

Using the FileSystem APIs



By [Eric Bidelman](#)

Published: January 4th, 2011

Updated: July 30th, 2013

Comments: [0](#)

Heads up!

In April 2014, it was [announced on public-webapps](#) that the [Filesystem API spec](#) is not being considered by other browsers. For now, the API is Chrome-specific and it's unlikely to be implemented by other browsers and is no longer being standardized with the W3C.

Introduction

I've always thought it would be handy if web applications could read and write files and directories. As we move from offline to online, applications are becoming more complex and the lack of file system APIs has been a hindrance for moving the web forward. Storing or interacting with binary data shouldn't be limited to the desktop. Thankfully, it no longer is thanks to the [FileSystem API](#). With the FileSystem API, a web app can create, read, navigate, and write to a sandboxed section of the user's local file system.

The API is broken up into various themes:

- Reading and manipulating files: [File/Blob](#), [FileList](#), [FileReader](#)
- Creating and writing: [Blob\(\)](#), [FileWriter](#)
- Directories and file system access: [DirectoryReader](#), [FileEntry/DirectoryEntry](#), [LocalFileSystem](#)

Browser support & storage limitations

At the time of writing this article, Google Chrome has the only working implementation of the FileSystem API. A dedicated browser UI does not yet exist for file/quota management. To store data on the user's system, may require your app to [request quota](#). However, for testing, Chrome can be run with the `--unlimited-quota-for-files` flag. Furthermore, if you're building an app or extension for the Chrome Web Store, the unlimitedStorage [manifest file](#) permission can be used in place of requesting quota. Eventually, users will receive a permission dialog to grant, deny, or increase storage for an app.

You may need the `--allow-file-access-from-files` flag if you're debugging your app from `file://`. Not using these flags will result in a `SECURITY_ERR` or `QUOTA_EXCEEDED_ERR` `FileError`.

Requesting a file system

A web app can request access to a sandboxed file system by calling `window.requestFileSystem()`:

```
// Note: The file system has been prefixed as of Google Chrome 12:  
window.requestFileSystem = window.requestFileSystem || window.webkitRequestFileSystem;
```

```
window.requestFileSystem(type, size, successCallback, opt_errorCallback)
```

type

Whether the file storage should be persistent. Possible values are `window.TEMPORARY` or `window.PERSISTENT`. Data stored using `TEMPORARY` can be removed at the browser's discretion (for example if more space is needed). `PERSISTENT` storage cannot be cleared unless explicitly authorized by the user or the app and requires the user to grant quota to your app. See [requesting quota](#).

size

Size (in bytes) the app will require for storage.

successCallback

Callback that is invoked on successful request of a file system. Its argument is a [FileSystem](#) object.

opt_errorCallback

Optional callback for handling errors or when the request to obtain the file system is denied. Its argument is a [FileError](#) object.

If you're calling `requestFileSystem()` for the first time, new storage is created for your app. It's important to remember that this file system is sandboxed, meaning one web app cannot access another app's files. This also means you cannot read/write files to an arbitrary folder on the user's hard drive (for example My Pictures, My Documents, etc.).

Example usage:

```
function onInitFs(fs) {
  console.log('Opened file system: ' + fs.name);
}

window.requestFileSystem(window.TEMPORARY, 5*1024*1024 /*5MB*/, onInitFs,
  errorHandler);
```

The `FileSystem` specification also defines a synchronous API, [LocalFileSystemSync](#) interface that is intended to be used in [Web Workers](#). However, this tutorial will not cover the synchronous API.

Throughout the remainder of this document, we'll use the same handler for processing errors from the asynchronous calls:

```
function errorHandler(e) {
  var msg = '';

  switch (e.code) {
    case FileError.QUOTA_EXCEEDED_ERR:
      msg = 'QUOTA_EXCEEDED_ERR';
      break;
    case FileError.NOT_FOUND_ERR:
      msg = 'NOT_FOUND_ERR';
      break;
    case FileError.SECURITY_ERR:
      msg = 'SECURITY_ERR';
      break;
    case FileError.INVALID_MODIFICATION_ERR:
      msg = 'INVALID_MODIFICATION_ERR';
      break;
    case FileError.INVALID_STATE_ERR:
      msg = 'INVALID_STATE_ERR';
      break;
    default:
      msg = 'Unknown Error';
      break;
  }
}
```

```
};

console.log('Error: ' + msg);
}
```

Granted, this error callback is very generic, but you get the idea. You'll want to provide human-readable messages to users instead.

