Partitioning the File System API

mek@chromium.org / Draft: 2021-06-17

Overview

This doc describes integrating the File System API with both storage partitioning and storage buckets. Most of the code for the file system API lives in `//storage/browser/file_system`, although some of the relevant entry points are in `//content/browser/file_system` and `//content/browser/file_system_access`.

Background/related documents

This design doc explains one particular aspect of the larger work needed to partition client side storage APIs. More background information can be found in the following explainers and related design documents:
https://github.com/wanderview/quota-storage-partitioning/blob/main/explainer.md
Storage Partitioning Design
Storage Key Design

Storage Partitioning

https://crbug.com/1221308

FileSystemURL changes

This mostly comes down to migrating from Origin to Storage Key, although it is a bit more complicated then other APIs due the File System APIs extensive dependencies on URLs as

internal identifiers for files. For most file system URLs (and `FileSystemURL` instances) the origin of the URL doesn't matter, as most of these URLs represent files or file systems that aren't tied to any particular origin. I don't think we want to (or need to) change the serialization format of file system URLs (i.e. those will keep only containing an origin), but we will need to change from `url::Origin` to `blink::StorageKey` in `storage::FileSystemURL`. In order to accomplish this, we will need to add a storage key to the methods that are used to create `FileSystemURL` instances. Specifically `FileSystemContext::CrackURL` will need to get a `StorageKey` parameter in addition to the existing `GURL` parameter, and `FileSystemContext::CreateCrackedFileSystemURL` will change its `url::Origin` parameter to a `StorageKey` parameter.

With that `FileSystemURL` can then be updated to have a storage key getter in addition to the existing origin getter, after which usage of the current origin getter can be modified to call the storage key getter instead. `FileSystemURL` should verify/DCHECK that the passed in storage key "matches" the origin that is parsed from the `GURL`.

Sandboxed FS changes

The majority of origin usage can remain as just using the origin of the storage key (in particular the code in `//chrome/browser/sync_file_system` is chrome apps only, deprecated, and does not need to care about non-first-party storage keys). The places that will need to be changed to use the full storage key are all related to the sandboxed file system (generally files called either `obfuscated_file*` or `sandboxed_file*`).

Besides changes to pass in the correct Storage Key when parsing URLs, `FileSystemContext::OpenFileSystem` (the method that is used to open a sandboxed file system) will also need to be modified to accept a `StorageKey` rather than an `Origin.

Sandboxed FS entry points

Besides the `filesystem://` URL handling described below, the other two entry points that need to make sure they end up using the correct Storage Key are the legacy file system API (in `content/browser/file_system/file_system_manager_impl.cc`) and the File System Access API (in `//content/browser/file_system_access`).

For the legacy file system API this is somewhat complicated, as this code currently relies on the renderer to pass in the correct origin to use, and we don't currently track what frame a request came from. Ideally we'd know what frame a request came from, and use that to pass the correct `StorageKey` to the `OpenFileSystem` and `CrackURL` methods throughout this code. We would do this by adding a `StorageKey` as context to `mojo::ReceiverSet<blink::mojom::FileSystemManager> receivers_`. That means adding a `StorageKey` parameter to `FileSystemManagerImpl::BindReceiver`. For requests initiated by a frame (i.e. `RenderFrameHostImpl::GetFileSystemManager`) it is straightforward to pass in the right storage key, but to support PPAPI this mojo interface is also exposed at the

process level, and there isn't really a good way to figure out the right storage key there (see also https://crbug.com/873661).

For the File System Access API things should be somewhat simpler. The first place that needs to be updated is the `OpenFileSystem` call in `FileSystemAccessManagerImpl::GetSandboxedFileSystem`. Here we will need to get the `StorageKey` from the frame that tried to open the sandboxed file system. To make this possible we should change `FileSystemAccessEntryFactory::BindingContext` from storing a `url::Origin` to storing a `blink::StorageKey`. The other place that will need to change is the `FileSystemAccessManagerImpl::DeserializeHandle` method (which is called by IndexedDB when loading a handle from the database). Here too we'll need to change the origin to a storage key and use that to create the cracked `FileSystemURL`.

filesystem:// URL handling

Adding a StorageKey parameter to the CrackURL method (and making sure the correct value is passed in) should be enough to make sure that loading of `filesystem://` URLs works correctly in non-first party contexts. However the plumbing to get the correct storage key (in the code in `content/browser/file_system/file_system_url_loader_factory.cc`) is probably going to be different from the other entry points.

Sandboxed File System file locations

Initially just leave alone, i.e. base location only on origin rather than full storage key. Once the Sandboxed File System has also been integrated with Storage Buckets we'll end up using storage buckets infra for what today `SandboxOriginDatabase` is used. As such there is not much point in modifying `SandboxOriginDatabase` to support non-first-party storage keys.

Storage Buckets

TODO

- add GetSandboxedFileSystem method to StorageBucketHost mojo interface matchin the one in FileSystemAccessManager (eventually probably remove it from the FileSystemAccessManager interface).
- Add bucket id/name to OpenFileSystem methods.
- Something with the code in ObfuscatedFileUtil::GetDirectoryForOrigin, and its callers. Either plumb bucket ID (or name) through FileSystemURL, or perhaps "better" change how the FSA API integrates with sandboxed FS? Not sure.. Also SandboxFileSystemBackendDelegate::GetBaseDirectoryForOriginAndType.

Document History

Date	Author	Description
2021-06-17	mek@	Initial draft, primarily talking about Storage Partitioning