4, 10
"bgfetch_request_" + unique_id + "_" + request_index	mojom::FetchAPIRequest, with a file path in place of the request body blob.	2	9	❌3, 🐙4, ❌7
"bgfetch_pending_request_" + *creation_timestamp* + "_" + unique_id + "_" + request_index	{unique_id, request_index}	2	🐙4, 8	🐙4
"bgfetch_active_request_" + unique_id + "_" + request_index	{download_service_guid, unique_id, request_index}	🐙4	❌6, 8	❌1

"bgfetch_initial_response_" + unique_id + "_" + request_index	{response_code, response_text, response_headers}	❌5	❌9	❌3,❌7
"bgfetch_completed_response_" + unique_id + "_" + request_index	mojom::FetchAPIResponse, excluding {response_code, response_text, response_headers}	❌6	❌9	❌3,❌7

Explanation

"bgfetch_pending_request_" and "bgfetch_active_request_" are split out to allow efficiently querying only the pending/active requests respectively – see Usage below.

"bgfetch_initial_response_" and "bgfetch_completed_response_" are split out from "bgfetch_request_" to allow adding their information without having to read+parse+update "bgfetch_request_" (which would be less efficient, but not the end of the world). Also, "bgfetch_completed_response_" could be extended to allow efficiently querying only completed requests, which may become useful in future.

The reason that `unique_id` and `request_index` are repeated in the types despite being present in the key is that GetUserDataForAllRegistrationsByKeyPrefix only returns a list of (`sw_reg_id`, `user_data_value`) pairs. We could avoid this redundancy by having a variant that also returns the `user_data_key`s and parse the key suffixes to extract the `unique_id` and `request_index`.

Usage

The trickiest operations to implement are #4 GetNextPendingRequestInfoAndMarkItActive and #1 GetOptionsAndDownloadGUIDsForActiveRegistrations:

With this schema, #1 GetOptionsAndDownloadGUIDsForActiveRegistrations could be implemented as:

1. Call GetUserDataForAllRegistrationsByKeyPrefix for key prefix "bgfetch_active_request_" (with no `limit`).
2. For each of the unique registrations amongst these requests:
    a. Call GetRegistrationUserData to get the "bgfetch_registration_" value.
3. It'll also need to do garbage collection and consistency checks, such as:
    - Restarting downloads for requests with a bgfetch_active_request_ that the DownloadService has forgotten about.
    - Deleting all inactive registrations (normally these are deleted once the refcount of BackgroundFetchRegistraton V8 objects goes to zero, but browser shutdown can prevent that).

With this schema, #4 GetNextPendingRequestInfoAndMarkItActive could be implemented as:

1. Add a `limit` parameter to GetUserDataForAllRegistrationsByKeyPrefix and make it only return the first `limit` values matching the key prefix, in lexicographic order (this is just an early out from the LevelDB iteration it already does).
2. Call GetUserDataForAllRegistrationsByKeyPrefix for key prefix "bgfetch_pending_request_" with a limit of 1. This will get the earliest request, since the keys are sorted in order of creation_timestamp.
3. ~~Skip requests that have already been started during the lifetime of this browser process and are awaiting a call to #4 WriteActiveDownloadMetadata to mark them as no longer pending.~~
4. Call GetRegistrationUserData to get its "bgfetch_request_" value.
5. Call GetRegistrationUserData to get its "bgfetch_registration_" value.

Note that since the SWDB lays out its keys intelligently, neither of these queries has to page in SWDB data for SWs that don't have a "bgfetch_active_request_" or "bgfetch_pending_request_" user data key respectively.

In both cases, this would all need to be repeated for each StoragePartition in the relevant BrowserContext, since each StoragePartition has a separate Service Worker Database. Initially we'd just support BrowserContext::GetDefaultStoragePartition, but we'll need to track these somehow once Site Isolation uses different StoragePartitions for different origins, or to support Background Fetch from extensions.

This all carefully avoids reading in unnecessary request data, even for requests belonging to the same BackgroundFetchRegistration, as that would use unnecessary RAM. We should only have all the request data for a given BackgroundFetchRegistration loaded simultaneously a) when the webapp calls BackgroundFetchManager.fetch and b) when delivering the backgroundfetched event to the SW.