The Chrome Storage Service

This Document is Public

Authors: pwnall@chromium.org, rockot@google.com, dmurph@chromium.org

One-page overview

Summary

Chrome's Web Platform implementation for client-side storage features will be moved behind a Mojo service. The service will eventually live in a separate process on desktop versions of Chrome.

Platforms

Mac, Windows, Linux, Chrome OS, Android.

Team

storage-dev@chromium.org

Bug

https://crbug.com/994911

Code affected

//content and the code depending on it, //components/services/storage, //storage

---

Design

Background

Consequences of moving features individually

- Chrome Storage Service State Sharing - Web Platform API Dependencies
- Early thoughts on Storage Service and Service Worker

Overview

High-level code map.

- `//content/browser/` - Holds storage features that haven't been moved, and any logic needed to connect features to the rest of the browser. AppCache's navigation interceptor is an example of connecting logic.
- `//storage/` - Holds building blocks, for features. Examples: LevelDB scopes, our SQLite abstraction. Unrestricted visibility.
- `//components/services/storage/public/mojom` - Mojo interface exposing storage service functionality to the browser. Unrestricted visibility.
- `//components/services/storage/public/cpp` - Additional C++ interface exposing storage service functionality to the browser. Unrestricted visibility.
- `//components/services/storage/some_feature` - Storage feature implementation. Visibility restricted to `//components/services/storage`. Contains its own BUILD.gn that defines source_sets for the feature's code and tests.
- `//third_party/blink/public/mojom/some_feature` - Feature-specific interface between the Storage Service and Blink. Should be its own mojom_component. See dom_storage for an example.
- `//third_party/blink/renderer/modules/some_feature` - Blink-side implementation. This should have minimal changes.
- `StorageService` - interface between the storage service and the browser
- `RestrictedStorageService` - interface between the storage service and untrusted code (e.g. renderer) representing an origin

High-level, overly simplified strategy.

1. Map out Storage APIs into the following dependency graphs.
    - **Code dependencies.** For example, IndexedDB depends on the File API, because of its ability to store Blobs.
    - **API usage dependencies.** Tracked in this document.
2. For each feature without code dependencies in `//content/browser`, the feature will be moved to `//components/services/storage` and exposed via `//components/services/storage/public/mojom`. Repeat step 2 until all features are moved. Ordering should be informed by API usage dependencies.

Moving a feature will follow the steps below.

1. Remove any code in `//storage/common`. Post Onion Souping, all the renderer code for the feature should be in Blink. The removal will generally entail moving code to `//storage/browser` and to `//third_party/blink`.
2. Move the browser-side implementation from `//content/browser` and `//storage/browser` to `//components/services/storage/some_feature`. This includes creating a BUILD.gn with source sets for the feature's code and tests and modifying other BUILD.gn files, clarifying the dependencies between the feature and the rest of the browser code.
3. Move the Mojo interface between the browser process and the renderer process from `//third_party/blink/public/mojom` to `//storage/public/mojom`.

4. Vend the moved interface via the `RestrictedStorageService` interface, instead of serving it via `RendererInterfaceBinders`.
5. If necessary, create a Mojo interface between the storage service and the browser in //storage/public/mojom and transition any direct calls into feature code from `//content/browser` to the Mojo interface.

Basic Service Architecture

The service maintains a set of **partition** objects which are analogous to Content's `StoragePartitionImpl`. A partition fully and exclusively encompasses ownership of the contents of a persistent (in-filesystem) directory or an in-memory database.

The main service interface can be defined roughly as:

```
interface StorageService {
  BindPartition(mojo_base.mojom.FilePath? path,
                pending_receiver<Partition> receiver);
};
```

NOTE: Binding with a null `path` yields an isolated in-memory partition accessible exclusively to the client making that specific call. Only persistent partitions may be shared by multiple clients, by providing the exact same `path` value when calling `BindPartition`.

Once a `Partition` is bound, it can be used to bind new origin-scoped endpoints, or **origin contexts**:

```
interface Partition {
  BindOriginContext(
      url.mojom.Origin origin,
      pending_receiver<OriginContext> receiver);
};
```

Finally, an `OriginContext` is used to bind endpoints for specific API backends, *e.g.*:

```
interface OriginContext {
   BindDomStorageContext(
       pending_receiver<DomStorageContext> receiver);


   BindIndexedDbContext(...);
   // etc...
};
```

Threading and Scheduling

The storage service will attempt to use the modern scheduling infrastructure.

When running in a separate process, there will be no dedicated IO thread. Mojo interfaces will be bound on the main thread, which will not allow blocking I/O calls. Features that issue blocking I/O calls will use `base::PostTask` with the `base::MayBlock()` trait. Code embedding third-party libraries, like LevelDB, may still spawn dedicated I/O threads.

