File handling API for web apps in Chrome (tinyurl.com/file-handling-design)

This Document is Public

Authors: huangdarwin@chromium.org (current), robertwoods@google.com (original/previous).

Last updated: 2021-03-25

Objective

The goal for this project is to implement an API that will allow Chrome progressive web apps (PWAs) to 'handle' (read) files in the host OS's file system, in much the same way as a native app would. This dovetails with the larger objective of increasing transparency between web apps and native apps, enabling a more consistent user experience across both per the Progressive Web App (PWA) paradigm.

Background

A **file handler** represents a web application's ability to handle a file with one of a given set of MIME types and/or file extensions. An image editor, for example, might support displaying several distinct image formats. When this editor is installed it should be possible to open image files in one of these formats from within the file browser of a user's operating system. A web app can register its ability to handle one of these file types in its manifest.

The file handling API will be implemented against the backdrop of a larger project to move desktop PWAs out of Extensions in Chrome (internal), and into their own system. Initially, both PWAs and Shortcuts (a mechanism for creating an OS-specific shortcut to launch a website in a Chrome tab), referred to collectively as 'Bookmark Apps', were implemented on top of the Extensions stack. This allowed us to leverage the existing Extensions infrastructure to furnish the key requirements for PWAs (internal); namely, that they be persistent, installable, updateable, launchable and syncable. But as time has passed, the need to separate Bookmark Apps and

Extensions has become more apparent, due to a lack of clarity around code ownership and technical debt, among other issues. Since this work is ongoing, the APIs for Bookmark Apps under the old system (in the `extensions` namespace) exist in parallel with those for Web Apps under the new system (in `web_app`). Part of this project will involve facilitating the transition from one system to the other, by implementing the API in both systems.

Overview

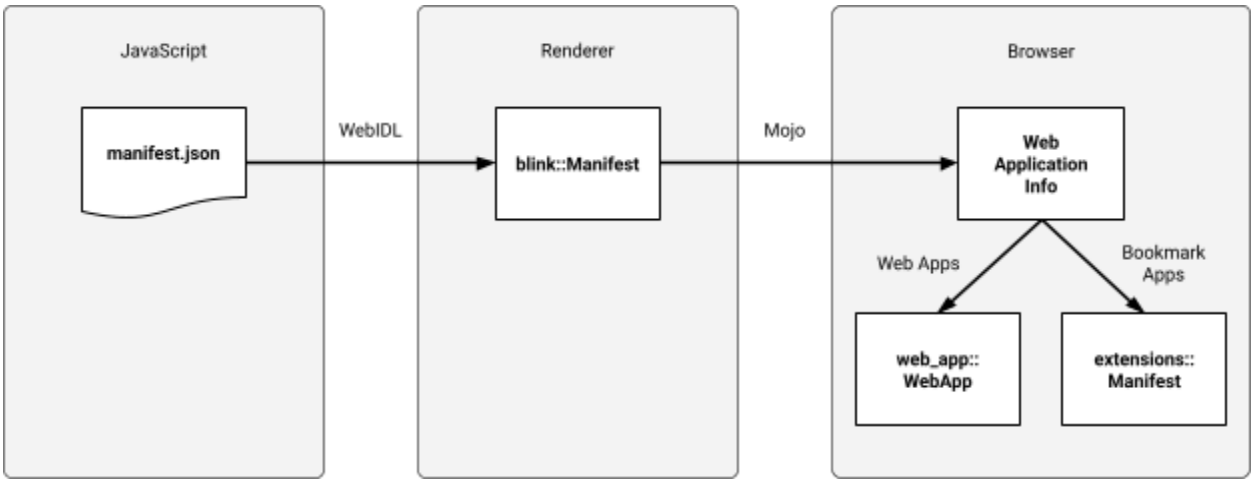
The implementation of the file handling APIs in Chrome will touch a number of systems. First, the handlers must be parsed from the web app's JSON manifest and registered with the browser's internal representation of the app. These representations must then be stored persistently (e.g. in LevelDB). APIs must be created for identifying which file handlers are available at any given point in time, and for retrieving them. Enabled file handlers must also be registered on the host OS, so that files of the given type are associated with the web app. (This implies the need for OS-specific implementations for common operations.) Finally, functionality must be put in place for launching a web app with files (and dispatching on the appropriate endpoint within the app), and for sending launched files to a running web app.

