



# Clipcode's Guide to the W3C FILE API

## ***Specification***

There are three FILE API specifications:

Reading Files: <http://dev.w3.org/2006/webapi/FileAPI/>

Writing Files: <http://dev.w3.org/2009/dap/file-system/file-writer.html>

File System & Directory: <http://dev.w3.org/2009/dap/file-system/file-dir-sys.html>

## ***Compatibility***

<http://caniuse.com/#feat=fileapi>

## ***Overview***

The W3C FILE API is a set of three API specifications to allow a modern browser expose secure isolated file storage to web applications. Local filesystem access greatly improves the capabilities of client web applications – it can lead to increased performance and new features, and it allows users to work offline - and later, when reconnected to the web, to sync with the cloud. However, it also can have serious security implications (you don't want a random web site editing files in your /etc or C:\ProgramData directories!). Therefore the browser places tight restrictions on what scripts can and can't do and the file locations that can be used.

The three File API specifications are layered one above another, with File API (for reading) at the base, layered above that is File Writer API for exposing writing of individual files, and layered above that is File System API for interacting with file systems and directories.

The File API specs are still being written, so visiting the editor's draft (as in the above links) is how to kind up to date.

## ***Options for Client-Side Storage***

Modern browsers have a rich set of options when it comes to client-side storage.

Cookies have been around for along time, and a new cookies spec is in preparation:

<http://tools.ietf.org/html/draft-ietf-httpstate-cookie-23>

Three more recent options are Web Storage, File API and Indexed DB API.

Web storage (<http://dev.w3.org/html5/webstorage/>) is similar to cookies but offers more functionality, such as allowing values to be stored for specific browser windows, and storing larger amounts of data between sessions.

File API is a API to a client-side file system, that allows web applications to store data as files on the client's disk and access them in a later session, but aware of the security considerations to protect misuse.

Indexed DB API is a client-side database to store Javascript objects in an indexed storage (e.g. A B-tree database).

Availability? For now cookies are supported everywhere, Web Storage is very well supported in currently deployed browsers, and File API and Indexed DB API are still evolving and typically require a modern HTML5 web browser (these standards are independent of HTML5, but nonetheless, all browsers that implement File API and Indexed DB API also implement HTML 5).

When to use? Cookies are good when you need to have multiple windows in a browser share small amounts of data, web storage when the data is to be local to each window, File API when you need to store documents and Indexed DB API when you need a database.

For larger amounts of data, the choice is between File API and Indexed DB API, and the recommendation is to use the former where you would use a file system in a desktop/server app, and use the latter where you would use a database.

## ***Async and Sync***

When examining the File API and the Indexed DB API, there are two sets of APIs, one async and the other sync. The async API is to be used in the main UI thread and either the async or sync can be used in workers (the equivalent of background threads). A sync API blocks the current thread – and were this to happen in the UI thread, then the user would perceive the application to hang.

Though the sync API can be used in worker threads, we would have a preference to using the async API everywhere, as it leads to common code, and avoids un-necessary context switching.

The Sync APIs have the same names as the Async APIs with the one difference that “Sync is appended to the method names.

One important difference between the operation of the Async API and Sync API is with error handling. With Async, `XxxError` objects are created and error handlers are called, whereas with Sync APIs, `XxxExceptions` are thrown.

## FILE (Reader) API Concepts

The following concepts are part of the File API – Reader (API definitions taken from <http://www.w3.org/TR/FileAPI> and are copyright W3C):

**FileList** – A sequence of files, made available by an `<input type="file">` element of HTML5

```
interface FileList {
  getter File item(unsigned long index);
  readonly attribute unsigned long length;
}
```

FileList does not have a public constructor and there is no way of creating one manually.

Instead, an HTML5 page that has an input element (of type=file) lets the user select which file (s) should be made available to the FileList.

A FileList returns a list of files and may have zero, one or more in the list. The length attribute states how many are there, and `item(index)` returns the zero-based file object.

