As we move from an offline world to a completely online world, we're demanding more from the Web, and more from web applications. Browser implementers are adding richer APIs by the day to support complex use cases. APIs for things like real-time communication, graphics, and client-side (offline) storage.

One area where the Web has lacked for some time is file I/O. Interacting with binary data and organizing that data into a meaningful hierarchy of folders is something desktop software has been capable of for decades. How amazing would it be if web apps could do the same? The lack of true filesystem access has hindered web applications from moving forward. For example, how can a photo gallery work offline without being able to save images locally? The answer is it can't! We need something more powerful.

The HTML5 File API: Directories and System aims to fill this void. The specification defines a means for web applications to read, create, navigate, and write to a sandboxed section of the user's local filesystem. The entirety of the Filesystem API can be broken down into a number of different related specifications:

- Reading and manipulating files: `File`/`Blob`, `FileList`, `FileReader`
- Creating and writing: `BlobBuilder`, `FileWriter`
- Directories and filesystem access: `DirectoryReader`, `FileEntry`/`DirectoryEntry`, `LocalFileSystem`

The specification defines two versions (asynchronous and synchronous) of the same API. The asynchronous API is useful for normal applications and prevents blocking UI actions. The synchronous API is reserved for use in Web Workers.

Use Cases

HTML5 has several storage options available. The Filesystem API is different in that it aims to satisfy client-side storage use cases not well served by databases such as IndexedDB or WebSQL DB. Generally, these are applications that deal with large binary blobs and share data with applications outside of the context of the browser. The specification lists several use cases worth highlighting:

- Persistent uploader
    - When a file or directory is selected for upload, it copies the files into a local sandbox and uploads a chunk at a time.
    - Uploads can be restarted after browser crashes, network interruptions, etc.
- Video game, music, or other apps with lots of media assets
    - It downloads one or several large tarballs, and expands them locally into a directory structure.
    - The same download works on any operating system.
    - It can manage prefetching just the next-to-be-needed assets in the background, so going to the next game level or activating a new feature doesn't require waiting for a download.
    - It uses those assets directly from its local cache, by direct file reads or by handing local URIs to image or video tags, WebGL asset loaders, etc.
    - The files may be of arbitrary binary format.
    - On the server side, a compressed tarball is often much smaller than a collection of separately compressed files. Also, one tarball instead of a 1,000 little files involves fewer seeks.
- Audio/Photo editor with offline access or local cache for speed
    - The data blobs are potentially quite large, and are read-write.
    - It might want to do partial writes to files (overwriting just the ID3/EXIF tags, for example).

    - The ability to organize project files by creating directories is important.
    - Edited files should be accessible by client-side applications (iTunes, Picasa).
- Offline video viewer
    - It downloads large files (>1 GB) for later viewing.
    - It needs efficient seek and streaming.
    - It should be able to hand a URI to the video tag.
    - It should enable access to partly downloaded files (for example, to let you watch the first episode of the DVD even if your download didn't complete before you got on the plane.)
    - It should be able to pull a single episode out of the middle of a download and give just that to the video tag.
- Offline web mail client
    - Downloads attachments and stores them locally.
    - Caches user-selected attachments for later upload.
    - Needs to be able to refer to cached attachments and image thumbnails for display and upload.
    - Should be able to trigger the UA's download manager just as if talking to a server.
    - Should be able to upload an email with attachments as a multipart post, rather than sending a file at a time in an XHR.

Security Considerations

The HTML5 Filesystem API can be used to read and write data to parts of the user's hard drive. Because of this privileged access, there are a number of security and privacy issues that have been considered in the API's design. A few are listed below:

- Local disk usage and IO bandwidth—this is mitigated in part through quota limitations. See Chapter 2, *Storage and Quota*.
- Leakage or erasure of private data—this is mitigated by limiting the scope of the HTML5 filesystem to a chroot-like, origin-specific sandbox. Applications cannot access another domain/origin's filesystem.
- Storing malicious executables or illegal data on a user's system—with any download there is a risk. The API mitigates against malicious executables by restricting file creation/rename to nonexecutable extensions, and by making sure the execute bit is not set on any file created or modified via the API.

Browser Support

At the time of writing, Google Chrome is the only browser to implement the Filesystem API. Version 8 of the browser was the first to see a partial implementation, but the majority of the API was later completed in version 11. In Chrome 13, a Chapter 2, *Storage and Quota* API was added to give applications a way to request addition space for storing data.

A Cautionary Tale

Before we dive in, I want to remind you that this book covers a working implementation of an evolving specification, a spec that has yet to be finalized by the World Wide Web Consortium (W3C). Take my word of caution and realize that until the spec is final, portions of the API could change.