Partitioned Service Workers Design (public version)

This Document is Public

Authors: wanderview@chromium.org
April, 2021

One-page overview

Summary

This design describes our proposal for partitioning service workers in 3rd party contexts. The motivation for these changes can be found in the explainer.

Platforms

Mac, Windows, Linux, Chrome OS, Android, Android WebView.

Team

mailto:service-worker-discuss@chromium.org

Bug

crbug.com/1191114

Code affected

Service workers

Design

This design describes our proposal for partitioning service workers in 3rd party contexts. The motivation for these changes can be found in the explainer.

Assumptions and Dependencies

StorageKey

This design assumes that there will be a `StorageKey` type as outlined in the Storage Partitioning Design. This type will include both the origin and information about the top level window of the tab. The top-level window information may be the window's site (scheme+etld+1) or the first-party-set owner.

Since we have not decided whether to commit to one or the other this design will be written to support either interchangeably. This is somewhat easy since they can both be expressed as a URL-like string.

This design assumes that the `StorageKey` will be accessible from the `RenderFrameHostImpl`, `DedicatedWorkerHost`, and `SharedWorkerHost`. This is largely true today if the key ends up being the top-level site. If first-party-set is used for the key, however, additional plumbing may be needed.

Exposing `StorageKey` on the `ServiceWorkerHost` will be part of this design.

Feature Flag

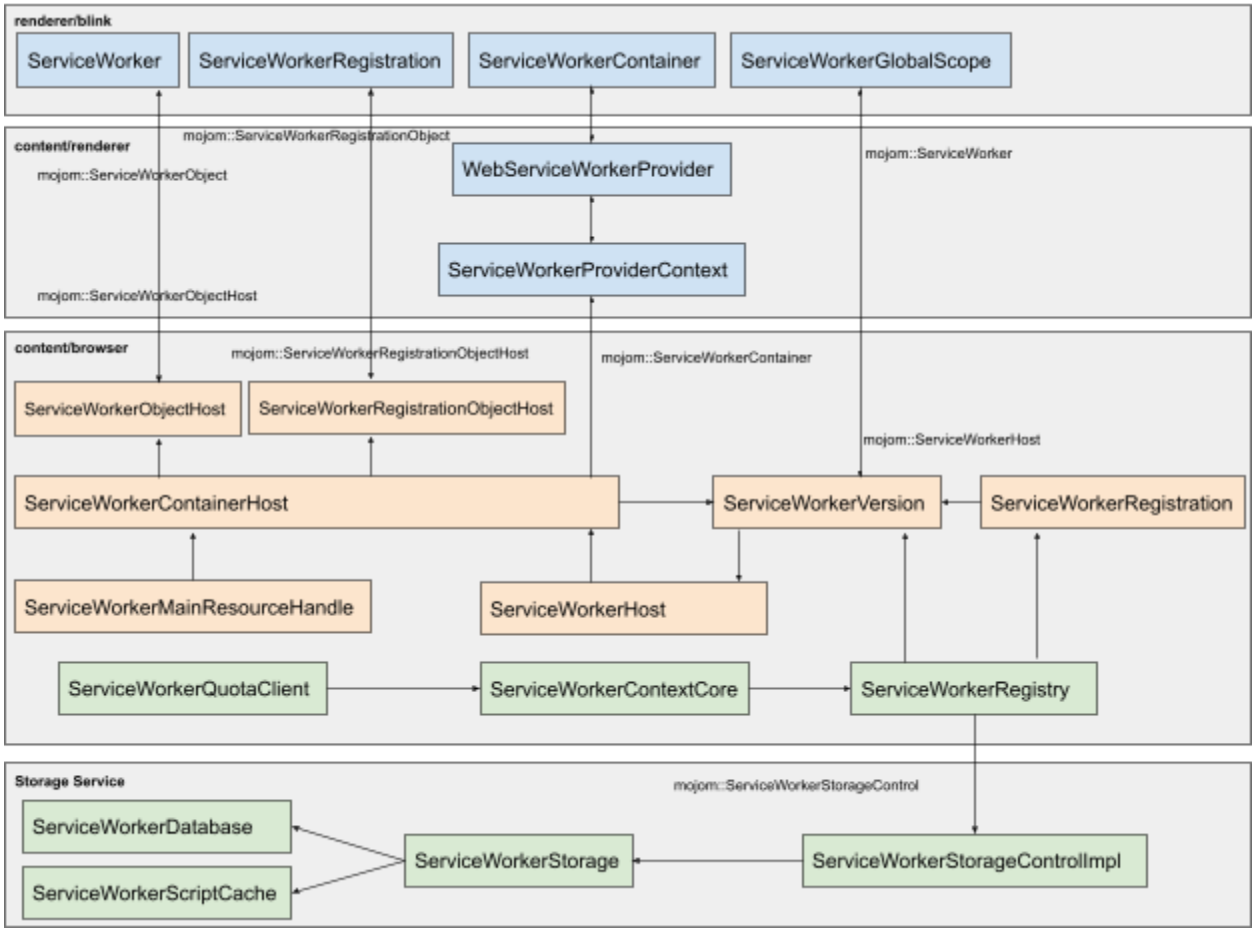
In addition to sharing the `StorageKey` with the overall Storage Partitioning effort, we will also share a single feature flag for enabling or disabling partitioning. It's important that we either partition all APIs or none of them in order to avoid confusing websites.