**Blob** – A blob of data

```
interface Blob {
  readonly attribute unsigned long long size;
  readonly attribute DOMString type;
  //slice Blob into byte-ranged chunks
  Blob slice(in optional long long start,
             in optional long long end,
             in [TreatUndefinedAs=EmptyString] optional DOMString contentType);
};
```

Blob does not have a constructor, so the only way to create a blob is either to get a File object (File inherits from Blob) or to call Slice on an existing blob (which creates a smaller blob out of a bigger one).

**File** – Represents a file

Note File inherits from Blob and adds two properties.

```
interface File : Blob {
  readonly attribute DOMString name;
  readonly attribute Date lastModifiedDate;
};
```

File does not have a constructor. File objects are accessed via the FileList (thus the user is in charge of which files the script within a web browser has access to).

**FileReader** – used to read a file

```
[Constructor]
interface FileReader: EventTarget {
  // async read methods
  void readAsArrayBuffer(in Blob blob) raises (OperationNotAllowedException);
  void readAsBinaryString(in Blob blob) raises (OperationNotAllowedException);
```

```

void readAsText(in Blob blob, optional in DOMString encoding) raises
(OperationNotAllowedException);

void readAsDataURL(in Blob blob) raises (OperationNotAllowedException);

void abort();

// states

const unsigned short EMPTY = 0;

const unsigned short LOADING = 1;

const unsigned short DONE = 2;

readonly attribute unsigned short readyState;

// File or Blob data

readonly attribute any result;

readonly attribute FileError error;

// event handler attributes

attribute Function onloadstart;

attribute Function onprogress;

attribute Function onload;

attribute Function onabort;

attribute Function onerror;

attribute Function onloadend;

};

```

Note FileReader has a constructor, and the spec states that conforming implementations need to provision this constructor via a Window or WorkerGlobalScope global object.

A FileReader has three states – Empty, Loading and Done. It has six states and equivalent handlers (onloadstart, onprogress, onload, onabort, onerror and onloadend). It has four ReadAs functions (readAsArrayBuffer, readAsBinaryString, readAsText and readAsDataURL), each of which reads the blob in various ways.

### **FileReaderSync – A synchronous reader which may be used with web workers**

```

[Constructor]

interface FileReaderSync {

    // Synchronously return strings

    // All methods raise FileException

    ArrayBuffer readAsArrayBuffer(in Blob blob) raises (FileException);

    DOMString readAsBinaryString(in Blob blob) raises (FileException);

    DOMString readAsText(in Blob blob, optional in DOMString encoding) raises (FileException);

    DOMString readAsDataURL(in Blob blob) raises (FileException);

};

```

Web workers can allow blocking of a thread, and so FileReaderSync is a little easier to use than asynchronous (in general, we would still recommend use of FileReader).

**FileError – Provides error information asynchronously for FileReader**

```

interface FileError {
    // File error codes
    // Found in DOMException
    const unsigned short NOT\_FOUND\_ERR = 1;
    const unsigned short SECURITY\_ERR = 2;
    const unsigned short ABORT\_ERR = 3;
    // Added by this specification
    const unsigned short NOT\_READABLE\_ERR = 4;
    const unsigned short ENCODING\_ERR = 5;
    readonly attribute unsigned short code;
    readonly attribute DOMString name name;
};

```

Note FileError is extended in the File Writer API spec and the File System API spec.

**FileException – Provides error information synchronously for FileReaderSync**

```

exception FileException {
    const unsigned short NOT\_FOUND\_ERR = 1;
    const unsigned short SECURITY\_ERR = 2;
    const unsigned short ABORT\_ERR = 3;
    const unsigned short NOT\_READABLE\_ERR = 4;
    const unsigned short ENCODING\_ERR = 5;
    unsigned short code;
    DOMString name name;
};

```

This exception is extended in the File Writer API spec and the File System API spec.

**Working with Blob URI (creates an object URL)**

```

[Supplemental]
interface URL {
    static DOMString createObjectURL(in Blob blob);
    static void revokeObjectURL(in DOMString url);
};

```

## File Writer API Concepts

The File Writer API extended File (Reader) API with writing capabilities.