Requesting storage quota

To use PERSISTENT storage, you must obtain permission from the user to store persistent data. The same restriction doesn't apply to TEMPORARY storage because the browser may choose to evict temporarily stored data at its discretion.

To use PERSISTENT storage with the FileSystem API, Chrome exposes a new API under `window.webkitStorageInfo` to request storage:

```
window.webkitStorageInfo.requestQuota(PERSISTENT, 1024*1024, function(grantedBytes) {
  window.requestFileSystem(PERSISTENT, grantedBytes, onInitFs, errorHandler);
}, function(e) {
  console.log('Error', e);
});
```

Once the user has granted permission, there's no need to call `requestQuota()` in the future (unless you wish to increase your app's quota). Subsequent calls for equal or lesser quota are a no-op.

There is also [an API](#) to query an origin's current quota usage and allocation:

`window.webkitStorageInfo.queryUsageAndQuota()`

Working with files

Files in the sandboxed environment are represented by the [FileEntry](#) interface. A `FileEntry` contains the types of properties (`name`, `isFile`, ...) and methods (`remove`, `moveTo`, `copyTo`, ...) that you'd expect from a standard file system.

Properties and methods of `FileEntry`:

```
fileEntry.isFile === true
fileEntry.isDirectory === false
fileEntry.name
fileEntry.fullPath
...

fileEntry.getMetadata(successCallback, opt_errorCallback);
fileEntry.remove(successCallback, opt_errorCallback);
fileEntry.moveTo(dirEntry, opt_newName, opt_successCallback, opt_errorCallback);
fileEntry.copyTo(dirEntry, opt_newName, opt_successCallback, opt_errorCallback);
fileEntry.getParent(successCallback, opt_errorCallback);
fileEntry.toURL(opt_mimeType);

fileEntry.file(successCallback, opt_errorCallback);
fileEntry.createWriter(successCallback, opt_errorCallback);
...
```

To better understand `FileEntry`, the rest of this section contains a bunch of recipes for performing common tasks.

Creating a file

You can look up or create a file with the file system's `getFile()` method, a method of the [DirectoryEntry](#) interface. After requesting a file system, the success callback is passed a `FileSystem` object that contains a `DirectoryEntry` (`fs.root`) pointing to the root of the app's file system.

The following code creates an empty file called "log.txt" in the root of the app's file system:

```
function onInitFs(fs) {

  fs.root.getFile('log.txt', {create: true, exclusive: true}, function(fileEntry) {

    // fileEntry.isFile === true
    // fileEntry.name == 'log.txt'
    // fileEntry.fullPath == '/log.txt'

    }, errorHandler);

}

window.requestFileSystem(window.TEMPORARY, 1024*1024, onInitFs, errorHandler);
```

Once the file system has been requested, the success handler is passed a `FileSystem` object. Inside the callback, we can call `fs.root.getFile()` with the name of the file to create. You can pass an absolute or relative path, but it must be valid. For instance, it is an error to attempt to create a file whose immediate parent does not exist. The second argument to `getFile()` is an object literal describing the function's behavior if the file does not exist. In this example, `create: true` creates the file if it doesn't exist and throws an error if it does (`exclusive: true`). Otherwise (if `create: false`), the file is simply fetched and returned. In either case, the file contents are not overwritten because we're just obtaining a reference entry to the file in question.

