The origin private file system

- Browser support
- Motivation
  - Traditional way of working with files on the web
  - Modern way of working with files on the web
  - Restrictions of working with files on the web
  - Files as the foundation of processing
- The user-visible versus the origin private file system
- Specifics of the origin private file system
- Getting access to the root directory
- Main thread or Web Worker
- Using the origin private file system on the main thread
  - Creating new files and folders
  - Accessing existing files and folders
  - Getting the file associated with a file handle for reading
  - Writing to a file by streaming
  - Deleting files and folders
  - Moving and renaming files and folders
  - Resolving the path of a file or folder
  - Checking if two file or folder handles point to the same file or folder
- Listing the contents of a folder
- Recursively listing the contents of a folder and all subfolders
- Using the origin private file system in a Web Worker
  - Getting a synchronous access handle
  - Synchronous in-place file methods
- Copying a file from the origin private file system to the user-visible file system
- Debugging the origin private file system
- Demo
- Conclusions

- Related links
- Acknowledgements

The origin private file system allows web apps to store and manipulate files in their very own origin-specific virtual filesystem, including low-level file manipulation, byte-by-byte access, and file streaming. The origin private file system is supported across all major browsers.

Browser support

The origin private file system is supported by modern browsers and is standardized by the Web Hypertext Application Technology Working Group (WHATWG) in the File System Living Standard.

Motivation

When you think of files on your computer, you probably think about a file hierarchy: files organized in folders that you can explore with your operating system's file explorer. For example, on Windows, for a user called Tom, their To Do list might live in `C:\Users\Tom\Documents\ToDo.txt`. In this example, `ToDo.txt` is the file name, and `Users`, `Tom`, and `Documents` are folder names. `C:\` on Windows represents the root directory of the drive.

In this article, I use the terms *folder* and *directory* interchangeably, disregarding the difference between the file system concept (directory) and the graphical user interface metaphor (folder).

Traditional way of working with files on the web

To edit the To Do list in a web application, this is the traditional flow:

1. The user *uploads* the file to a server or *opens* it on the client with `<input type="file">`.
2. The user makes their changes, and then *downloads* the resulting file with an injected `<a download="ToDo.txt>` that you programmatically `click()` via JavaScript.
3. For opening folders, you use a special attribute in `<input type="file" webkitdirectory>`, which, despite its proprietary name, has practically universal browser support.

Modern way of working with files on the web

This flow is not representative of how users think of editing files, and means users end up with downloaded *copies* of their input files. Therefore, the File System Access API introduced three picker methods—`showOpenFilePicker()`, `showSaveFilePicker()`, and `showDirectoryPicker()`—that do exactly what their name suggests. They enable a flow as follows:

1. Open `ToDo.txt` with `showOpenFilePicker()`, and get a `FileSystemFileHandle` object.
2. From the `FileSystemFileHandle` object, get a `File` by calling the file handle's `getFile()` method.
3. Modify the file, then call `requestPermission({mode: 'readwrite'})` on the handle.
4. If the user accepts the permission request, save the changes back to the original file.
5. Alternatively, call `showSaveFilePicker()` and let the user pick a new file. (If the user picks a previously opened file, its contents will be overwritten.) For repeat saves, you can keep the file handle around, so you don't have to show the file save dialog again.

Restrictions of working with files on the web

Files and folders that are accessible via these methods live in what can be called the *user-visible* file system. Files saved from the web, and executable files specifically, are marked with the mark of the web, so there's an additional warning the operating system can show before a potentially dangerous file gets executed. As an additional security feature, files obtained from the web are also protected by Safe Browsing, which, for the sake of simplicity and in the context of this article, you can think of as a cloud-based virus scan. When you write data to a file using the File System Access API, writes are not in-place, but use a temporary file. The file itself is not modified unless it passes all these security checks. As you can imagine, this work makes file operations relatively slow, despite improvements applied where possible, for example, on macOS. Still every `write()` call is self-contained, so under the hood it opens the file, seeks to the given offset, and finally writes data.