**BlobBuilder** – Builds up a blob by appending strings, other blobs or (binary) ArrayBuffers

```
[Constructor]
interface BlobBuilder {
    Blob getBlob (optional in DOMString contentType);
    void append (in DOMString text, optional in DOMString endings)
        raises (FileException);
    void append (in Blob data);
    void append (in ArrayBuffer data);
};
```

**FileSaver** – A base interface for saving files and monitoring writing via function handlers

```
[Constructor(in Blob data)]
interface FileSaver {
    void abort () raises (FileException);
    const unsigned short INIT = 0;
    const unsigned short WRITING = 1;
    const unsigned short DONE = 2;
    readonly attribute unsigned short readyState;
    readonly attribute FileError error;
    attribute Function onwritestart;
    attribute Function onprogress;
    attribute Function onwrite;
    attribute Function onabort;
    attribute Function onerror;
    attribute Function onwriteend;
};
```

The file saver can be in one of three states – INIT, WRITING and DONE.

**FileSaverSync** – in planning

**FileWriter** – supports multiple write/seek/truncate actions

```
[NoInterfaceObject]
interface FileWriter : FileSaver {
    readonly attribute unsigned long long position;
    readonly attribute unsigned long long length;
    void write (Blob data) raises (FileException);
    void seek (long long offset) raises (FileException);
    void truncate (unsigned long long size) raises (FileException);
};
```

This is where the actual writing occurs.

**FileWriterSync** – writes the file synchronously

```
[NoInterfaceObject]
interface FileWriterSync {
    readonly attribute unsigned long long position;
    readonly attribute unsigned long long length;
    void write (Blob data) raises (FileException);
    void seek (long long offset) raises (FileException);
    void truncate (unsigned long long size) raises (FileException);
};
```

**FileError – Describes error situations - extends definition from File (Reader) API**

```
interface FileError {
    const unsigned short NOT_FOUND_ERR = 1;
    const unsigned short SECURITY_ERR = 2;
    const unsigned short ABORT_ERR = 3;
    const unsigned short NOT_READABLE_ERR = 4;
    const unsigned short ENCODING_ERR = 5;
    const unsigned short NO_MODIFICATION_ALLOWED_ERR = 6;
    const unsigned short INVALID_STATE_ERR = 7;
    const unsigned short SYNTAX_ERR = 8;
    const unsigned short QUOTA_EXCEEDED_ERR = 10;
    readonly attribute unsigned short code;
};
```

**FileException – exception raised by FileWriterSync when errors occur**

```
exception FileException {
    const unsigned short NOT_FOUND_ERR = 1;
    const unsigned short SECURITY_ERR = 2;
    const unsigned short ABORT_ERR = 3;
    const unsigned short NOT_READABLE_ERR = 4;
    const unsigned short ENCODING_ERR = 5;
    const unsigned short NO_MODIFICATION_ALLOWED_ERR = 6;
    const unsigned short INVALID_STATE_ERR = 7;
    const unsigned short SYNTAX_ERR = 8;
    const unsigned short QUOTA_EXCEEDED_ERR = 10;
    unsigned short code;
};
```

## File API: Directories and System

The File API: Directories and System specification extends File (Reader) API and File Writer API with additional APIs to deal with file systems and directories.

It supports both temporary (does not last beyond browser session) and persistent storage (lasts beyond browser session).

### LocalFileSystem – Provides a way to access a file system (local to the client)

```
[Supplemental, NoInterfaceObject]
interface LocalFileSystem {
  const unsigned short TEMPORARY = 0;
  const unsigned short PERSISTENT = 1;
  void requestFileSystem (unsigned short type, unsigned long long size,
    FileSystemCallback successCallback, optional ErrorCallback errorCallback);
  void resolveLocalFileSystemURL (DOMString url,
    EntryCallback successCallback, optional ErrorCallback errorCallback);
};
```

The LocalFileSystem interface is implemented by Window and WorkerGlobalScope.

LocalFileSystem represents a way to access a file system, but is not the filesystem itself.

### LocalFileSystemSync - Represents an isolated local file system (local to the client)

```
[Supplemental, NoInterfaceObject]
interface LocalFileSystemSync {
  const unsigned short TEMPORARY = 0;
  const unsigned short PERSISTENT = 1;
  FileSystemSync requestFileSystemSync (unsigned short type, unsigned long
long size) raises (FileException);
  EntrySync resolveLocalFileSystemSyncURL (DOMString url) raises
(FileException);
};
```

The LocalFileSystemSync interface is implemented by WorkerGlobalScope.