QuotaManager

This design also depends on the `QuotaManager` and `QuotaClient` internal APIs being updated to accept a `StorageKey`. This will be necessary to properly integrate with the quota system. This work is part of the Storage Partitioning Design.

Service Worker Overview

The service worker subsystem has code stretching across the browser process, renderer process, and the storage service. The following diagram provides a high level view of this code.



Much of this effort will focus on integrating the `StorageKey` into these classes. To that end the diagram has been color coded by how the code interacts with the `StorageKey`.

Classes marked in orange need to be conceptually locked to a single `StorageKey`. These classes already have a `url::Origin` associated with them directly or indirectly. We will need to augment or replace these origin attributes with `StorageKey`.

The classes marked green represent code that will need to understand `StorageKey`. For some classes this will simply be a pass-through. For others, like the `ServiceWorkerDatabase`, we will need to make more extensive changes to persist the `StorageKey` on disk.

Classes marked in blue should not need to be modified.

ServiceWorkerContainerHost

The creation of the `ServiceWorkerContainerHost` is the entrypoint for most of the service worker machinery at runtime. An instance is created for each window, iframe, and worker in the browser. This is done via the `ServiceWorkerMainResourceLoaderInterceptor`. The

interceptor creates an empty `ServiceWorkerContainerHost` at first. Later, the `updateUrls()` method is called to identify the newly created context.
The `updateUrls()` method currently sets the load URL, `net::SiteForCookies`, and optionally a top-frame `url::Origin`. This is where we will introduce the `StorageKey`; either by deriving from the currently available information or requiring it to be passed in.

For example, one of the places `updateUrls()` is called is from `ServiceWorkerControlleeRequestHandler::InitializeContainerHost()`. In this case there is a `network::ResourceRequest` available which in turn exposes a `NetworkIsolationKey`. We should be able to convert the `NetworkIsolationKey` to a `StorageKey`.

Once we have the `StorageKey` in `ServiceWorkerContainerHost` it can be directly passed on to the `ServiceWorkerObject` and `ServiceWorkerRegistrationObject` instances as well.

The `StorageKey` also allows `ServiceWorkerContainerHost` to use it when executing mojo methods invoked from the web API. For example, `navigator.serviceWorker.register()` invokes `ServiceWorkerContainerHost::Register()`. The `StorageKey` can be automatically included and passed along in the call to `ServiceWorkerContextCore::RegisterServiceWorker()` in order to properly persist the key with the new registration. This also applies to other methods like `GetRegistrations()`, etc.

The fact that the `StorageKey` is locked to `ServiceWorkerContainerHost` in the browser process is an important security feature. It prevents a compromised renderer process from lying to us about the `StorageKey` in order to bypass partitioning.

In addition, adding `StorageKey` to `ServiceWorkerContainerHost` helps implement the Clients API. This API allows a service worker to find window and worker contexts in order to message them, etc. This operation is ultimately supported by `ServiceWorkerContextCore::GetClientContainerHostIterator()` which iterators over a map of `ServiceWorkerContainerHost` objects. Currently this only filters on origin, but we will be able to also now filter on the `StorageKey` of the host object.

ServiceWorkerRegistry

The `ServiceWorkerContextCore` and `ServiceWorkerRegistry` are mainly singleton classes that handle operations across many different origins, registrations, etc. In general the `StorageKey` will need to be plumbed through the majority, if not all, of these operations.