Files as the foundation of processing

At the same time, files are an excellent way to record data. For example, SQLite stores entire databases in a single file. Another example are mipmaps used in image processing. Mipmaps are pre-calculated, optimized sequences of images, each of which is a progressively lower resolution

representation of the previous, which makes many operations like zooming faster. So how can web applications get the benefits of files, but without the performance costs of traditional web-based file processing? The answer is the *origin private file system*.

The user-visible versus the origin private file system

Unlike the user-visible file system browsed via the operating system's file explorer, with files and folders you can read, write, move, and rename, the origin private file system is not meant to be seen by users. Files and folders in the origin private file system, as the name suggests, are private, and more concretely, private to the origin of a site. Discover the origin of a page by typing `location.origin` in the DevTools Console. For example, the origin of the page `https://developer.chrome.com/articles/` is `https://developer.chrome.com` (that is, the part `/articles` is *not* part of the origin). You can read more about the theory of origins in Understanding "same-site" and "same-origin". All pages that share the same origin can see the same origin private file system data, so `https://developer.chrome.com/docs/extensions/mv3/getstarted/extensions-101/` can see the same details as the previous example. Each origin has its own independent origin private file system, which means the origin private file system of `https://developer.chrome.com` is completely distinct from the one of, say, `https://web.dev`. On Windows, the root directory of the user-visible file system is `C:\`. The equivalent for the origin private file system is an initially empty root directory per origin accessed by calling the asynchronous method `navigator.storage.getDirectory()`. For a comparison of the user-visible file system and the origin private file system, see the following diagram. The diagram shows that apart from the root directory, everything else is conceptually the same, with a hierarchy of files and folders to organize and arrange as needed for your data and storage needs.

Diagram of the user-visible file system and the origin private file system with two exemplary file hierarchies. The entry point for the user-visible file system is a symbolic harddisk, the entry point for the origin private file system is calling of the method 'navigator.storage.getDirectory'.

Specifics of the origin private file system

Just like other storage mechanisms in the browser (for example, localStorage or IndexedDB), the origin private file system is subject to browser quota restrictions. When a user clears all browsing

data or all site data, the origin private file system will be deleted, too. Call `navigator.storage.estimate()` and in the resulting response object see the `usage` entry to see how much storage your app already consumes, which is broken down by storage mechanism in the `usageDetails` object, where you want to look at the `fileSystem` entry specifically. Since the origin private file system is not visible to the user, there are no permissions prompts and no Safe Browsing checks.

Getting access to the root directory

To get access to the root directory, run the command below. You end up with an empty directory handle, more specifically, a `FileSystemDirectoryHandle`.

```
const opfsRoot = await navigator.storage.getDirectory();
// A FileSystemDirectoryHandle whose type is "directory"
// and whose name is "".
console.log(opfsRoot);
```

Main thread or Web Worker

There are two ways of using the origin private file system: on the main thread or in a Web Worker. Web Workers cannot block the main thread, which means in this context APIs can be synchronous, a pattern generally disallowed on the main thread. Synchronous APIs can be faster as they avoid having to deal with promises, and file operations are typically synchronous in languages like C that can be compiled to WebAssembly.

```
// This is synchronous C code.
FILE *f;
f = fopen("example.txt", "w+");
fputs("Some text\n", f);
fclose(f);
```

If you need the fastest possible file operations and/or you deal with WebAssembly, skip down to Using the origin private file system in a Web Worker. Else, you can read on.

Using the origin private file system on the main thread

Creating new files and folders

Once you have a root folder, create files and folders using the `getFileHandle()` and the `getDirectoryHandle()` methods respectively. By passing `{create: true}`, the file or folder will be created if it doesn't exist. Build up a hierarchy of files by calling these functions using a newly created directory as the starting point.

```
const fileHandle = await opfsRoot
    .getFileHandle('my first file', {create: true});
const directoryHandle = await opfsRoot
    .getDirectoryHandle('my first folder', {create: true});
const nestedFileHandle = await directoryHandle
    .getFileHandle('my first nested file', {create: true});
const nestedDirectoryHandle = await directoryHandle
    .getDirectoryHandle('my first nested folder', {create: true});
```

The resulting file hierarchy from the earlier code sample.