All of this must be done alongside existing file handling implementations in Extensions, without breaking the legacy Chrome Apps file handling APIs.

Note as well that this design document will be extended to also provide File Handling icon support. (TODO: Update this design document to incorporate icon support)

Getting file handlers from a web application manifest

The app's JSON manifest is parsed into a `blink::Manifest` by `blink::ManifestParser`. This new object contains Blink's representation of the list of file handlers included in the original manifest, and is transferred to the browser using Mojo.



In the browser, the `blink::Manifest` is converted into a `WebApplicationInfo` by `web_app::UpdateWebAppInfoFromManifest`. This is the representation of a web app's metadata used by the browser when installing it as a PWA.

Saving file handlers for an installed app

Each of the file handlers in the `WebApplicationInfo` is converted into an `apps::FileHandler` object. Previously, file handlers were represented in the `apps::FileHandlerInfo` format used by Chrome Apps. (The new format is in the `apps` namespace so that it is available to both Web Apps and Bookmark Apps, as Chrome dependencies are not permitted under `extensions`. It may be beneficial to move all systems onto a shared representation in future.)

```
namespace apps {

struct FileHandler {
  ...
  struct AcceptEntry {
    std::string mime_type;
    base::flat_set<std::string> file_extensions;
  };

  GURL action;
  std::vector<AcceptEntry> accept;
};

}  // namespace apps
```

```
namespace apps {

struct FileHandlerInfo {
  ...
  std::string id;
  std::set<std::string> extensions;
  std::set<std::string> types;
  ...
};

}  // namespace apps
```

The new `apps::FileHandler`format was intended to closely mirror the proposed file handling API explainer, while preserving an explicit mapping between MIME types and file extensions consistent with the web standards emphasis on the former. This has the added benefit of

making Linux-specific implementations of file handling possible, since this mapping is needed for registering new MIME types with the OS and cannot be recovered from the old format.

For Bookmark Apps, the conversion to the new format happens in `extensions::ConvertWebAppToExtension`. The `apps::FileHandler`s are serialized on the `web_app_file_handlers` key of the `extensions::Manifest`, along with other app metadata. The legacy `apps::FileHandlerInfo` format is also still serialized under the `file_handlers` key, so as not to disrupt any possible existing functionality. (TODO(crbug.com/1173256): Now that BMO has landed, the `apps::FileHandlers` serialization and parsing code can be removed.)

For Web Apps, the conversion is performed by `web_app::WebAppInstallFinalizer::SetWebAppFileHandlers`. We have created code for saving web apps to and loading them from the `web_app::WebAppDatabase`.

Functionality has also been implemented for migrating the file handlers from a Bookmark App to a Web App when BMO lands. After an extensive refactor of file handling code, this amounts to a simple copy, since both platforms now share the same representation.

Finding out which file handlers are available

Again, as Bookmark Apps and Web Apps work differently, there are multiple code paths involved. `web_app::FileHandlerManager` exposes common methods for working with file handlers, such as retrieving available file handlers and enabling or disabling them. It has a single abstract method, `GetAllFileHandlers`, which is implemented by a system-specific subclass, allowing as much code to be shared between the two systems as possible. This method is implemented by `extensions::BookmarkAppFileHandlerManager` and `web_app::WebAppFileHandlerManager`.

Registering file handlers

There are two parts to registering file handlers, the second of which is not applicable on Chrome OS due to its tighter integration with Chrome.

1. Store a bit saying file handlers are enabled. This lets us determine whether to return everything or nullptr in `FileHandlerManager::GetEnabledFileHandlers`, which let's us determine what file handlers to attach to ShortcutInfos when creating shortcuts (necessary on OSX and Linux), and, on Chrome OS, to determine what file handlers are currently available.
2. Update file associations in the operating system. This is done by web_app_file_handler_registration, which has platform-specific implementations.