The `ServiceWorkerRegistry` is responsible for creating `ServiceWorkerVersion` and `ServiceWorkerRegistration` objects. It also depends on the `ServiceWorkerDatabase` and `ServiceWorkerScriptCache` for persisting data to disk.

ServiceWorkerVersion

Some of the core capabilities of the `ServiceWorkerRegistry` is to register new service workers and to look up old existing workers. When this happens it instantiates a `ServiceWorkerVersion` object. Each service worker registration supports the concept of a changing worker script. When this happens the registration progresses through installing, waiting, and activated phases. The `ServiceWorkerVersion` object represents a particular instance of the worker script for a registration.

Currently the `ServiceWorkerVersion` contains a `url::Origin` member. This will need to be enhanced with a `StorageKey` passed from the `ServiceWorkerRegistry`. This is then used to pass back into the registry for various operations; e.g. triggering an update check.

The `ServiceWorkerVersion` also implements methods like `GetClients()` in order to support the Clients API. These methods will need to pass on the `StorageKey` into the associated algorithm in order to compare to the key associated with each `ServiceWorkerContainerHost`.

Again, having the `StorageKey` bound in the browser process object helps secure the clients API by preventing the renderer from lying about the value.

The `ServiceWorkerVersion` will also need to pass the `StorageKey` to the `EmbeddedWorkerInstance` when starting the worker thread. This will then pass just the top-level site from `StorageKey` into the `IsolationInfo` as the top-level origin. This information is necessary to properly isolate information in the network layer, http cache, cookies, etc.

Note, we pass just the top-level site here even though `IsolationInfo` wants the full origin. We do this because the service worker may be started for pages embedded under different origins that are same-site, but we can only possibly remember the top-level origin for the page that originally registered the service worker. Indeed, pages with different top-level origins could all be controlled by the service worker at the same time. In truth we only isolate by site, so we pass the site instead of the origin here. This will produce the most predictable outcome.

If a user attempts to block cookies for a particular subdomain of the top-level origin, however, we may miss blocking those cookies. Of course, even if we passed the full top-level origin we still risk missing cookies to block if the target subdomain differs from the one that registered the service worker and we also have less predictable behavior.

Currently we get the top-level site from StorageKey. If the `StorageKey` changes to contain the first-party-site owner instead we may need to persist and plumb the `top_level_origin_` member from `ServiceWorkerContainerHost`. We may also need to perform these changes if we decide it's better to use the top-level origin of the page that registered the service worker.

clients.openWindow()

One special API that requires particular attention is `clients.openWindow()`. This can be called after a user clicks on a notification triggering a `notificationclick` event in the service worker. When called, `openWindow()` will create a new top level window for the given URL. Typically this involves creating a new browser tab, etc.

We must be careful not to allow a partitioned service worker to create an unpartitioned top level window. This would allow the service worker to exfiltrate data out of the partition defeating our privacy goals.

Therefore, this design proposes that `clients.openWindow()` should reject with an `InvalidAccessError` exception if the `StorageKey` indicates the service worker is in a partition. This check should be added in the `ServiceWorkerVersion::OpenNewTab()` method in the browser process.

This may seem to block a useful feature, but as discussed below under "related APIs", the notification permission is currently not granted in 3rd party contexts. Therefore this scenario should not really happen. We should add the check, however, as an extra safety belt in case the permission is relaxed in the future.

If 3rd party notifications do get allowed in the future we can revisit this decision and explore relaxing the `openWindow()` block. For example, we could pursue creating a top level window that is also partitioned. That would be a bit weird, though, so we don't want to take on that work until necessary.

ServiceWorkerHost

In addition to basic `StorageKey` plumbing, the `ServiceWorkerHost` will also need to be modified to return the correct `NetworkIsolationKey`. This should be computable from the information stored in the `StorageKey`.

ServiceWorkerQuotaClient

The `ServiceWorkerQuotaClient` lives in the browser process and implements the abstract `QuotaClient` interface. It uses `ServiceWorkerContextCore` to send quota related requests down to the `ServiceWorkerRegistry` and `ServiceWorkerStorage` set of objects.

In general `QuotaClient` interface methods that operate on origins will need to be modified to operate on `StorageKey` partitionings instead. This work to update the interface is covered under the Storage Partitioning effort. The service worker effort will mainly need to implement the modified interface and pass the `StorageKey` on to the context and registry classes.