Reading a file by name

The following code retrieves the file called "log.txt", its contents are read using the `FileReader` API and appended to a new `<textarea>` on the page. If `log.txt` doesn't exist, an error is thrown.

```
function onInitFs(fs) {

  fs.root.getFile('log.txt', {}, function(fileEntry) {

    // Get a File object representing the file,
    // then use FileReader to read its contents.
    fileEntry.file(function(file) {
      var reader = new FileReader();

      reader.onloadend = function(e) {
        var txtArea = document.createElement('textarea');
        txtArea.value = this.result;
        document.body.appendChild(txtArea);
      };

      reader.readAsText(file);
    }, errorHandler);

  }, errorHandler);

}

window.requestFileSystem(window.TEMPORARY, 1024*1024, onInitFs, errorHandler);
```

Writing to a file

The following code creates an empty file called "log.txt" (if it doesn't exist) and fills it with the text 'Lorem Ipsum'.

```
function onInitFs(fs) {

  fs.root.getFile('log.txt', {create: true}, function(fileEntry) {

    // Create a FileWriter object for our FileEntry (log.txt).
    fileEntry.createWriter(function(fileWriter) {

      fileWriter.onwriteend = function(e) {
        console.log('Write completed.');
```

This time, we call the FileEntry's createWriter() method to obtain a FileWriter object. Inside the success callback, event handlers are set up for error and writeend events. The text data is written to the file by creating a blob, appending text to it, and passing the blob to FileWriter.write().

Appending data to a file

The following code appends the text 'Hello World' to the end of our log file. An error is thrown if the file does not exist.

```
function onInitFs(fs) {

  fs.root.getFile('log.txt', {create: false}, function(fileEntry) {

    // Create a FileWriter object for our FileEntry (log.txt).
    fileEntry.createWriter(function(fileWriter) {

      fileWriter.seek(fileWriter.length); // Start write position at EOF.

      // Create a new Blob and write it to log.txt.
      var blob = new Blob(['Hello World'], {type: 'text/plain'});

      fileWriter.write(blob);

    }, errorHandler);

  }, errorHandler);

}

window.requestFileSystem(window.TEMPORARY, 1024*1024, onInitFs, errorHandler);
```

Duplicating user-selected files

The following code allows a user to select multiple files using `<input type="file" multiple />` and creates copies of those files in the app's sandboxed file system.

```
<input type="file" id="myfile" multiple />
```

```
document.querySelector("#myfile").onchange = function(e) {
  var files = this.files;

  window.requestFileSystem(window.TEMPORARY, 1024*1024, function(fs) {
    // Duplicate each file the user selected to the app's fs.
    for (var i = 0, file; file = files[i]; ++i) {

      // Capture current iteration's file in local scope for the getFile() callback.
      (function(f) {
        fs.root.getFile(f.name, {create: true, exclusive: true}, function(fileEntry) {
          fileEntry.createWriter(function(fileWriter) {
            fileWriter.write(f); // Note: write() can take a File or Blob object.
          }, errorHandler);
        }, errorHandler);
      })(file);

    }
  }, errorHandler);
};
```

Although we've used an input for the file import, one could easily leverage [HTML5 Drag and Drop](#) to achieve the same objective.

As noted in the comment, `FileWriter.write()` can accept a Blob or File. This is because File inherits from Blob. Therefore, all file objects are blobs.

Removing a file

The following code deletes the file 'log.txt'.

```
window.requestFileSystem(window.TEMPORARY, 1024*1024, function(fs) {
  fs.root.getFile('log.txt', {create: false}, function(fileEntry) {

    fileEntry.remove(function() {
      console.log('File removed.');
}, errorHandler);

  }, errorHandler);
}, errorHandler);
```

Working with directories

Directories in the sandbox are represented by the [DirectoryEntry](#) interface, which shares most of [FileEntry](#)'s properties (they inherit from a common Entry interface). However, DirectoryEntry has additional methods for manipulating directories.

Properties and methods of DirectoryEntry:

```
dirEntry.isDirectory === true
// See the section on FileEntry for other inherited properties/methods.
```

```
...
```

```
var dirReader = dirEntry.createReader();
dirEntry.getFile(path, opt_flags, opt_successCallback, opt_errorCallback);
dirEntry.getDirectory(path, opt_flags, opt_successCallback, opt_errorCallback);
dirEntry.removeRecursively(successCallback, opt_errorCallback);
```

```
...
```

Creating directories

Use the `getDirectory()` method of `DirectoryEntry` to read or create directories. You can pass either a name or path as the directory to look up or create.

