**HTML приложения работа с файлами папками и дисками (ч.1)**

06.05.2011 12:06 | [dronov\_va](http://www.thevista.ru/users.php?m=details&id=107577)

Просмотров 10379 Комментарии 0

[-](javascript:%20void(0)) [T](javascript:%20void(0)) [+](javascript:%20void(0))

В предыдущей статье цикла, посвящённого HTML-приложениям Internet Explorer (HTA), мы рассматривали возможности файлового ввода-вывода и учились читать текстовую информацию из файлов и записывать её в них. Теперь мы можем наделить HTML-приложения возможностью работы с документами, хранящимися в файлах.  
  
Однако возможности WSH по работе с файлами этим не исчерпываются. Различные объекты, реализуемые этой программой, позволяют нам также:

* получать сведения о файлах (имя, размер, даты создания, последнего изменения и последнего открытия и др.);
* получать сведения о папках (имя, размер, даты создания, последнего изменения и последнего открытия, является ли папка корневой и др.);
* получать сведения о дисках (тип, общий объём, объём свободного пространства и др.);
* создавать, копировать, перемещать, переименовывать и удалять файлы и папки;
* выполнять вспомогательные задачи (получать путь к папке, в которую вложена данная папка, получать уникальное имя временного файла и др.).

Данная статья целиком посвящена рассмотрению всех перечисленных выше возможностей WSH и соответствующих им объектов.  
  
**Примечание:**  
Перед чтением этой статьи настоятельно рекомендуется ознакомиться с предыдущими статьями цикла, посвящённого HTML-приложениям Internet Explorer.  
  
Если средства файлового ввода-ввода, рассмотренные нами в предыдущей статье цикла, поддерживали работу только с текстовыми файлами, то инструменты, что мы рассмотрим в этой статье, позволяют работать с файлами любого формата.  
  
  
**1. Получение доступа к средствам файлового ввода-вывода WSH**  
Как получить доступ к средствами для работы с файловой системой WSH, описывалось в предыдущей статье цикла. Для этого достаточно создать экземпляр объекта Scripting.FileSystemObject - ключевого объекта WSH. Поскольку этот объект реализуется компонентом ActiveX (то есть является внешним объектом), для его создания используется конструктор ActiveXObject, а в качестве единственного его параметра указывается строка с именем объекта, экземпляр которого нужно создать.  
  
var oFSO = new ActiveXObject("Scripting.FileSystemObject");  
  
  
**2. Работа с файлами**  
Начнём с рассмотрения принципов работы с файлами, так как именно с этими объектами файловой системы мы будем чаще всего иметь дело.  
  
**2.1. Получение нужного файла**  
Первое, что нам нужно сделать, - получить файл, с которым мы собираемся работать. Забегая вперёд, скажем, что файл в WSH представляется экземпляром особого объекта; с помощью его свойств мы сможем получить сведения о нём, а с помощью методов - управлять им.  
  
Для получения файла, так сказать, напрямую мы используем метод GetFile объекта FileSystemObject. Вот формат его вызова:  
  
<экземпляр объекта FileSystemObject>.GetFile(  
<путь к файлу>  
);  
  
Первым параметром этому методу передаётся путь к открываемому файлу. Он должен быть указан в виде строки.  
  
**Примечание:**  
Если мы зададим путь к несуществующему файлу, то при выполнении веб-сценария получим сообщение об ошибке. Поэтому лучше перед получением файла проверить, действительно ли он существует на диске (об этом - позже).  
  
Кроме того, если мы укажем путь к файлу, расположенному на диске, который в это время ещё не готов к работе (например, привод компакт-дисков, в который не вставлен диск), то также получим ошибку. Поэтому перед доступом к такому диску следует проверить, готов ли он (как это сделать, будет описано далее).  
  
Метод GetFile возвращает экземпляр объекта File, представляющий полученный файл. Да-да, File - тот самый объект, которым представляется файл и который упоминался в начале этого параграфа.  
  
var oF = oFSO.GetFile("c:\\Work\\testdata.txt");  
  
Здесь мы получили в виде экземпляра объекта File файл testdata.txt, расположенный в папке Work диска C:\.  
  
