Filesystem API: FileReader API, FileWriter API

С помощью FileSystem API любое веб-приложение может создавать, считывать, просматривать и записывать данные в тестовой среде локальной файловой системы пользователя.

***В API можно выделить несколько аспектов:***

- Чтение файлов и операции с ними: File/Blob, FileList, FileReader.

- Создание и запись: BlobBuilder, FileWriter.

- Доступ к каталогам и файловым системам: DirectoryReader, FileEntry/DirectoryEntry, LocalFileSystem.

В спецификации указаны некоторые интерфейсы для доступа к файлам в локальной файловой системе.

***File*** – отдельный файл. Такая информация о нем, как название, размер файла, тип MIME и ссылка на обработчик, доступны только для чтения.

***FileList*** – последовательность объектов File в виде массива, позволяющая, например, реализовать функцию <input type="file" multiple> или перетаскивание папки с файлами с рабочего стола.

***Blob***– позволяет разделить файл на фрагменты заданной величины.

File создается на основе Blob, а значит, оба файловых параметра являются blob-объектами.

***Обращение к файловой системе***

Веб-приложение может запрашивать доступ к файловой системе тестовой среды с помощью метода 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*** *-* Доступность хранилища файлов.

Возможные значения: ***window.TEMPORARY*** и ***window.PERSISTENT***. Данные, которые сохраняются с ключом ***TEMPORARY***, могут быть удалены на усмотрение браузера (например, при нехватке места).

Ключ ***PERSISTENT*** не позволяет удалять данные без разрешения пользователя или приложения, и для него необходима квота, предоставляемая приложению пользователем.

***size -*** Размер необходимого приложению хранилища (в байтах).

***successCallback -*** Функция обратного вызова, которая вызывается при успешном запросе файловой системы. В качестве аргумента выступает объект FileSystem.

***opt\_errorCallback*** - Необязательный вызов для обработки ошибок или при отказе в ответ на запрос файловой системы. В качестве аргумента выступает объект FileError.

Если функция ***requestFileSystem()*** вызывается в первый раз, для приложения создается новое хранилище. Никакое веб-приложение не может получить доступ к данным другой программы. Это также означает невозможность считывать и записывать файлы в произвольную папку на жестком диске пользователя (например, "Мои изображения", "Мои документы" и т. д.).

function onInitFs(fs) {

console.log('Opened file system: ' + fs.name);

}

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);

}

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

***Запрос квоты хранилища***

Для использования постоянного хранилища (ключ PERSISTENT) необходимо получить разрешение пользователя на хранение постоянных данных.

На временное хранилище (ключ TEMPORARY) это правило не распространяется, поскольку браузер может удалять такую информацию по собственному усмотрению.

Для использования постоянного хранилища (ключ PERSISTENT) с FileSystem API браузер Chrome запрашивает хранилище, указывая новый API в атрибуте window.webkitStorageInfo:

window.webkitStorageInfo.requestQuota (PERSISTENT, 1024\*1024, function(grantedBytes) {

window.requestFileSystem(PERSISTENT, grantedBytes, onInitFs, errorHandler);

}, function(e) {

console.log('Error', e);

});

Как только пользователь дает разрешение, необходимость в вызове функции requestQuota() отпадает. Все последующие вызовы являются пустыми операциями.

Существует также API для запроса ***текущей степени использования и распределения исходного хранилища***:

window.webkitStorageInfo.queryUsageAndQuota()

***Работа с файлами***

Для работы с файлами в тестовой среде предназначен интерфейс ***FileEntry.***

***Свойства и методы 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);

***Выбор файлов***

Сначала проверьте, поддерживает ли ваш браузер API файлов.

if (window.File && window.FileReader && window.FileList && window.Blob) {

// Great success! All the File APIs are supported.

} else {

alert('The File APIs are not fully supported in this browser.');

}

***Создание файла***

Для ***поиска или создания*** файла можно воспользоваться методом ***getFile()*** интерфейса ***DirectoryEntry*** файловой системы.

После запроса файловой системы обратный вызов возвращает объект ***FileSystem***, содержащий объект ***DirectoryEntry (fs.root)*** с указанием на ***корневую папку*** файловой системы приложения.

***Приведенный ниже код создает в корневой папке файловой системы приложения пустой файл log.txt.***

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);

fs - объект ***FileSystem,***

fs.root - объект ***DirectoryEntry*** с указанием на ***корневую папку*** файловой системы приложения.