Accessing existing files and folders

If you know their name, access previously created files and folders by calling the `getFileHandle()` or the `getDirectoryHandle()` methods, passing in the name of the file or folder.

```
const existingFileHandle = await opfsRoot.getFileHandle('my first file');
const existingDirectoryHandle = await opfsRoot
    .getDirectoryHandle('my first folder);
```

Getting the file associated with a file handle for reading

A `FileSystemFileHandle` represents a file on the file system. To obtain the associated `File`, use the `getFile()` method. A `File` object is a specific kind of `Blob`, and can be used in any context that a

`Blob` can. In particular, `FileReader`, `URL.createObjectURL()`, `createImageBitmap()`, and `XMLHttpRequest.send()` accept both `Blobs` and `Files`. If you will, obtaining a `File` from a `FileSystemFileHandle` "frees" the data, so you can access it and make it available to the user-visible file system.

```
const file = await fileHandle.getFile();
console.log(await file.text());
```

Writing to a file by streaming

Stream data into a file by calling `createWritable()` which creates a `FileSystemWritableFileStream` to that you then `write()` the contents. At the end, you need to `close()` the stream.

```
const contents = 'Some text';
// Get a writable stream.
const writable = await fileHandle.createWritable();
// Write the contents of the file to the stream.
await writable.write(contents);
// Close the stream, which persists the contents.
await writable.close();
```

Deleting files and folders

Delete files and folders by calling their file or directory handle's particular `remove()` method. To delete a folder including all subfolders, pass the `{recursive: true}` option.

```
await fileHandle.remove();
await directoryHandle.remove({recursive: true});
```

The `remove()` method is currently only implemented in Chrome. You can feature-detect support via `'remove' in FileSystemFileHandle.prototype`.

As an alternative, if you know the name of the to-be-deleted file or folder in a directory, use the `removeEntry()` method.

```
directoryHandle.removeEntry('my first nested file');
```

As a quick tip, `await (await navigator.storage.getDirectory()).remove({recursive: true})` is the fastest way to clear the entire origin private file system.

Moving and renaming files and folders

Rename and move files and folders using the `move()` method. Moving and renaming can happen together or in isolation.

```
// Rename a file.
await fileHandle.move('my first renamed file');
// Move a file to another directory.
await fileHandle.move(nestedDirectoryHandle);
// Move a file to another directory and rename it.
await fileHandle
    .move(nestedDirectoryHandle, 'my first renamed and now nested file');
```

Renaming and moving folders is not implemented yet in Chrome. You also can't move files from the origin private file system to the user-visible file system. You can copy them, though.

Resolving the path of a file or folder

To learn where a given file or folder is located in relation to a reference directory, use the `resolve()` method, passing it a `FileSystemHandle` as the argument. To obtain the full path of a file or folder in the origin private file system, use the root directory as the reference directory obtained via `navigator.storage.getDirectory()`.

```
const relativePath = await opfsRoot.resolve(nestedDirectoryHandle);
// `relativePath` is `['my first folder', 'my first nested folder']`.
```

Checking if two file or folder handles point to the same file or folder

Sometimes you have two handles and don't know if they point at the same file or folder. To check whether this is the case, use the `isSameEntry()` method.

```
fileHandle.isSameEntry(nestedFileHandle);
// Returns `false`.
```

Listing the contents of a folder

`FileSystemDirectoryHandle` is an asynchronous iterator that you iterate over with a `for await…of` loop. As an asynchronous iterator, it also supports the `entries()`, the `values()`, and the `keys()` methods, from which you can choose depending on what information you need:

```
for await (let [name, handle] of directoryHandle) {}
for await (let [name, handle] of directoryHandle.entries()) {}
for await (let handle of directoryHandle.values()) {}
for await (let name of directoryHandle.keys()) {}
```