Unregistering file handlers

Unregistering file handlers is done in `FileHandlerManager`::`DisableAndUnregisterOsFileHandlers`.

Notably, on Windows, file type associations cannot be unregistered between a Web App and a file type, but a web app can be disassociated as a valid file handler. Therefore, on Windows, a web app with at least one file handler registered, will have all past file handlers registered on Windows. (This could potentially be fixed in crbug.com/1205519)

Chrome OS-specific code

On Chrome OS, some bespoke code was required to enable the Chrome OS Files App to talk to FileHandlerManager. This code is in web_file_tasks, which provides methods for finding handlers for files (`FindWebTasks`) and launching a handler for a file (`ExecuteWebTask`). This follows the pattern set by Arc, Crostini and Chrome Apps file handlers. It is completely agnostic to the current web apps system.

Linux-specific code

In freedesktop.org-compliant Linux desktop environments, application-to-filetype associations are created by specifying a set of MIME types supported by the app in its `.desktop` file. This file is then used to register the app among those that users can run in the desktop environment. The Linux-specific implementation of the file handling API includes code to achieve this. In some cases, the MIME types and file extensions associated with a particular app might not exist in the OS, and so we need a way of registering new mappings. This is done by calling out to `xdg-mime`.

Use of MIME Types vs file extensions

All platforms use File Extensions to register file type associations, except Linux, where mime types are used. It's possible that MacOS also intends to use mime types (bug).

Operating System	Mime Type	File Extension
Windows	No	Yes
ChromeOS	No	Yes
MacOS	No (bug)	Yes
Linux	Yes	No

Launching a web app with files

A web app may open a file using a PWA by selecting the PWA in the OS file manager's "Open

With" menu, or by simply double-clicking if the PWA is the default/only opener for the file type.

There are currently two different code paths for launching a PWA, one in the extensions system (`OpenEnabledApplication`), and one in the Web Apps system (`WebAppLaunchManager::OpenApplication`). The file handling parts of these functions are largely identical and involve two steps:

1. Identifying the correct launch URL (`FileHandlerManager::GetMatchingFileHandlerURL`). This is done by identifying the file handler that best matches the file extensions or MIME types of the launch files, then returning the `action` URL for that handler. At the moment, we only match by file extension, and MIME types are only used for matching on Linux.
2. Passing the files to the launched application (`WebLaunchFilesHelper::SetLaunchPaths`).

Sending launched files to a web app

From the browser process, launched files can be sent to a running web app using `WebLaunchFilesHelper::SetLaunchPaths`. This creates the mojom file handles and passes them to the renderer in `WebLaunchServiceImpl::SetLaunchFiles`, which calls `DOMWindowLaunchQueue::UpdateLaunchFiles` to enqueue the launch files to window.launchQueue. These launch files can be read by setting a consumer on the launchQueue in JavaScript:

```
window.launchQueue.setConsumer(launchFiles => {
  // Do something with launchFiles...
});
```

Manifest Update

PWAs may choose to update file handlers available on their site by updating their manifest. When this happens, file handlers will update similarly to PWA update surfaces, by detecting the update as the PWA is navigated to in `HaveFileHandlersChanged()`, and updating the registered file handlers as the PWA closes in `OsIntegrationManager::UpdateFileHandlers()`, by uninstalling all file handlers and reinstalling all file handlers. More context on manifest update is available in this design document (internal).

On manifest update, the file handling permission state, if currently in "ALLOW", will be set back to "ASK", to avoid a possibility of a PWA requesting more dangerous file types after the permission was previously granted for less dangerous file types. If the permission state was previously in "ASK" or "BLOCK", the permission state will not change.

Permissions

To ensure user trust and the safety of users' files when file handling is used to open a file, a permission prompt will be shown before a PWA can view a file. This permission prompt will be shown right after:

1. The user selects the PWA to open a file, so that the permission is tightly coupled to the action of opening a file using the PWA, making it more understandable and relevant.
2. The site loads without the file, so that the user has an expectation of what the PWA is and why it would like to view the file.