### Metadata and Flags for access

The modification time:

```
[NoInterfaceObject]
interface Metadata {
  readonly attribute Date modificationTime;
};
```

Flags for working with entries:

```
[NoInterfaceObject]
interface Flags {
  attribute boolean create;
  attribute boolean exclusive;
};
```

**FileSystem - Represents an isolated file system (local to the client)**

```
[NoInterfaceObject]
interface FileSystem {
  readonly attribute DOMString      name;
  readonly attribute DirectoryEntry root;
};
```

The root is the root of the isolated file system for the browser and NOT the entire client operating system (it is not envisaged that a browser could access '/' on Linux or [c:\](#) on Windows). Instead, a securely written browser will allocate a certain amount of disk space, and make available to web applications. Write/navigation access to disk outside of this area will be blocked.

**FileSystemSync – A synchronous version of FileSystem.**

```
[NoInterfaceObject]
interface FileSystemSync {
  readonly attribute DOMString      name;
  readonly attribute DirectoryEntrySync root;
};
```

**Entry – Base interface for a file system contract (this is derived by DirectoryEntry/FileEntry).**

```
[NoInterfaceObject]
interface Entry {
  readonly attribute boolean      isFile;
  readonly attribute boolean      isDirectory;
  void      getMetadata (
    MetadataCallback successCallback,
    optional ErrorCallback errorCallback);
  readonly attribute DOMString     name;
  readonly attribute DOMString     fullPath;
  readonly attribute FileSystem filesystem;
  void      moveTo (DirectoryEntry parent, optional DOMString newName,
    optional EntryCallback successCallback,
    optional ErrorCallback errorCallback);
  void      copyTo (DirectoryEntry parent, optional DOMString newName,
    optional EntryCallback successCallback,
    optional ErrorCallback errorCallback);
  DOMString toURL ();
  void      remove (VoidCallback successCallback,
    optional ErrorCallback errorCallback);
  void      getParent (EntryCallback successCallback,
    optional ErrorCallback errorCallback);
};
```

**EntrySync – a synchronous version of Entry**

```
[NoInterfaceObject]
interface EntrySync {
  readonly attribute boolean      isFile;
  readonly attribute boolean      isDirectory;
  Metadata      getMetadata () raises (FileException);
  readonly attribute DOMString     name;
  readonly attribute DOMString     fullPath;
  readonly attribute FileSystemSync filesystem;
  EntrySync      moveTo (DirectoryEntrySync parent,
    optional DOMString newName) raises (FileException);
  EntrySync      copyTo (DirectoryEntrySync parent,
    optional DOMString newName) raises (FileException);
  DOMString      toURL ();
};
```

```

void remove () raises (FileNotFoundException);
DirectoryEntrySync getParent ();
};

```

### DirectoryEntry – Represents a directory in the file system

```

[NoInterfaceObject]
interface DirectoryEntry : Entry {
    DirectoryReader createReader ();
    void getFile (DOMString path, optional Flags options,
                optional EntryCallback successCallback,
                optional ErrorCallback errorCallback);
    void getDirectory (DOMString path, optional Flags options,
                    optional EntryCallback successCallback,
                    optional ErrorCallback errorCallback);
    void removeRecursively (VoidCallback successCallback,
                          optional ErrorCallback errorCallback);
};

```

The root directory is accessible from FileSystem, and using DirectoryEntry, one can navigate to contained sub-directories and files.

### DirectoryEntrySync – A synchronous version of DirectoryEntry.

```

[NoInterfaceObject]
interface DirectoryEntrySync : EntrySync {
    DirectoryReaderSync createReader () raises (FileNotFoundException);
    FileEntrySync getFile (DOMString path, optional Flags options) raises
    (FileNotFoundException);
    DirectoryEntrySync getDirectory (DOMString path, optional Flags options)
    raises (FileNotFoundException);
    void removeRecursively () raises (FileNotFoundException);
};

```

### DirectoryReader – Used to read entries in a directory.

```

[NoInterfaceObject]
interface DirectoryReader {
    void readEntries (
        EntriesCallback successCallback,
        optional ErrorCallback errorCallback);
};

```

Does not include the directory itself or its parent.