Recursively listing the contents of a folder and all subfolders

Dealing with asynchronous loops and functions paired with recursion is easy to get wrong. The function below can serve as a starting point for listing the contents of a folder and all its subfolders, including all files and their sizes. You can simplify the function if you don't need the file sizes by, where it says `directoryEntryPromises.push`, not pushing the `handle.getFile()` promise, but the `handle` directly.

```
  const getDirectoryEntriesRecursive = async (
    directoryHandle,
    relativePath = '.',
  ) => {
    const fileHandles = [];
    const directoryHandles = [];
    const entries = {};
    // Get an iterator of the files and folders in the directory.
    const directoryIterator = directoryHandle.values();
    const directoryEntryPromises = [];
    for await (const handle of directoryIterator) {
      const nestedPath = `${relativePath}/${handle.name}`;
      if (handle.kind === 'file') {
        fileHandles.push({ handle, nestedPath });
        directoryEntryPromises.push(
          handle.getFile().then((file) => {
            return {
              name: handle.name,
              kind: handle.kind,
```

```
            size: file.size,
            type: file.type,
            lastModified: file.lastModified,
            relativePath: nestedPath,
            handle
          };
        }),
      );
    } else if (handle.kind === 'directory') {
      directoryHandles.push({ handle, nestedPath });
      directoryEntryPromises.push(
        (async () => {
          return {
            name: handle.name,
            kind: handle.kind,
            relativePath: nestedPath,
            entries:
                await getDirectoryEntriesRecursive(handle, nestedPath),
            handle,
          };
        })(),
      );
    }
  }
  const directoryEntries = await Promise.all(directoryEntryPromises);
  directoryEntries.forEach((directoryEntry) => {
    entries[directoryEntry.name] = directoryEntry;
  });
  return entries;
};
```

Using the origin private file system in a Web Worker

As outlined before, Web Workers can't block the main thread, which is why in this context synchronous methods are allowed.

Getting a synchronous access handle

The entry point to the fastest possible file operations is a `FileSystemSyncAccessHandle`, obtained from a regular `FileSystemFileHandle` by calling `createSyncAccessHandle()`.

```
const fileHandle = await opfsRoot
    .getFileHandle('my highspeed file.txt', {create: true});
const syncAccessHandle = await fileHandle.createSyncAccessHandle();
```

It might seem confusing, but you actually get a *synchronous* `FileSystemSyncAccessHandle` from a regular `FileSystemFileHandle`. Also note that the `createSyncAccessHandle()` method is *asynchronous*, despite the `Sync` in its name.

Synchronous in-place file methods

Once you have a synchronous access handle, you get access to fast in-place file methods that are all synchronous.

- `getSize()`: Returns the size of the file in bytes.
- `write()`: Writes the content of a buffer into the, optionally at a given offset, and returns the number of written bytes. Checking the returned number of written bytes allows callers to detect and handle errors and partial writes.
- `read()`: Reads the contents of the file into a buffer, optionally at a given offset.
- `truncate()`: Resizes the file to the given size.
- `flush()`: Ensures that the contents of the file contain all the modifications done through `write()`.
- `close()`: Closes the access handle.