После запроса файловой системы обработчику передается объект FileSystem. В пределах обратного вызова можно вызвать функцию fs.root.getFile() с именем файла, который нужно создать. Путь может быть абсолютным или относительным (но он должен быть действительным).

Например, не пытайтесь создать файл в несуществующей родительской папке.

Вторым аргументом функции getFile() является объект, описывающий поведение функции в случае, если файл не существует.

В нашем примере атрибут ***create: true создает файл, если он не существует***, и выдает ошибку в противном случае ***(exclusive: true).***

Атрибут ***create: false*** приводит только к поиску и выдаче файла.

Содержание файла не перезаписывается ни при каких обстоятельствах, поскольку мы получаем только ссылку на соответствующий файл.

Как только завершается загрузка файла, на стороне пользователя вызывается событие onload. Его атрибут result можно использовать для доступа к данным в файле.

Существует ***четыре метода асинхронного чтения файлов*** с помощью объекта FileReader:

***FileReader.readAsBinaryString(Blob|File)*** – свойство ***result*** содержит данные о файле или его фрагменте в виде строки бинарного кода. Каждый байт представлен целым числом от 0 до 255.

***FileReader.readAsText(Blob|File, opt\_encoding)*** – свойство ***result*** содержит данные о файле или его фрагменте в виде текстовой строки. По умолчанию используется кодировка UTF-8. Чтобы задать другой формат, используйте необязательный параметр кодировки opt\_encoding.

***FileReader.readAsDataURL(Blob|File)*** – свойство ***result*** содержит данные о файле или его фрагменте в виде схемы *data:URL*.

***FileReader.readAsArrayBuffer(Blob|File)*** – свойство ***result*** содержит данные о файле или его фрагменте в виде объекта ArrayBuffer.

Если для объекта FileReader вызывается один из этих методов, то ***ход его обработки*** можно отслеживать с помощью атрибутов

***- onloadstart***

***- onprogress***

***- onload***

***-  onabort***

***-  onerror***

***-  onloadend.***

***READASTEXT()***

Метод *readAsText()* может использоваться для чтения текстовых файлов. Он принимает 2 параметра:

Первый - *File* или *Blob* объект, который следует прочитать

Второй параметр – опциональный, в нем можно указывать кодировку, по умолчанию UTF-8.

Данный метод является асинхронным.

var reader = new FileReader();

reader.onload = function(e) {

var text = reader.result;

}

reader.readAsText(file, encoding);

***READASDATAURL()***

Метод *readAsDataURL()* в качестве параметра принимает *File* или *Blob* и возвращает *data URL* – строка кодировки base64.

*data URL* можно использовать для того чтоб проставлять атрибут изображения src.

var reader = new FileReader();

reader.onload = function(e) {

var dataURL = reader.result;

}

reader.readAsDataURL(file);

***READASBINARYSTRING()***

*readAsBinaryString()* может использоваться для чтения файла любого типа.

Возвращает raw binary data from the file.

var reader = new FileReader();

reader.onload = function(e) {

var rawData = reader.result;

}

reader.readAsBinaryString(file);

***READASARRAYBUFFER()***

readAsArrayBuffer() в качестве параметра принимает *Blob* или *File* и возвращает *ArrayBuffer*.

*ArrayBuffer* is a fixed-length binary data buffer.

var reader = new FileReader();

reader.onload = function(e) {

var arrayBuffer = reader.result;

}

reader.readAsArrayBuffer(file);

***ABORT()***

Метод abort() – прерывает операцию чтения.

reader.abort();

***Чтение файла по названию***

Приведенный ниже код извлекает файл под названием log.txt, читает его содержание с помощью FileReader API и записывает в новый блок <textarea> на странице. Если файл log.txt не существует, выдается ошибка.

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);

***Разделение файла на фрагменты***

В некоторых случаях считывание всего файла в память не является оптимальным вариантом.

Чтобы увеличить скорость загрузки файла, можно разделить его на фрагменты одинаковой величины, которые будут считываться и отправляться по отдельности. На сервере фрагменты будут снова объединяться в нужном порядке.