For example, the following code creates a directory named "MyPictures" in the root directory:

```
window.requestFileSystem(window.TEMPORARY, 1024*1024, function(fs) {
  fs.root.getDirectory('MyPictures', {create: true}, function(dirEntry) {
    ...
  }, errorHandler);
}, errorHandler);
```

Subdirectories

Creating a subdirectory is exactly the same as creating any other directory. However, the API throws an error if you attempt to create a directory whose immediate parent does not exist. The solution is to create each directory sequentially, which is rather tricky to do with an asynchronous API.

The following code creates a new hierarchy (music/genres/jazz) in the root of the app's FileSystem by recursively adding each subdirectory after its parent folder has been created.

```
var path = 'music/genres/jazz/';

function createDir(rootDirEntry, folders) {
  // Throw out '.' or '/' and move on to prevent something like '/foo/./bar'.
  if (folders[0] == '.' || folders[0] == '/') {
    folders = folders.slice(1);
  }
  rootDirEntry.getDirectory(folders[0], {create: true}, function(dirEntry) {
    // Recursively add the new subfolder (if we still have another to create).
    if (folders.length) {
      createDir(dirEntry, folders.slice(1));
    }
  }, errorHandler);
};

function onInitFs(fs) {
  createDir(fs.root, path.split('/')); // fs.root is a DirectoryEntry.
}

window.requestFileSystem(window.TEMPORARY, 1024*1024, onInitFs, errorHandler);
```

Now that "music/genres/jazz" is in place, we can pass its full path to `getDirectory()` and create new subfolders or files under it. For example:

```
window.requestFileSystem(window.TEMPORARY, 1024*1024, function(fs) {
  fs.root.getFile('/music/genres/jazz/song.mp3', {create: true}, function(fileEntry) {
    ...
  }, errorHandler);
```

```
}, errorHandler);
```

Reading a directory's contents

To read the contents of a directory, create a `DirectoryReader` and call its `readEntries()` method. There is no guarantee that all of a directory's entries will be returned in a single call to `readEntries()`. That means you need to keep calling `DirectoryReader.readEntries()` until no more results are returned. The following is code that demonstrates this:

```
<ul id="filelist"></ul>
```

```
function toArray(list) {
  return Array.prototype.slice.call(list || [], 0);
}

function listResults(entries) {
  // Document fragments can improve performance since they're only appended
  // to the DOM once. Only one browser reflow occurs.
  var fragment = document.createDocumentFragment();

  entries.forEach(function(entry, i) {
    var img = entry.isDirectory ? '' :
      '';
    var li = document.createElement('li');
    li.innerHTML = [img, '<span>', entry.name, '</span>'].join("");
    fragment.appendChild(li);
  });

  document.querySelector('#filelist').appendChild(fragment);
}

function onInitFs(fs) {
  var dirReader = fs.root.createReader();
  var entries = [];

  // Call the reader.readEntries() until no more results are returned.
  var readEntries = function() {
    dirReader.readEntries(function(results) {
      if (!results.length) {
        listResults(entries.sort());
      } else {
        entries = entries.concat(toArray(results));
        readEntries();
      }
    }, errorHandler);
  };

  readEntries(); // Start reading dirs.
}

window.requestFileSystem(window.TEMPORARY, 1024*1024, onInitFs, errorHandler);
```

Removing a directory

The `DirectoryEntry.remove()` method behaves just like `FileEntry`'s. The difference: attempting to delete a non-empty directory results in an error.

The following removes the empty directory "jazz" from `"/music/genres/"`:


```

window.requestFileSystem(window.TEMPORARY, 1024*1024, function(fs) {
  fs.root.getDirectory('music/genres/jazz', {}, function(dirEntry) {

    dirEntry.remove(function() {
      console.log('Directory removed.');
```

```
    }, errorHandler);
```

```
  }, errorHandler);
}, errorHandler);
```

Recursively removing a directory

If you have a pesky directory that contains entries, `removeRecursively()` is your friend. It deletes the directory and its contents, recursively.