Browser Integration

The browser will own a single main `StorageService` pipe connected to the service, and each `StoragePartitionImpl` will own a corresponding `Partition` pipe. As subsystems are migrated into the service, implementation will migrate out of `StoragePartitionImpl` and/or its dependencies.

Restartability

We want Chrome to survive a Storage Service crash. While we can't make crashes non-observable, we can minimize the impact.

Our main reference point is the Network Service, which supports restartability. When the Network Service crashes, in-flight requests are failed. Existing tabs can fire new network requests, which will be correctly routed through the newly started Network Service. This looks like a transient network failure, so it's reasonable to expect applications to handle the errors. Given the Network Service's low crash rate, this approach and level of effort is sufficient.

DOMStorage maintains caches in renderers. Changes accumulate in these caches, and are written to disk every few seconds. If the Storage Service crashes, the changes that were not written to disk must be pushed from renderer-side caches to the restarted Storage Service. This doesn't need to be perfect, but we need to expose reasonable semantics to applications.

- Blob URLs will not persist across storage service crashes
- Blob URLs will be broken across storage service clients
- Quota will move separately from APIs by making QuotaClient a mojo interface
- Want to measure reads vs writes for every API, in case we figure out how to analyze that

Sandboxing

Running the Storage Service out-of-process but unsandboxed should still provide ample stability benefits such that sandboxing is not a blocker to ship. As such, the current plan is to first experiment with out-of-process Storage Service with sandboxing disabled.

A sandboxed Storage Service process will have no direct privileges to traverse the filesystem, and will instead need to rely on limited filesystem access through IPC to the browser process. A naive implementation of such a system may incur performance regressions, and so sandboxing the process warrants independent experimentation and analysis from the overall service launch effort.

The bug tracking development of sandboxing support is https://crbug.com/1052045.

Briefly, the sandboxed service process will use our tightest (AKA "utility") sandbox configuration in Chrome. Upon launch, Chrome will provide the service with a single Mojo interface which can be used to perform operations on the filesystem using *strict relative paths* -- non-absolute paths with no parent references, validated automatically by Mojo internals. The strict relative paths will be interpreted by the browser as relative to a fixed location on disk, namely the "user data dir" where all Chrome profiles are stored.

DOM Storage

The DOM Storage implementation has already been partially servicified, in the sense that its dependencies are reasonably well isolated into `//content/browser/dom_storage/`. The interesting dependencies still hanging around are `SpecialStoragePolicy` -- an interface used to support Chrome extensions removing browsing data during shutdown; and the File Service, which was part of an early effort to servicify browser storage.

SpecialStoragePolicy

It's not immediately clear what to do about this API. The object's only use for DOM Storage is to ask synchronous questions during shutdown in order to affect the behavior of `LocalStorageContextMojo`.
One possibility is to add a synchronous shutdown API to `Partition` (which we may need for other use cases anyway), which the browser uses to send a snapshot of the policy state it sees at the moment, and we invoke this on shutdown so the partition can propagate it to any relevant subsystems. We would like to avoid this if possible since (a) sync IPC on shutdown is risky, and (b) sync IPC in general is a sadness.

Alternatively we may just push configuration state changes down to the Storage service when they occur, and the service can make a best effort to adhere to last-known policy settings during shutdown.

File Service Removal

This service is only used to support DOM storage. It runs in the browser process with an instance per `BrowserContext`, and each instance is effectively associated with the corresponding `BrowserContext`'s directory path.

In order for the service to work properly, the browser injects (through global state) a mapping of Service Manager service "instance group" identity to file path. This is an unnecessary source of complexity given the limited use.

Part of the implementation work for DOM Storage servicification will thus involve deleting the File service. Each `Partition` instance in the Storage service will already have the same amount of information that the File service had, and the Storage service can access the file system directly. There is therefore no need for any subsystem's implementation to go through another service to reach persistent disk storage.

Migration Into Storage Service

The basic strategy for implementation will be to temporarily allow dependencies from `//content/browser` onto specific private details of `//components/services/storage` as code is migrated with its Content dependencies stripped away.

Once all relevant implementation is moved into `//components/services/storage`, private DOM Storage-specific dependencies will be disallowed in `//content/browser` and `StoragePartitionImpl` will broker access to the DOM Storage subsystem exclusively through its interface to the Storage service.

Security

The browser-side implementation of both local and session storage do synchronous checks of `ChildProcessSecurityPolicyImpl` every time certain origin-bound requests come in. Namely the calls to SessionStorageNamespace.OpenArea and StoragePartitionService.OpenLocalStorage both check the security policy of the render process ID associated with the implementation at interface binding time to ensure that the renderer should be allowed to access storage for the given origin.