### DirectoryReaderSync – A synchronous version of DirectoryReader.

```

[NoInterfaceObject]
interface DirectoryReaderSync {
    EntrySync[] readEntries () raises (FileNotFoundException);
};

```

### FileEntry – Represents a file entry

```

[NoInterfaceObject]
interface FileEntry : Entry {
    void createWriter (FileWriterCallback successCallback,
                    optional ErrorCallback errorCallback);
    void file (FileCallback successCallback,
             optional ErrorCallback errorCallback);
};

```

**FileEntrySync – Synchronous version of FileEntry.**

```
[NoInterfaceObject]
interface FileEntrySync : EntrySync {
    FileWriterSync createWriter () raises (FileException);
    File file () raises (FileException);
};
```

**Callbacks – handlers that script writers implement for specific actions**

```
[NoInterfaceObject, Callback=FunctionOnly]
interface FileSystemCallback {
    void handleEvent (FileSystem filesystem);
};
```

```
[NoInterfaceObject, Callback=FunctionOnly]
interface EntryCallback {
    void handleEvent (Entry entry);
};
```

```
[NoInterfaceObject, Callback=FunctionOnly]
interface EntriesCallback {
    void handleEvent (Entry[] entries);
};
```

```
[NoInterfaceObject, Callback=FunctionOnly]
interface MetadataCallback {
    void handleEvent (Metadata metadata);
};
```

```
[NoInterfaceObject, Callback=FunctionOnly]
interface FileWriterCallback {
    void handleEvent (FileWriter fileWriter);
};
```

```
[NoInterfaceObject, Callback=FunctionOnly]
interface FileCallback {
    void handleEvent (File file);
};
```

```
[NoInterfaceObject, Callback=FunctionOnly]
interface VoidCallback {
    void handleEvent ();
};
```

```
[NoInterfaceObject, Callback=FunctionOnly]
interface errorCallback {
    void handleEvent (FileError err);
};
```

**FileError – extended with directory&system specific errors**

```
interface FileError {
    const unsigned short NOT_FOUND_ERR = 1;
    const unsigned short SECURITY_ERR = 2;
    const unsigned short ABORT_ERR = 3;
    const unsigned short NOT_READABLE_ERR = 4;
    const unsigned short ENCODING_ERR = 5;
    const unsigned short NO_MODIFICATION_ALLOWED_ERR = 6;
    const unsigned short INVALID_STATE_ERR = 7;
```

```
const unsigned short SYNTAX_ERR = 8;
const unsigned short INVALID_MODIFICATION_ERR = 9;
const unsigned short QUOTA_EXCEEDED_ERR = 10;
const unsigned short TYPE_MISMATCH_ERR = 11;
const unsigned short PATH_EXISTS_ERR = 12;
attribute unsigned short code;
};
```

### **FileException – extended with didirectory&system specific errors**

```
exception FileException {
    const unsigned short NOT_FOUND_ERR = 1;
    const unsigned short SECURITY_ERR = 2;
    const unsigned short ABORT_ERR = 3;
    const unsigned short NOT_READABLE_ERR = 4;
    const unsigned short ENCODING_ERR = 5;
    const unsigned short NO_MODIFICATION_ALLOWED_ERR = 6;
    const unsigned short INVALID_STATE_ERR = 7;
    const unsigned short SYNTAX_ERR = 8;
    const unsigned short INVALID_MODIFICATION_ERR = 9;
    const unsigned short QUOTA_EXCEEDED_ERR = 10;
    const unsigned short TYPE_MISMATCH_ERR = 11;
    const unsigned short PATH_EXISTS_ERR = 12;
    readonly unsigned short code;
};
```

## **Security & Privacy**

To use File API from some browsers with [file://](#) protocol (e.g. accessing HTML files stores locally on your disk), for some browsers (e.g. Chrome) you need to add to the command line: --allow-file-access-from-files.