The following code recursively removes the directory "music" and all the files and directories that it contains:

```

window.requestFileSystem(window.TEMPORARY, 1024*1024, function(fs) {
  fs.root.getDirectory('/misc/./music', {}, function(dirEntry) {

    dirEntry.removeRecursively(function() {
      console.log('Directory removed.');
```

```
    }, errorHandler);
```

```
  }, errorHandler);
}, errorHandler);
```

Copying, renaming, and moving

`FileEntry` and `DirectoryEntry` share common operations.

Copying an entry

Both `FileEntry` and `DirectoryEntry` have a `copyTo()` for duplicating existing entries. This method automatically does a recursive copy on folders.

The following code example copies the file "me.png" from one directory to another:

```

function copy(cwd, src, dest) {
  cwd.getFile(src, {}, function(fileEntry) {

    cwd.getDirectory(dest, {}, function(dirEntry) {
      fileEntry.copyTo(dirEntry);
    }, errorHandler);

  }, errorHandler);
}

window.requestFileSystem(window.TEMPORARY, 1024*1024, function(fs) {
  copy(fs.root, 'folder1/me.png', 'folder2/mypics/');
```

```
}, errorHandler);
```

Moving or renaming an entry

The `moveTo()` method present in `FileEntry` and `DirectoryEntry` allows you to move or rename a file or directory. Its first argument is the parent directory to move the file under, and its second is an optional new name for the file. If a new name isn't provided, the file's original name is used.

The following example renames "me.png" to "you.png", but does not move the file:

```
function rename(cwd, src, newName) {
  cwd.getFile(src, {}, function(fileEntry) {
    fileEntry.moveTo(cwd, newName);
  }, errorHandler);
}

window.requestFileSystem(window.TEMPORARY, 1024*1024, function(fs) {
  rename(fs.root, 'me.png', 'you.png');
}, errorHandler);
```

The following example moves "me.png" (located in the root directory) to a folder named "newfolder".

```
function move(src, dirName) {
  fs.root.getFile(src, {}, function(fileEntry) {

    fs.root.getDirectory(dirName, {}, function(dirEntry) {
      fileEntry.moveTo(dirEntry);
    }, errorHandler);

  }, errorHandler);
}

window.requestFileSystem(window.TEMPORARY, 1024*1024, function(fs) {
  move('/me.png', 'newfolder/');
}, errorHandler);
```

filesystem: URLs

The FileSystem API exposes a new URL scheme, `filesystem:`, that can be used to fill `src` or `href` attributes. For example, if you wanted to display an image and have its [fileEntry](#), calling `toURL()` would give you the file's filesystem: URL:

```
var img = document.createElement('img');
img.src = fileEntry.toURL(); // filesystem:http://example.com/temporary/myfile.png
document.body.appendChild(img);
```

Alternatively, if you already have a filesystem: URL, `resolveLocalFileSystemURL()` will get you back the [fileEntry](#):

```
window.resolveLocalFileSystemURL = window.resolveLocalFileSystemURL ||
  window.webkitResolveLocalFileSystemURL;

var url = 'filesystem:http://example.com/temporary/myfile.png';
window.resolveLocalFileSystemURL(url, function(fileEntry) {
  ...
});
```