In both cases, no subsequent checks are done for calls on the corresponding StorageArea once it's bound, and StorageArea endpoints only appear to be purged once the client (renderer) disconnects from them. As such, it seems the current implementation does not attempt to protect against e.g. an Evil™ renderer retaining a prior origin's `StorageArea` pipe after a navigation away from that origin. Presumably we are OK with this.

In any case, we will leave these checks in the browser, so all `StorageArea` binding requests from renderers must continue to be brokered through the browser. The Storage service will likely implement slightly different interfaces from `SessionStorageNamespace` and `StoragePartitionService` given the definition of `OriginContext` as a scoping layer.

The service will most likely implement Blink's `StorageArea` interface as it is currently defined.

IndexedDB

See Storage Service IndexedDB Tracking Document for the IndexedDB work planning & progress.

IndexedDB is mostly isolated in the browser. The main complex changes are going to be:

1. Content policy checks (using ChildProcessSecurityPolicy), similar to DOMStorage
2. Interactions with the Blob Storage system
    a. Copying blobs to a files on transaction commit
    b. Creating new blobs for 'get' results

Quota

The quota system has three main integration points.

- QuotaClient is the interface to quota-managed storage features. Each storage feature has an implementation of `QuotaClient`, and a `QuotaManager` owns one instance of each implementation.
- QuotaManager is the interface to quota users. It is used by sites and browser features to query the current quota status, and by quota-managed storage features to report changes in quota usage.
- QuotaDispatcherHost is the main interface between the browser process and renderer processes.

The quota system will be moved to the Storage Service. `QuotaManager` will become a Mojo interface, and will be used to communicate between the browser process and the storage service process. QuotaManagerProxy will be removed, as we can rely on Mojo to do any thread hopping as needed.

`QuotaClient` will become a Mojo interface, so we can have storage features that live outside the storage service. This is a pragmatic decision, as it allows us to move quota-managed storage features into the Storage Service one by one. In a world where all the storage features are migrated, we may turn `QuotaClient` back into a C++ abstract base class.

`QuotaDispatcherHost` will be converted to an origin-scoped `QuotaHost` interface. Renderers will obtain instances from the OriginContext interface.

Service Worker

The Service Worker integration with the Storage Service is described in this document.

Metrics

Success metrics

Landing the refactorings proposed here is a code health improvement, and a success in itself.

When running with an out-of-process Storage Service, the browser process crash rate should be reduced.

Regression metrics

The code refactoring needed for the in-process Storage Service will rely on the perf bot infrastructure to catch performance regressions. The out-of-process Storage Service will be rolled out via Finch, and we'll check for no significant regressions in speed launch metrics as well as various performance metrics specific to any affected storage subsystems.

For specific subsystems there are two potential sources of regression: increased latency from additional IPC hops, which should be minimal; and increased latency from sandboxed file I/O. For some web APIs, such regressions could have user-visible impact such as jank, slower page loads, or diminished storage consistency.

DOM Storage Metrics

`LocalStorage.MojoTimeToPrime` measures how long it takes for a renderer to synchronously populate its Local Storage cache from the service's backend. Because this process involves substantial IPC and file I/O from within the service, the metric serves as a good indicator of end-to-end, user-visible DOM Storage performance.

TODO: Add something tracking commit failures. Should this be tracked at the service level?

IndexedDB Metrics

TODO

Service Worker Metrics

TODO

`PageLoad.Clients.ServiceWorker2.PaintTiming.NavigationToFirstContentfulPaint` tracks the First Contentful Paint (FCP) of pages that are controlled by service workers. This is a primary metric for service worker performance analysis. Service worker controlled navigations require storage access to retrieve registrations.
`ServiceWorker.StartWorker.Time` measures time to start a service worker. Starting a service worker needs registration information and may access storage (if these aren't on in-memory cache). Service worker startup is a critical path for cold navigation to pages that are controlled by service workers.

TODO: LevelDB write errors, open failures.

Quota Metrics

TODO
`Quota.EvictedOriginsPerHour`, **`Quota.EvictionRoundsPerHour`** - Ensure that evictions run at the same rate, without any performance problems.

`Quota.TimeDeltaOfEvictionRounds` - Ensure that evictions run at the same speed, without any performance problems.

`Quota.UsageByOrigin` - Ensure that storage usage doesn't drop.

TODO: Add metrics for I/O errors, database open time.

Cache Storage Metrics

TODO
Make sure to include metrics about performance, open / warmup time.

WebLocks Metrics

TODO

Rollout plan

The in-process Storage Service will be rolled out via waterfall, as it is essentially an incremental refactoring of the existing system.