ServiceWorkerDatabase

The `ServiceWorkerDatabase` class is responsible for persisting all registration information to disk. This is done by storing the data in a leveldb database. Our main goal will be to enhance places where the origin is stored in the database with the `StorageKey`.

The origin shows up in rows with two different kinds of keys. These keys are prefixed with either`REG:` and `INITDATA_UNIQUE_ORIGIN:`.

The `REG:` prefix keys map to a protobuf value containing the main registration info. The overall structure of the `REG:` key is currently:

```
"REG:" + <origin> + "\x00" + <registration id>
```

There are a few ways to modify this key to support our needs. While not strictly necessary, though, we would like to modify the key in a way that keeps 1st-party registrations unmodified. This means that 1st-party key entries will be backward compatible with the legacy code. This is an important advantage if we find we need to rollback for problems encountered during deployment.

Therefore we propose to change the key to:

```
"REG:" + <origin> + [ "," + <storage key> ] + "\x00" + <registration id>
```

Basically we append a comma and the storage key to the origin when partitioned. If the registration is for a 1st-party context then we do not append anything and the key looks like it did previously.

Note, by "storage key" here we mean only the non-origin parts of the `StorageKey`. We do not want to duplicate the origin in the leveldb key.

This gets us mostly backward compatible. Entries for 1st party service workers will work with the old code. Some operations, such as `GetOriginsWithRegistrations`, may need to be modified to be more forgiving of invalid origins. This can be done in advance of storing any data in the new database schema, however.

Another advantage of this key structure is we don't need to run any migration on the overall service worker database. 1st party service workers continue to be stored and loaded as they used to. We only need to start storing entries in the new format when we enable partitioning.

The `REG:` entries are used when finding a service worker with a matching scope. The `StorageKey` will have to be plumbed through `ServiceWorkerStorage::FindForClientUrlInDB()` and passed to `ServiceWorkerDatabase::GetRegistrationsForOrigin()`. The `StorageKey` will then be passed to `CreateRegistrationKeyPrefix()` to be used to create the key prefix used to seek into the leveldb. The iteration through the known registrations may have to be modified to ignore partitioned entries when the feature flag is disabled.

The other relevant database entry looks like:

```
"INITDATA_UNIQUE_ORIGIN:" + <origin>
```

These entries have an empty value. They essentially exist as keys in order to index the known origins in the database.

To be consistent with the proposal for the previous entry type we propose that these keys also get an appended command and storage key when partitioned. If the entry is for a 1st-party origin then nothing is appended.

Again, only the non-origin parts of the storage key are appended to avoid duplication.

```
"INITDATA_UNIQUE_ORIGIN:" + <origin> + [ "," + <storage key> ]
```

It's somewhat unfortunate that the key still has the string "origin" in it when it might be better named "partition", but it does not seem worth triggering a full migration of the database for this one issue.

In addition to these two keys we must also modify the value of another entry type. The `REGID_TO_ORIGIN` entry maps the registration id to an origin value. Again we propose to append `,<storage key>` to the value to be consistent with the other changes.

Note, this proposal favors minimal migration and backward compatibility at the cost of increased complexity. The code must understand what it means when there is or is not a comma delimited value. Another approach would be to use a more fixed schema structure where we always have a key field for the partition key. This would require migrating the

database structure. The downsides of a migration are that it is a costly one-time operation that can be noticeable to users and it requires a reverse migration process if we need to rollback.

Given the incremental nature of the data we are adding it does not seem worth it to do a migration here. In addition, the service worker pattern scopes effort will likely be performing a database schema migration later this year. We can make more structural changes to better accommodate partitioning at that time if necessary.

ServiceWorkerScriptCache

The diagram greatly simplifies the `ServiceWorkerScriptCache` and its related operations. Because of the way disk operations are located in the storage service and script loading operation is in the browser process there are actually a number of different mojo interfaces at work.

For the purposes of this design, however, the important thing to understand is that each script stored in the cache is given a unique 64-bit integer id. The registration then stores a mapping of script URLs to cache identifiers. This mapping is maintained in the `ServiceWorkerScriptCacheMap` in the browser process. Each `ServiceWorkerVersion` owns a separate `ServiceWorkerScriptCacheMap` for its particular resources.

