|  |
| --- |
| EPAM Systems |
| FileSystem API&File API |
| [Document subtitle] |

|  |
| --- |
| Oksana Spolnyk  [Date] |

# Определение

С помощью FileSystem API и File API веб приложение может создавать, читать, просматривать и записывать файлы находящиеся в области пользовательской «песочницы».

Описание API разбито на следующие секции:

* Чтение и управление файлами: File/Blob, FileList, FileReader
* Создание и запись: BlobBuilder, FileWriter
* Работа с директориями и права доступа: DirectoryReader, FileEntry/DirectoryEntry, LocalFileSystem

Объекты, с которыми предстоит работать:

* **File** — собственно файл; позволяет получить такую доступную только для чтения информацию, как имя, размер, mimetype и прочее.
* **FileList** — «массив» объектов File.
* **Blob** — сущность, позволяющая разбирать файл по байтам.

Поддержка браузерами и ограничение на хранение

На момент написания статьи только Google Chrome 9+ имеет рабочую реализацию FileSystem API. И на данный момент пока нет никаких диалоговых окон для управления файлами и квотами на хранилище, поэтому нужно будет использовать флаг --unlimited-quota-for-files (в случае разработки приложений для Chrome Web Store будет достаточно манифеста с разрешением unlimitedStorage). Но все меняется и пользователи в скором времени получат возможность для управления правами по работе с файлами, которые будут требоваться приложению.

Вам может потребоваться использование флага --allow-file-access-from-files , если вы дебажите приложение с использованием file://. Если этот флаг не использовать, то будут выброшены исключения типа SECURITY\_ERR или QUOTA\_EXCEEDED\_ERR.

## Обращаемся к файловой системе

Веб приложение может обратиться к файловой системе вызвав следующий метод window.requestFileSystem():

window.requestFileSystem(type, size, successCallback, opt\_errorCallback)

type - правила хранения, доступные значения window.TEMPORARY и window.PERSISTENT. данные, хранящиеся с использованием ключа TEMPORARY могут быть удаление по усмотрению браузером (например, если не хватает места). Если же выставлен ключ PERSISTENT, то данные могут быть очищены только после действий пользователя или приложения.

size - Размер (в байтах) хранилища, которое потребуется приложению.

successCallback - Callback-функция, выполняемая в случае успешного обращения к файловой системе. Ее аргументом является объект типа FileSystem.

opt\_errorCallback - Необязательная callback-функция для обработки ошибок. Так же вызывается, когда возникают ошибки обращения к файловой системе. Параметром является объект типа FileError.

Если вы вызываете метод requestFileSystem() в рамках вашего приложения в первый раз, то в этот момент и будет создано хранилище. Очень важно помнить, что данное хранилище является закрытым и другое приложение не будет иметь к нему доступа. Это так же значит, что приложение не может менять прочие файлы и папки, расположенные на жестком диске.

**function** onInitFs(fs) {  
 console.log('Opened file system: ' + fs.name);  
}  
window.requestFileSystem(window.PERSISTENT, 5\*1024\*1024 */\*5MB\*/*, onInitFs, errorHandler);

Возвращаясь к методу requestFileSystem() стоит описать возможные варианты возникающих ошибок:

**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);  
}

# Работа с файлам

Для работы с файлам предусмотрен интерфейс FileEntry. Он обладает рядом методов и свойств, которые мы привыкли ассоциировать с обычными файлами.

## Создание файла

Получить или создать файл можно с помощью метода getFile() у интерфейса DirectoryEntry. После обращения к хранилищу, callback возвращает нам объект FileSystem, содержащий в себе DirectoryEntry (fs.root), ссылающийся на корневую папку хранилища.

**function** onInitFs(fs) {  
  
 fs.root.getFile('log.txt', {create: **true**, exclusive: **true**}, **function**(fileEntry) {  
 // fileEntry будет иметь следующие свойства  
 // fileEntry.isFile === true  
 // fileEntry.name == 'log.txt'  
 // fileEntry.fullPath == '/log.txt'  
  
 }, errorHandler);  
  
}  
window.requestFileSystem(window.PERSISTENT, 1024\*1024, onInitFs, errorHandler);

## Чтение файла по имени

Следующий код обращается к файлу «log.txt» и читает его содержимое с помощью FileReader API, после чего записывает все содержимое в блок <textarea>. Если файл не существует, то будет выброшена ошибка.

**function** onInitFs(fs) {  
  
 fs.root.getFile('log.txt', {}, **function**(fileEntry) {  
  
 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.PERSISTENT, 1024\*1024, onInitFs, errorHandler);

FileReader помимо прочего предоставляет следующие методы для чтения:

* FileReader.readAsBinaryString(Blob|File) — результат будет содержать байтовую строку.
* FileReader.readAsText(Blob|File, opt\_encoding) — результат будет содержать текстовую строку. Кодировка по умолчанию — 'UTF-8', менять можно с помощью задания опционального параметра
* FileReader.readAsDataURL(Blob|File) — на выходе имеем data URL.
* FileReader.readAsArrayBuffer(Blob|File) — получаем данные в виде ArrayBuffer.