The out-of-process Storage Service will ship on desktop only. It warrants going through the full Chrome feature launch process and will be rolled out via a Finch experiment.

Two separate feature flags have been introduced to Chrome already: **`StorageServiceOutOfProcess`** and **`StorageServiceSandbox`**. The latter only has meaning when the former is enabled, and both are disabled by default.

Testing

Extra `browser_tests` and `content_browsertests` steps will be initially added to FYI bots with both unsandboxed and sandboxed out-of-process Storage Service enabled.

Unsandboxed out-of-process Storage Service is expected to be continuously functioning and stable ASAP, so it will be moved to the main waterfall once the FYI tests remain green for a while.

On the other hand, the additional challenges of sandboxing mean that sandboxed tests are expected to break frequently during ongoing feature development. Once all high-priority features are ported to the service and testing has stabilized, the sandboxed tests can also move to the main waterfall.

Finch Experiment

The Finch experiment for out-of-process Storage Service will initially define a single experiment group that enables the unsandboxed service process. Per the best practices guidelines, the proposed rollout will use an experiment group population of 1% on Stable and 25% on Canary, Dev, and Beta.

The experiment group is expected to see a reduction in browser crash rates, and the primary purpose of the experiment is to analyze performance impact across the various affected storage subsystems.

Once the service's feature set has stabilized and there is sufficient waterfall coverage of the sandboxed service, the experiment will be reset with *two* experiment groups: one for the unsandboxed service process (same behavior as the initial experiment group) and one for the sandboxed process. Both groups will use the same population numbers as above (1% Stable, 25% C/D/B).

Once enabled, the new sandboxed experiment group is expected to see a similar reduction in browser crash rates, and again the primary purpose of introducing this group is to analyze relative performance among all three scenarios: in-process, out-of-process unsandboxed, and out-of-process sandboxed.

Performance will be analyzed primarily through the collection of new or existing UMA metrics as described in the Regression Metrics section above.

Launch

Once metrics appear satisfactory for the unsandboxed Storage Service and all other launch requirements are met, out-of-process mode will be switched on by default for all desktop platforms. At this point, separate waterfall steps can be removed for the unsandboxed version, as it will have full CQ coverage.

A similar process will be applied to launch the sandboxed service once all features are properly adapted to a sandbox environment and any outstanding performance regressions are addressed.

Core principle considerations

Speed

The in-process Storage Service is not expected to have a significantly different performance profile from today's code.

The out-of-process Storage Service will introduce extra process hops in establishing Mojo connections from the renderer processes to the implementation of Web Platform storage features. We don't expect this to result in a material regression on the speed launch metrics.

Most importantly, once the initial Mojo connection is established, the renderer-side implementation of storage features will talk directly to the Storage Service process. This results in the same number of IPC messages as the current architecture, where storage feature backends live in the browser process.

The concern for IPC overhead ruled out an alternative design where the Storage Service would expose low-level primitives such as a File service, a LevelDB service, and a SQLite service.

Security

The in-process Storage Service model does not bring any meaningful security changes.

In the out-of-process Storage Service, the browser process will start the Storage Service and connect to it, then broker all requests from renderers to Storage interfaces. This enforces Site Isolation. Other processes (network, GPU, utility) should not need to connect to the Storage Service.

The out-of-process Storage Service has the security benefit of moving the implementation of some complex features, like IndexedDB, outside the browser process. A successfully exploited vulnerability in storage code will not immediately translate into full browser process control. The browser will not trust the Storage Service process any more than it trusts other sandboxed service processes. Any requests for resources or operations performed on the service's behalf must be validated appropriately by the browser.

At the same time, security bugs in storage code will still have High impact, because taking control of the Storage Service process allows the attacker access to user data from other origins, such as `google.com`.

Chrome would have better resilience to storage code vulnerabilities if the bulk of the implementation of Web Platform features would move to Blink, and run inside the renderer process. A vulnerability in Blink code would only expose the data stored by the origins that the exploited renderer process can access. In particular, on platforms where full Site Isolation is deployed, an origin would not be able to access other origins' data. While this project does not aim to move code to the renderer, the work here is a step forward in that process, because it entails untangling storage features from the rest of `//content/browser`.

Privacy considerations

None. No functional changes are planned.

Testing plan

Chrome's Web Platform features have extensive integration tests, in the form of browser tests and Blink web-tests. We will create a separate suite that runs these tests with an out-of-process Storage Service.

Followup work

The current plan does not require any follow-up work, such as code cleanup.

This project does not aim to move storage feature code to the renderer process, as discussed in the section on security considerations. Moving code to the renderer is a great potential follow-up project.

Related Documents

Chrome Storage Service State Sharing - Web Platform API Dependencies