Интерфейс File поддерживает метод slice для разбиения файла на фрагменты. Первым аргументом этого метода является *начальный байт файла*, вторым – *его последний байт*, а третьим – необязательная строка с описанием *типа содержания*. Семантика этого метода недавно была изменена, и теперь необходимо использовать префикс браузера:

if (file.webkitSlice) {

var blob = file.webkitSlice(startingByte, endindByte);

} else if (file.mozSlice) {

var blob = file.mozSlice(startingByte, endindByte);

}

reader.readAsBinaryString(blob);

***Контроль хода чтения***

Одним из преимуществ асинхронной обработки событий является возможность отслеживать ход чтения: это особенно удобно при работе с большими файлами, для контроля ошибок и для определения момента завершения чтения.

Ход чтения можно отслеживать с помощью событий *onloadstart* и *onprogress*.

***Запись в файл***

Приведенный ниже код создает пустой файл под названием log.txt (если он не существует) и заполняет его текстом 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.');

};

fileWriter.onerror = function(e) {

console.log('Write failed: ' + e.toString());

};

// Create a new Blob and write it to log.txt.

var bb = new BlobBuilder(); // Note: window.WebKitBlobBuilder in Chrome 12.

bb.append('Lorem Ipsum');

fileWriter.write(bb.getBlob('text/plain'));

}, errorHandler);

}, errorHandler);}

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

На этот раз мы вызываем метод createWriter() объекта FileEntry и получаем элемент FileWriter. В обратном вызове настроены обработчики событий error и writeend. Для записи текстовых данных в файл создается объект blob, в него добавляется текст, после чего он передается функции FileWriter.write().

***Прикрепление данных к файлу***

Приведенный ниже код добавляет в конец файла log.txt текст Hello World. Если файл не существует, выдается ошибка.

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 bb = new BlobBuilder(); // Note: window.WebKitBlobBuilder in Chrome 12.

bb.append('Hello World');

fileWriter.write(bb.getBlob('text/plain'));

}, errorHandler);

}, errorHandler);

}

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

***Создание копий выбранных файлов***

Приведенный ниже код позволяет пользователю выбрать несколько файлов с помощью атрибута <input type="file" multiple /> и создать их копии в тестовой файловой системе приложения.

<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(file.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);

};

***Удаление файла***

Приведенный ниже код удаляет файл log.txt.

window.requestFileSystem(window.TEMPORARY, 1024\*1024, function(fs) {

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

fileEntry.remove(function() {

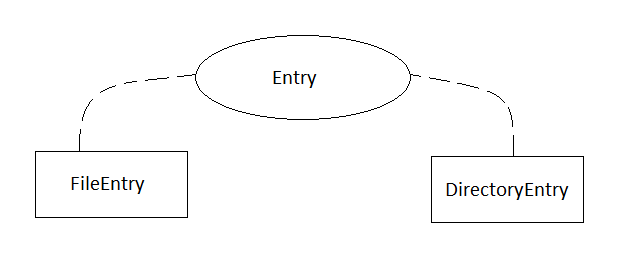
console.log('File removed.');

}, errorHandler);

}, errorHandler);

}, errorHandler);

***Работа с каталогами***



Для работы с каталогами в тестовой среде предназначен интерфейс *DirectoryEntry*, который обладает большинством свойств *FileEntry* (они наследуются из общего интерфейса Entry).

При этом *Directory Entry* поддерживает дополнительные методы для работы с каталогами.

***Свойства и методы DirectoryEntry***

dirEntry.isDirectory === true

// See the section on [FileEntry](http://www.html5rocks.com/ru/tutorials/file/filesystem/#toc-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);

...

***Примеры использования***

В HTML5 реализовано несколько вариантов хранения, но API FileSystem отличается тем, что позволяет решать проблемы использования хранилищ со стороны клиента, с которыми не справляются базы данных. Чаще всего это приложения, которые работают с большими объемами двоичных blob-объектов или обмениваются данными с программами за пределами браузера.

В спецификации приведено несколько примеров использования.

*Инструмент для постоянной загрузки.* Когда для загрузки выбирается файл или каталог, файлы копируются в локальную тестовую среду и загружаются по частям. В случае сбоя браузера, сети и т. д. загрузку можно возобновить.

*Офлайн-программа для просмотра видео*

Загружает большие файлы (больше 1 ГБ) для их последующего просмотра.

*Автономный почтовый веб-клиент*

Загружает прикрепленные файлы и хранит их локально.

Кэширует выбранные пользователем прикрепленные файлы для их последующей загрузки.

Links

<http://www.html5rocks.com/ru/tutorials/file/dndfiles/>

http://www.html5rocks.com/ru/tutorials/file/filesystem/

http://blog.teamtreehouse.com/reading-files-using-the-html5-filereader-api