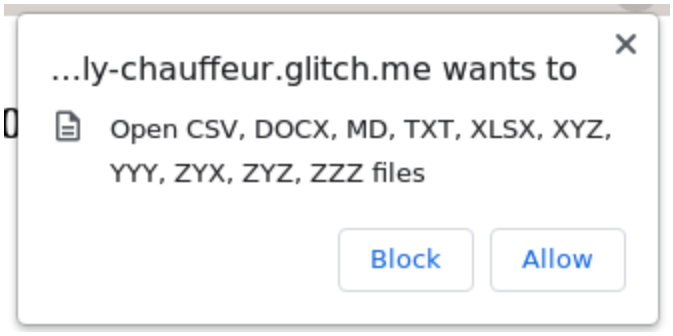
This permission will show every time until the user clicks to "Allow" or "Block" file handling for the site, or ignores the prompt three times (after which Chrome will embargo and BLOCK this permission), after which the selected setting will persist across the PWA closing and reopening. After manifest updates change file handlers though, permissions will notably be reset per the "manifest update" section above.

This permission is implemented as `ContentSettingsType::FILE_HANDLING` (CL), and will apply across all file types the PWA handles, as it is difficult for users to comprehend and make an actionable decision based on file extensions. As spoofing is a major concern (ex. evil.com masquerading as taxsoftware.com, and trying to read your tax software), the permission will clearly display the origin, as well as the site in the content area underneath the prompt, and the site title in the PWA top bar, so that users can have a clear understanding of the site attempting to view the file.

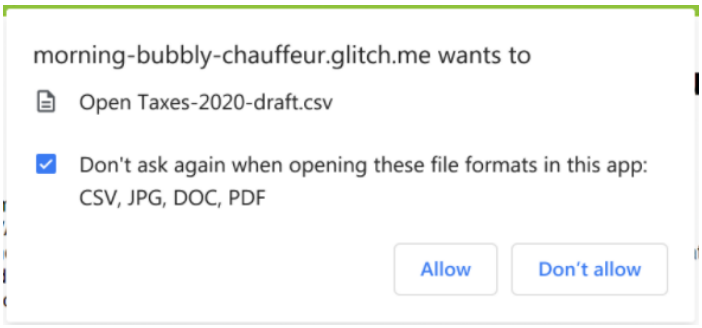
When a permission is moved from "ALLOW" or "ASK" to "BLOCK", file handlers will be unregistered from the underlying operating system (CL). Similarly, when a permission is moved from "BLOCK" to "ALLOW" or "ASK", file handlers will be re-registered in the system.

Notably, file handlers will be registered when the PWA is first installed, without any prompt specific to file handling (as part of the "install" prompt/flow). This is because installation of the PWA is considered to be when all OS integrations are registered into the operating system. From there, the permission here is to guard the user from any information leakage or improper use of files.

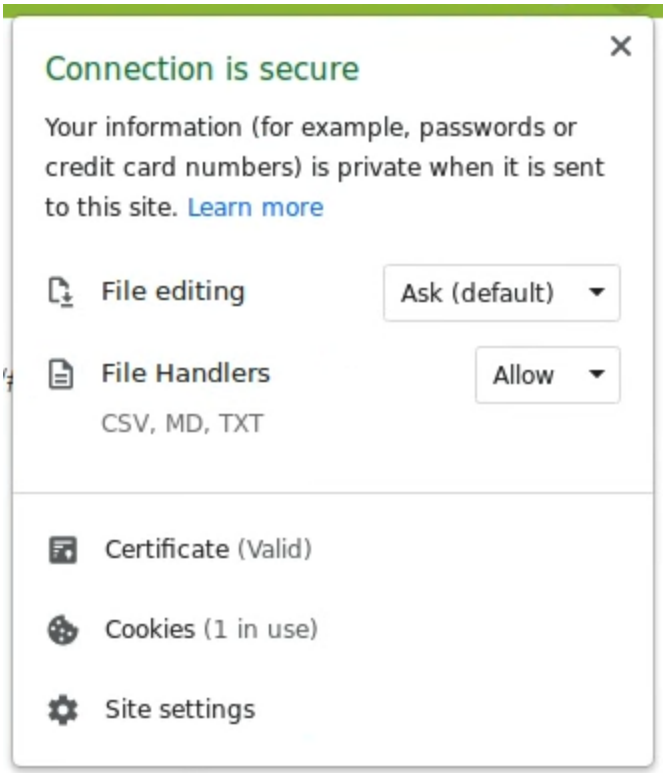
Another related permission that interacts with file handling is the File System Access (formerly Native File System) API, which can ask for write permission in a separate prompt, if the site would later like to write to the file opened via file handling. A prompt will show up every time for write permission, regardless of the file handling permission status.



