Supporting Active Fetches for Background Fetch

nator@, Oct 15, 2018
Contributors: rayankans@, beverloo@, jakearchibald@

Requirements:

Currently, a developer only gets access to the downloaded content once the fetch has completed. Access is always through match() and matchAll() on the BackgroundFetchRegistration object.

We'd like to enable access to downloaded content as soon as it's available. This allows for developers to start playing a podcast, for instance, once some percentage of the audio resource has downloaded. Access to active fetches should be intuitive, and easy.

Terminology

A fetch is active when we have started showing UI for it but haven't finished downloading content.

Considerations

When should response be available?

The BackgroundFetchRegistration objects expose match() and matchAll(). These return BackgroundFetchRecords. Each record has a request and a (promise of a) response.

When the data for a given request is still being downloaded, is there benefit in exposing whatever's been downloaded in a response object?
Yes. Many podcasts are streamed, and the content will be downloaded as a single large file. Movies, on the other hand, are generally sent down as fragments. To support being able to play podcasts while they're being downloaded, we need to expose access to responses that are still being downloaded.

Chrome processes downloads via the DownloadManager. Only once a response has completed downloading is it transferred to the Cache Storage. Therefore, exposing in-progress downloads will require changes to the DownloadManager. Perhaps, it can write into a blob that Cache Storage can consume, and then the blobhandle can be shared with the renderer as soon as it's created. Since this is a wider change, it makes sense to implement access to active fetches in two stages:

a. Stage 1: Support access to data for completed requests in the active fetch.
b. Stage 2: Support access to data for all requests in the active fetch.

Let's focus on Stage 1 in this document.

The multiple access problem

The tricky bit with accessing in-progress responses is that data can be accessed from multiple sources. Luckily, this is read-only.

One option is to keep a list of BackgroundFetchRecords for every request in that registration, then we can simplify things greatly. By returning the same record/blob for multiple calls.
This has two issues:

1. It makes match() and matchAll() inconsistent with Cache API's match() and matchAll(), which return (a) new independently-consumable response(s) each time they're called. (https://jsbin.com/cipavob/edit?js,console for reference)
2. It breaks retries for responses, when a server doesn't support resuming of downloads. For instance, if a download of a podcast is ongoing, and we're streaming it, and the user goes offline. The next time the user comes back online, the download is retried, but the server simply sends the entire response from the beginning, so we start downloading again. Now do we replace the existing response with one containing less data? That breaks the ongoing streaming. (Spec bug to call out the fact that responseReady getting rejected != the fetch failing).

Therefore, we need to return new BackgroundFetchRecord(s) everytime match or matchAll is called.

Once we've implemented this, the following will hold:

```
const matches = await Promise.all([
  registration.match('resources/feature-name.txt'),
  registration.match('resources/feature-name.txt')
]);

assert_equals(matches[0], matches[1]);  // will be FALSE
```

And

```
const responses = [
  await matches[0].responseReady,
  await matches[1].responseReady
];

assert_equals(responses[0], responses[1]);  // will be FALSE
```

However, the response text can only be consumed once, from one record object. Different record objects allow consumption of the same content independently.

```
const content = [
  await responses[0].text(),
  await responses[1].text()  // won't throw.
];
```

The increased lifetime problem

Once match() and matchAll() support access to active fetches, they'll be called from the Document context as well as ServiceWorker context. Previously, when we allowed access to downloaded content only once the background fetch had completed, we could safely delete the data corresponding to the fetch once the completion event had gone out of scope. Now, BackgroundFetchRecords can potentially be kept around indefinitely from the Document context.

<u>Do we delete them?</u>
BackgroundFetchRecord objects hold a request and a response. The former can hold large content in case of upload requests (body), and the latter can hold a reference to large content in case of download requests, which prevents that data from getting deleted.
For the sake of argument, also consider that if the developer leaves these records in scope after they're needed, they're leaking memory which will get attributed to Chrome. However, they can already do so by, for instance, doing
new Uint8array(massiveNumber)
and keeping a reference to it.
(This might expose garbage collector behaviour by querying available quota and monitoring when it decreases, but that's an issue that already exists for fetch())
One might also expect these records to be available if they're in scope, and deleting them might be unexpected.

On the other hand, keeping these around long after the fetch has settled doesn't make sense.

If we should, when should we delete data?