Here is an example that uses all the methods mentioned above.

```
const opfsRoot = await navigator.storage.getDirectory();
const fileHandle = await opfsRoot.getFileHandle('fast', {create: true});
const accessHandle = await fileHandle.createSyncAccessHandle();

const textEncoder = new TextEncoder();
const textDecoder = new TextDecoder();

// Initialize this variable for the size of the file.
let size;
// The current size of the file, initially `0`.
size = accessHandle.getSize();
```

```
// Encode content to write to the file.
const content = textEncoder.encode('Some text');
// Write the content at the beginning of the file.
accessHandle.write(content, {at: size});
// Flush the changes.
accessHandle.flush();
// The current size of the file, now `9` (the length of "Some text").
size = accessHandle.getSize();

// Encode more content to write to the file.
const moreContent = textEncoder.encode('More content');
// Write the content at the end of the file.
accessHandle.write(moreContent, {at: size});
// Flush the changes.
accessHandle.flush();
// The current size of the file, now `21` (the length of
// "Some textMore content").
size = accessHandle.getSize();

// Prepare a data view of the length of the file.
const dataView = new DataView(new ArrayBuffer(size));

// Read the entire file into the data view.
accessHandle.read(dataView);
// Logs `"Some textMore content"`.
console.log(textDecoder.decode(dataView));

// Read starting at offset 9 into the data view.
accessHandle.read(dataView, {at: 9});
// Logs `"More content"`.
console.log(textDecoder.decode(dataView));

// Truncate the file after 4 bytes.
accessHandle.truncate(4);
```

Note that the first parameter for `read()` and `write()` is an `ArrayBuffer` or an `ArrayBufferView` like a `DataView`. You cannot directly manipulate the contents of an `ArrayBuffer`. Instead, you create one of the typed array objects like an `Int8Array` or a `DataView` object which represents the buffer in a specific format, and use that to read and write the contents of the buffer.

Copying a file from the origin private file system to the user-visible file system

As mentioned above, moving files from the origin private file system to the user-visible file system isn't possible, but you can copy files. Since `showSaveFilePicker()` is only exposed on the main thread, but not in the Worker thread, be sure to run the code there.

```
// On the main thread, not in the Worker. This assumes
// `fileHandle` is the `FileSystemFileHandle` you obtained
// the `FileSystemSyncAccessHandle` from in the Worker
// thread. Be sure to close the file in the Worker thread first.
const fileHandle = await opfsRoot.getFileHandle('fast');
try {
  // Obtain a file handle to a new file in the user-visible file system
  // with the same name as the file in the origin private file system.
  const saveHandle = await showSaveFilePicker({
    suggestedName: fileHandle.name || ''
  });
  const writable = await saveHandle.createWritable();
  await writable.write(await fileHandle.getFile());
  await writable.close();
} catch (err) {
  console.error(err.name, err.message);
}
```

Debugging the origin private file system

Until built-in DevTools support is added (see crbug/1284595), use the OPFS Explorer Chrome extension to debug the origin private file system. The screenshot above from the section Creating new files and folders is taken straight from the extension by the way.

The OPFS Explorer Chrome DevTools extension in the Chrome Web Store.

After installing the extension, open the Chrome DevTools, select the **OPFS Explorer** tab, and you're then ready to inspect the file hierarchy. Save files from the origin private file system to the user-visible file system by clicking the file name and delete files and folders by clicking the trash can icon.

Demo

See the origin private file system in action (if you install the OPFS Explorer extension) in a demo that uses it as a backend for a SQLite database compiled to WebAssembly. Be sure to check out the source code on Glitch. Note how the embedded version below does not use the origin private file system backend (because the iframe is cross-origin), but when you open the demo in a separate tab, it does.

https://glitch.com/edit/#!/sqlite-wasm-opfs

Conclusions

The origin private file system, as specified by the WHATWG, has shaped the way we use and interact with files on the web. It has enabled new use cases that were impossible to achieve with the user-visible file system. All major browser vendors—Apple, Mozilla, and Google—are on-board and share a joint vision. The development of the origin private file system is very much a collaborative effort, and feedback from developers and users is essential to its progress. As we continue to refine and improve the standard, feedback on the whatwg/fs repository in the form of Issues or Pull Requests is welcome.

Related links

- File System Standard spec
- File System Standard repo
- The File System API with Origin Private File System WebKit post
- OPFS Explorer extension

Acknowledgements

This article was reviewed by Austin Sully, Etienne Noël, and Rachel Andrew. Hero image by Christina Rumpf on Unsplash.

□□□ #3
□ □□□ □□□ 2026-10-31 22:08:25 UTC
□ cocobook-sync □□□ 2026-11-04 17:37:44 UTC