What this all means is that there is no additional work to partition the `ServiceWorkerScriptCache` and its associated code. Since each `ServiceWorkerVersion` already stores its own mapping of resource URL to unique identifier the script cache gains partitioning automatically due to the `ServiceWorkerVersion` being partitioned.

Related APIs

There are a number of APIs associated with service workers. We analyze these below, but in general they do not need modification. They generally associate data with the registration and our proposed changes above partition registrations. Therefore these APIs get partitioning for "free".

Push Notifications

The push notification API allows pages to subscribe for push notifications using an existing service worker registration. This invokes the `PushMessagingManager::Subscribe()` method which tasks a `service_worker_registration_id` identifying the associated registration. Later this id value is used to look up the registration in `PushMessagingRouter::StartServiceWorkerForDispatch()` in order to fire a `PushEvent` on the worker.

Since this system uses a unique identifier to find the associated registration there should not need to be any modifications to support partitioning. The registrations will already be partitioned resulting in different identifiers.

In addition, this code is unlikely to be exercised for 3rd party partitioned service workers anyway. Currently we do not allow 3rd party contexts to request the push notification permission.

As discussed above, notifications also interact with the `clients.openWindow()` API. To be extra cautious we will add a check in `clients.openWindow()` and reject the operation if it's called in a partitioned service worker.

Background Sync

The background sync API allows a page to register a callback to be invoked when the browser detects the network connection has come back online.

When the `SyncManager.register()` method is called a mojo message is sent to `OneShotBackgroundSyncServiceImpl::Register()` in the browser process. This method includes the registration identifier, similar to push notifications above. The registration id is persisted and later used Later `BackgroundSyncManager::FireReadyEventsImpl()` uses this identifier to find the registration by calling `ServiceWorkerContextCore::FindReadyRegistrationForId()`.

Again, similar to push notifications above this should automatically work correctly with partitioned service workers because the identifiers will be associated with specific registrations. Since the registrations are partitioned the background sync operation will be as well.

Also, background sync is similarly prevented from granting necessary permissions in 3rd party contexts. So this code is unlikely to get exercised unless that restriction is relaxed.

Background Fetch

The background fetch API allows a page to initiate large downloads that run in the background. When the downloads complete they may fire events on the service worker.

Initiating a background fetch triggers `BackgroundFetchBridge::Fetch()` in the renderer which then sends a mojo message to `BackgroundFetchServiceImpl::Fetch()`. This passes the service worker registration id value and associates it with the operation. Later the id is used to find the registration in `BackgroundFetchEventDispatcher::LoadServiceWorkerRegistrationForDispatch()`.

Again, similar to the other APIs this should automatically work with partitioned service workers. Since the API is based on the identity of the associate registration and registrations are already partitioned the partitioning policy will be honored.

This API is slightly different, however, in that it is usable from a 3rd party context. It does not require a permission and therefore is not blocked.

Payments

Payments API appears to also be based on registration identifiers, so it should also just work with partitioning.

Primary Use Case Coverage

This section ties the proposed changes above to the primary use cases for service workers.

Registering a New Service Worker

When a page wants to register a service worker they call the `navigator.serviceWorker.register()` method. This will send a mojo message to the `ServiceWorkerContainerHost` which will then use its `StorageKey` to invoke `ServiceWorkerContextCore::RegisterServiceWorker()`.

Controlling a Context on Navigation

Another primary use case is controlling a new window when the browser navigates to its document URL.

Service workers tap into the navigation process via the `ServiceWorkerMainResourceLoaderInterceptor` which calls into `ServiceWorkerControlleeRequestHandler::MaybeCreateLoader()`. This code then invokes `ServiceWorkerContainerHost::UpdateUrls()` via `InitializeContainerHost()`.

At this point the `ServiceWorkerContainerHost` now has its load URL and `StorageKey`. The `ServiceWorkerControlleeRequestHandler` then attempts to find a matching registration by calling `ServiceWorkerRegistry::FindRegistrationForClientUrl()`. The `StorageKey` will need to be passed into this method to correctly partition the resulting controlling service worker.

Clients.matchAll()

When the service worker `clients.matchAll()` API is called a mojo message is sent to the `ServiceWorkerVersion::GetClients()` method. This will use the `ServiceWorkerVersion`'s associated `StorageKey` to iterate through the `ServiceWorkerContainerHost` list to find matching clients in the same partition.