The Cache Storage implementation keeps track of all references to the downloaded content held in blobs, and deletes the stored data once all references to it have gone. Once the completion event goes out of scope, we can delete the request and response from the BackgroundFetchRecord objects. The skeleton BackgroundFetchRecord objects can be left until the Document goes out of scope.

To implement this, we'd need the BackgroundFetchRecord object to be notified when the completion event goes out of scope.

CONCLUSION: Keep the BackgroundFetchRecord objects for the lifetime of the scope. Once they go out of scope, their destructors will be called, releasing any references to the stored data which will in turn, cause the Cache Storage to delete it.

responseReady

We'd like to return a responseReady promise that gets resolved with the correct response once the request has completed. To enable that, we'd need to either:

a. Keep track of all active BackgroundFetchRecord objects for the registration, or,
b. Have the BackgroundFetchRecord objects listen to an update from the BackgroundFetchRegistration object that created them. These updates will notify completion of requests, and if the request corresponds to BackgroundFetchRecord's request, then they can resolve the responseReady promise, else ignore the update.

Duplicate URLs

The current implementation blocks requests with duplicate URLs within a background fetch. We plan to remove this check in the future, to support uploads.
A layer will be added to Cache Storage to add an extra identifier to make these requests distinct for Cache Storage. This extra identifier can also be used to disambiguate request completion updates send from BackgroundFetchRegistration to all listening BackgroundFetchRecord objects.

Details of this identifier are out of the scope of this document, and will be covered elsewhere.

match() and matchAll()

1. This should work once we update the MatchRequestTask to query registration user data by all three key prefixes: active, pending, and completed, for the unique_id. This'll make sure we match requests in all states and return responses for them, wherever possible.

2. There's an additional check on the renderer side to reject match() and matchAll() if the fetch hasn't completed yet. This should be taken out.

onProgress events

These should continue working without any extra work, since we already send these whenever 'downloaded' changes.

However, we can reuse the registration notifier setup to also update the registration object on the renderer side with completed response for a request, once it has been downloaded. This can in turn be used to notify the relevant BackgroundFetchRecord objects listening for this, so they can resolve any pending responseReady promises.

To achieve this, we would need the following:

1. A new notifier method to update the BackgroundFetchRegistration object in the renderer. This would be something like OnRequestCompleted(blink::mojom::FetchAPIRequest request, blink::mojom::FetchAPIResponse response).
   This does send a completed response. This is what the Cache API implementation does. Note that this object contains a BlobHandle, not the entire data blob.
2. A BackgroundFetchRegistrationObserver class, which BackgroundFetchRecord would inherit from. BackgroundFetchRegistration would have a list of observers, which all be notified in response to OnRequestCompleted.
3. Each BackgroundFetchRecord will have logic to response to OnRequestCompleted. This will be a no-op if the request in the update doesn't correspond to the record's request. If it does, the Record will attempt to resolve any pending responseReady() promise, via its SetResponseAndUpdateState() method.

recordsAvailable

This is currently set to false once the Background Fetch completion events go out of scope. This logic doesn't have to be updated to support access to active fetches.

Lifetime Management

This could've been the hardest part of supporting access to active fetches, if we had to manage the lifetime of request-response pairs ourselves. Luckily, the Cache Storage API gives us that for free, and will delete data stored in the Cache Storage once the last reference to it is released.
To test: Will that work across browser restarts?

Quota Management

Currently, when a fetch is completed and developers get notified of the same, they have about five minutes (service worker timeout) to copy that data to a private cache. Potentially, for the duration of this timeout, there could be a double quota penalty for the user, since both copies are charged against the same quota account. Since this timeout is time-bounded, we were okay with that for v1 of Background Fetch.

However, giving access to ongoing fetches increases this time window to the lifetime of the fetch + service worker timeout, and we must hence do the correct accounting.

Permissions

When a background fetch is started from a non-top level frame, we start the fetch in a paused state and wait indefinitely until the user explicitly resumes it. The developer might not understand what's going on if they keep getting records for this fetch but no data gets downloaded for them. I don't have a good solution in mind here, except that we should have some written guidance about this common caveat.

We can consider failing these fetches after a set (long) amount of time if the user doesn't take any action, so the developer can try again later.

BackgroundFetchSettledFetch

Since we'd not be interested in just settled fetches, this name would no longer be appropriate. It'll be a widespread name change, and methods such as GetSettledFetches* would need to be renamed too.

Tests

We'll need to add tests to ensure any downloaded data is being exposed appropriately, and doesn't go away upon browser restart.