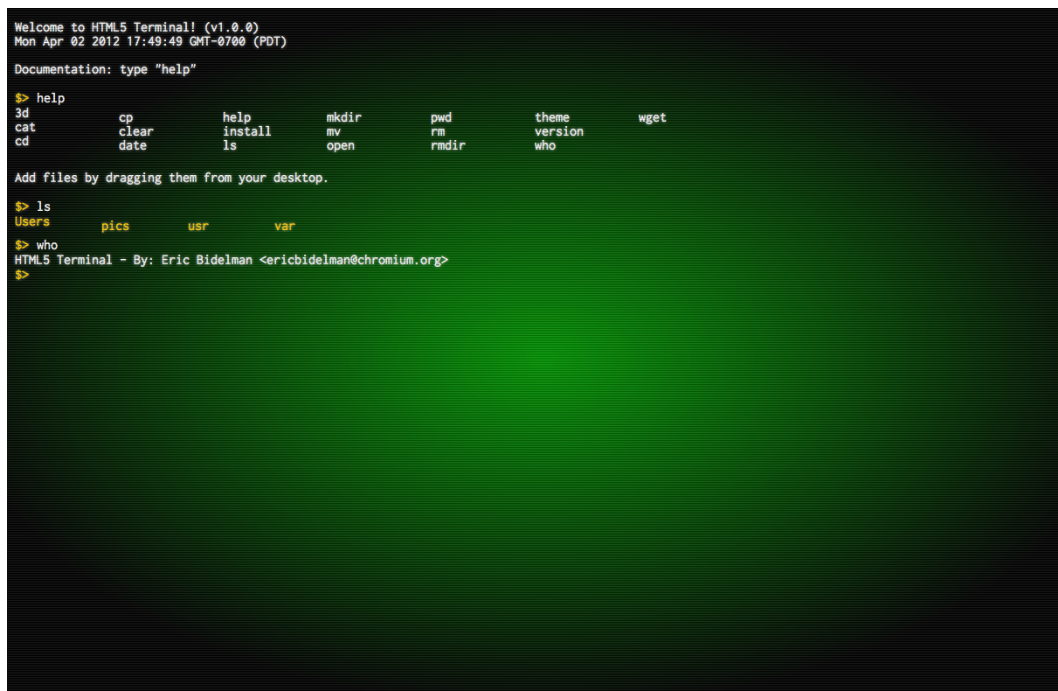
Putting it all together

Basic example

This demo lists the files/folders in the filesystem.

HTML5 Terminal

This shell replicates some of the common operations in a UNIX filesystem (such as `cd`, `mkdir`, `rm`, `open`, and `cat`) by abstracting the FileSystem API. To add content, open the app, then drag and drop files from your desktop onto the terminal window.



```
Welcome to HTML5 Terminal! (v1.0.0)
Mon Apr 02 2012 17:49:49 GMT-0700 (PDT)
Documentation: type "help"

> help
3d      cp      help    mkdir   pwd      theme    wget
cat     clear   install mv      rm       version
cd      date    ls      open    rmdir    who

Add files by dragging them from your desktop.

> ls
Users   pics    usr     var

> who
HTML5 Terminal - By: Eric Bidelman <ericbidelman@chromium.org>
>
```

[Open the HTML5 Terminal](#)

Use Cases

There are several [storage options](#) available in HTML5, but the FileSystem is different in that it aims to satisfy client-side storage use cases not well served by databases. Generally, these are applications that deal with large binary blobs and/or share data with applications outside of the context of the browser.

The specification lists several use cases:

1. 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.

2. Video game, music, or other app 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 will often be much smaller than a collection of separately-compressed files. Also, 1 tarball instead of 1000 little files will involve fewer seeks, all else being equal.

3. Audio/Photo editor with offline access or local cache for speed

- The data blobs are potentially quite large, and are read-write.
- It may want to do partial writes to files (overwriting just the ID3/EXIF tags, for example).
- The ability to organize project files by creating directories would be useful.
- Edited files should be accessible by client-side applications [iTunes, Picasa].

4. Offline video viewer

- It downloads large files (>1GB) for later viewing.
- It needs efficient seek + streaming.
- It must be able to hand a URI to the video tag.
- It should enable access to partly-downloaded files e.g. 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.

5. 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.

Reference specifications

- [FileSystem](#)
- [FileWriter](#)
- [FileReader](#)
- [File](#)
- [Blob](#)

[0 comments.](#)



Next steps

Share



Subscribe

Enjoyed this article? Grab the [RSS feed](#) and stay up-to-date.

Except as otherwise noted, the content of this page is licensed under the Creative Commons Attribution 3.0 License, and code samples are licensed under the Apache 2.0 License.