FileReader предоставляет нам следующие типы событий:

* onloadstart
* onprogress
* onload,
* onabort
* onerror
* onloadend

## Запись в файл

С помощью следующего кода мы создадим файл «log.txt» (если он не существует) и запишем в него 'Ipsum Lorem'.

**function** onInitFs(fs) {  
 fs.root.getFile('log.txt', {create: **true**}, **function**(fileEntry) {  
 fileEntry.createWriter(**function**(fileWriter) {  
  
 fileWriter.onwriteend = **function**(e) {  
 console.log('Write completed.');  
 };  
 fileWriter.onerror = **function**(e) {  
 console.log('Write failed: ' + e.toString());  
 };  
  
 **var** bb = **new** BlobBuilder();  
 bb.append('Ipsum Lorem');  
 fileWriter.write(bb.getBlob('text/plain'));  
  
 }, errorHandler);  
 }, errorHandler);  
}  
window.requestFileSystem(window.PERSISTENT, 1024\*1024, onInitFs, errorHandler);

## Дописываем данные в файл

**function** onInitFs(fs) {  
 fs.root.getFile('log.txt', {create: **false**}, **function**(fileEntry) {  
  
 fileEntry.createWriter(**function**(fileWriter) {  
  
 fileWriter.seek(fileWriter.length); // Start write position at EOF  
 **var** bb = **new** BlobBuilder();  
 bb.append('Hello World');  
 fileWriter.write(bb.getBlob('text/plain'));

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

## Удаление файлов

window.requestFileSystem(window.PERSISTENT, 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 директориями

Работа с директориями осуществляется за счет использования DirectoryEntry, который обладает большинством свойств FileEntry ( они оба наследуют интерфейс Entry).

## Создание директорий

Для создания и обращения к директориям используется getDirectory() интерфейса DirectoryEntry. Можно передавать как имя, так и путь до директории.

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

## Поддиректории

**var** path = 'music/genres/jazz/';  
  
**function** createDir(rootDirEntry, folders) {  
 //Фильтруем './' и '/'   
 **if** (folders[0] == '.' || folders[0] == '') {  
 folders = folders.slice(1);  
 }  
 rootDirEntry.getDirectory(folders[0], {create: **true**}, **function**(dirEntry) {  
 **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.PERSISTENT, 1024\*1024, onInitFs, errorHandler);

# 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.

1. Инструмент для постоянной загрузки.
   1. Когда для загрузки выбирается файл или каталог, файлы копируются в локальную тестовую среду и загружаются по частям.
   2. В случае сбоя браузера, сети и т. д. загрузку можно возобновить.
2. Видеоигры, музыка и другие приложения с большим объемом мультимедийных данных
   1. Инструмент загружает один или несколько больших архивов и локально разворачивает их в структуру каталогов.
   2. Общий способ загрузки должен работать в любой операционной системе.
   3. Данные, которые вскоре понадобятся, могут выбираться в фоновом режиме, чтобы их загрузка при переходе на следующий уровень игры или при активации новой функции не занимала слишком много времени.
   4. Загрузчик выбирает элементы непосредственно из локального кэша путем прямого считывания файлов или обработки локальных URL изображений или видеотегов, с помощью программ для загрузки объектов WebGL и т. д.
   5. Файлы могут иметь любой двоичный формат.
   6. Сжатый архив на сервере обычно намного меньше, чем набор файлов, сжатых по отдельности. Кроме того, один архив вместо 1000 небольших файлов требует меньше обращений. В остальном никакой разницы нет.
3. Аудио- или фоторедактор с автономным доступом или локальным кэшем для скорости
   1. Blob-объекты могут быть довольно большими и использоваться для чтения и записи.
   2. Файлы могут перезаписываться частично (например, только теги ID3/EXIF).
   3. Полезной является возможность систематизации файлов проекта с помощью структуры каталогов.
   4. У клиентских приложений [iTunes, Picasa] должен быть доступ к измененным файлам.
4. Офлайн-программа для просмотра видео
   1. Загружает большие файлы (больше 1 ГБ) для их последующего просмотра.
   2. Требует эффективного поиска и потоковой передачи.
   3. Должна уметь передавать URL в видеотеги.
   4. Требует доступа к частично загруженным файлам (например, чтобы можно было смотреть первый эпизод DVD-фильма, не дожидаясь полной загрузки).
   5. Должна уметь извлекать один эпизод из загруженных данных и передавать его непосредственно в видеотег.
5. Автономный почтовый веб-клиент
   1. Загружает прикрепленные файлы и хранит их локально.
   2. Кэширует выбранные пользователем прикрепленные файлы для их последующей загрузки.
   3. Должен уметь обращаться к кэшированным прикрепленным файлам и миниатюрам изображений для отображения и загрузки.
   4. Должен уметь вызывать менеджер загрузок UA так же, как при обращении к серверу.
   5. Должен уметь загружать электронные письма с прикрепленными файлами как сообщения из нескольких частей, а не отправлять сразу весь файл с помощью XHR-запроса.

# Источники

<http://habrahabr.ru/post/112286/>

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