Permission prompt UX mock for M92 (link)



Permission prompt UX mock for M93 (link)



Content setting UX mock (link)

See links in privacy and security considerations for more information and many alternative permission models considered.

What do we still have to do?

Dispatch on MIME type

At present, `FileHandlerManager::GetMatchingFileHandlerURL` dispatches to a handler's action URL on the basis of a file's extension. Functionality to dispatch on MIME type as well (in the absence of a file extension, or perhaps to be preferred over it) needs to be added in here.

Linux-specific code

Although new MIME type will be registered on installation, there is currently no cleanup done on uninstall. This means that any custom MIME types remain on the system once the app has been removed. Removing them will involve deleting the corresponding shared MIME info XML file wherever it has been installed (e.g. `~/.local/share/mime/packages`), and then calling another Linux shell command on the MIME directory (`update-mime-database ~/.local/share/mime`).

Mitigations for not being able to dissociate file type associations from web apps on Windows

Per the unregistering file handlers section, Chrome cannot dissociate file type associations from web apps. Therefore, we should account for unregistered file handlers both on permission surfaces (bug), and ensure that launch events can only be received for file types the web app still has registered as file handlers in the provided manifest (bug).

Project information

Chrome OS: mgiuca, alancutter, etc.
Linux: huangdarwin (previously harrisjay and robertwoods).
Windows: davidbienvenu.
macOS: ccameron.

Relevant high-level bugs include 829689, 1028448 and 938103.

Bug trackers: crbug for code and GitHub issues for the WICG standards proposal. See also handover notes (internal) and minutes from an exploration (internal) for more information.

Privacy and security considerations

Registering a file association requires a degree of trust in the application the file is associated with: that application will be able to see the contents of and potentially make modifications to any files that it is opened with from the file manager, which poses privacy and security concerns. This problem is more pronounced when an application is made the default handler for certain types of files, as the user may not realise which application will be opening the file. In some cases, an app may become the default handler without the user's intervention, such as when it is the only installed app capable of handling a particular file type.

For in-depth information on further security and privacy concerns, as well as mitigations implemented or considered, please see the File Handling Security Model.

Other mitigations include a limit of 10 file handlers per PWA (CL), only providing read access on open, and not allowing access to directories.

For in-depth information on the permission model, see the permission model (internal) and UX mocks (internal).

Launch plans

Origin trials

There is an existing system for handling origin trials in Chromium. However, the file handling APIs are somewhat different to other APIs that go through the origin trial process, as enabling and disabling the trial results in some state being changed in the operating system (registering the file handlers).

This means that the origin trial requires some special architecture: Each time a web app is visited, we check if it has a valid origin trial token, and, if so, we register the file handlers, and store the expiry time of the token. If the token is not valid, we unregister the file handlers. On Chrome startup, we also unregister file handlers for all app's where their origin trial token has expired.

The `FileHandlingExpiry` service handles determining when the origin trial expires.

`FileHandlerManager::CleanupAfterOriginTrials` handles unregistering app's whose file handlers have expired on Chrome startup.

`FileHandlerManager::UpdateFileHandlingOriginTrialExpiry` handles updating when an app's origin trial is going to expire (and registering/unregistering it's handlers if the trial validity changed)

References (internal):

- Previous (now-deprecated) version of this document
- Getting started on File Handling
- UX Mocks
- Testing and Validation
- Manual Test Cases