Clients.claim()

When the service worker `clients.claim()` API is called a mojo message is sent to the `ServiceWorkerVersion::ClaimClients()` method. This will use `ServiceWorkerVersion`'s associated `StorageKey` to find the associated `ServiceWorkerRegistration` by calling `ServiceWorkerContextCore::GetLiveRegistration()`. Finally the code invokes `ServiceWorkerRegistration::ClaimClients()` which iterates over all the `ServiceWorkerContainerHost` instance objects. This iteration will use the passed in `StorageKey` to filter on only clients in the same partition.

Clients.openWindow()

When the service worker `clients.openWindow()` API is called a mojo message is sent to the `ServiceWorkerVersion::OpenNewTab()` method. This will check the `ServiceWorkerVersion`'s associated `StorageKey` to determine if the service worker is partitioned. If it is, then `clients.openWindow()` will reject the operation with an `InvalidStateError`.

Metrics

Success metrics

This feature is targeted towards restricting functionality in 3rd party contexts and is not intended as a performance improvement. Therefore most of the success criteria will be based on functional testing and external reports of compatibility issues.

Regression metrics

Our main regression metric will be the
`PageLoad.Clients.ServiceWorker2.PaintTiming.NavigationToFirstContentfulPaint`
histogram. We will look at two parts of the histogram:

1. We will look for decreases in the total count in order to assess if there is a stability issue leading to fewer service worker controlled loads.
2. We will look at the median and P95 percentiles in order to determine if there is a loading performance regression.

In addition to this UMA histogram we will also monitor crbug reports for web compatibility problem reports.

Experiments

No planned experiments beyond a finch controlled rollout.

Rollout plan

Service worker partitioning should be rolled out simultaneously with storage partitioning. A site may get confused if only some APIs are isolated while others are not.

To that end the service worker rollout will be controlled via the `base::Feature` flag defined by the storage partitioning effort. It should roll out slowly in order to give developers time to file compatibility bug reports.

In addition, we should include a planned rollback of the feature while in the canary/dev phase of the rollout. The purpose of this will be to prove to ourselves that the feature can be safely turned down without corrupting user data, etc.

Finally, the rollout should include an enterprise policy to allow admins to disable the feature. After a number of milestones this policy will then be removed.

Core principle considerations

Speed

The main performance consideration for this effort is how service workers operate within the critical path of navigation. We must be able to quickly find if there is a service worker matching a particular URL. This design adds a new piece of information to this lookup process.

To maintain speed of navigation we index the `StorageKey` information in the leveldb database by including it in the `REG:` key. This will allow us to find matching service worker registrations in the same order of complexity as before.

There will be additional registrations stored in the database, but this can already occur today via the user visiting more origins. The quota system will clear partitions in much the same way as origins are cleared today helping to contain the size of the database. While there will be some increase, it should not be significant for lookup times since we are using a database index.

Security

In general the APIs considered here are all protected by the same-origin policy. This design does not weaken this policy. It further increases separation between different contexts by partitioning APIs within the same-origin when loaded under different top-level sites.

This helps improve the security posture of these APIs by preventing information leakages across host/embedder boundaries. For example, it will prevent history sniffing attacks using service workers controlling an embedded iframe.

In addition, this design also enables us to begin passing top-level site information into the network stack for fetches initiated by the service worker. This enables network level isolation and its associated security benefits.

Privacy considerations

Similar to the security section above, this design helps improve the privacy characteristics of the APIs being modified. Partitioning these APIs by the top-level site when used in 3rd party contexts will prevent state from being shared in order to track the user across different host pages.

Also, this design allows us to pass top-level site information into the network stack in order to gain privacy benefits from isolation in the http cache and other network stack level state.

The design does not fully support passing top-level origin to the network stack due to the service worker potentially servicing requests with multiple different values. This will have some impact on cookie blocking and other network hooks that can specify exact origins. While the design does not fully support this, we do support cases where these hooks match against the top-level site which is an incremental improvement over previous status quo where no top-level information was passed.

Testing plan

The implementation of this design should include web-platform tests to verify correctness. We plan to specify this behavior and therefore should build tests that can be run in other browsers.

We will use unit tests to cover any parts of the implementation that cannot be easily verified in wpt.

Followup work

After launch we will eventually need to clean up the enterprise policy and feature flag. In addition, there will be follow-up work to implement partitioning for other APIs; e.g. SharedWorker, BroadcastChannel, etc.