При указании путей к файлам следует иметь в виду следующее. Для разделения имён папок и файла в путях применяется символ обратного слеша (\). Этот же символ применяется для указания специальных символов (*литералов*) в строковых значениях JavaScript; например, литерал \" обозначает символ двойной кавычки, который нельзя просто так вставить в строку. Так вот, чтобы поместить в строку именно символ обратного слеша, следует использовать литерал \\ (что мы и сделали в предыдущем примере). Запомним это на будущее.  
  
**2.2. Получение сведений о файле**  
Получив файл, мы сможем извлечь различные сведения о нём. Это выполняется с помощью набора свойств объекта File, которые мы сейчас рассмотрим. Почти все эти свойства доступны только для чтения; если какие-то свойство доступно также и для записи, мы специально это укажем.  
  
Свойство Name хранит строку с именем файла. Отметим, что это именно имя файла с расширением, а не полный путь. Данное свойство доступно также и для записи, что позволяет нам переименовывать файлы.  
  
var sFileName = oF.Name;  
  
Свойство ShortName возвращает строку с сокращённым до размеров 8.3 именем файла. И в этом случае возвращается именно имя файла с расширением, а не полный путь. Свойство Path возвращает строку с полным путём к данному файлу. Свойство ShortPath возвращает строку с полным путём к данному файлу, имена папок и файла в котором сокращены до размеров 8.3.  
  
var sPath = oF.Path;  
  
Свойство ParentFolder возвращает строку с полным путём к папке, в которой находится файл. При этом путь возвращается без конечного символа обратного слеша.  
  
var sFolder = oF.ParentFolder;  
  
Данное выражение поместит в переменную sFolder строку "c:\\Work" - путь к папке, в которой находится данный файл. Отметим, что этот путь не имеет конечного символа обратного слеша.  
  
Свойство Drive возвращает строку с наименованием диска, на котором хранится файл. Это наименование имеет вид буквы, отведённой системой данному приводу, за которой следует символ двоеточия. Никаких обратных слешей возвращённая строка не содержит.  
  
var sDrive = oF.Drive;  
  
Выполнив это выражение, мы получим в переменной в переменной sDrive окажется строка "c:" - наименование диска, где хранится файл.  
  
Свойство Type возвращает строку с обозначением типа файла, под которым он зарегистрирован в системе. Насколько эти сведения будут нам полезны - непонятно...  
  
var sType = oF.Type;  
  
Это выражение поместит в переменную sType строку "Текстовый документ" - именно под таким обозначением зарегистрирован в системе текстовый тип файлов (расширение txt).  
  
Свойство Size возвращает размер файла в байтах в виде целого числа.  
  
iSize = oF.Size;  
  
Три перечисленных ниже свойства позволяют узнать различные значения даты и времени, связанными с файлом. Все они возвращают экземпляры объекта Date.

* DateCreated - возвращает дату создания файла.
* DateLastModified - возвращает дату последнего изменения файла.
* DateLastAccessed - возвращает дату последнего доступа к файлу.

var dCreated = oF.DateCreated;  
  
Свойство Attributes хранит целое число, обозначающее атрибуты файла. Это целое число представляет собой сумму чисел, каждое из которых указывает на какой-либо установленный атрибут; все эти числа перечислены ниже.

* 0 - ни один из атрибутов не установлен.
* 1 - установлен атрибут "только для чтения".
* 2 - установлен атрибут "скрытый".
* 4 - установлен атрибут "системный".
* 8 - это не файл, а метка диска.
* 16 - это не файл, а папка.
* 32 - установлен атрибут "архивный".
* 1024 - файл представляет собой ярлык или жёсткую ссылку NTFS.
* 2048 - это сжатый файл.

Так, если свойство Attributes хранит число 37, то у файла установлены атрибуты "только для чтения", "системный" и "архивный".  
  
if (oF.Attributes & 4) {  
//Файл является системным  
}  
  
Обратим внимание, как мы проверяем, установлен ли у файла какой-либо атрибут. Мы выполняем логическое умножение (операцию И) значения свойства Attributes и числа, обозначающего нужный нам атрибут в установленном состоянии (см. список выше). Если в результате мы получим 1, значит, атрибут установлен, если 0 - сброшен. (Будучи использованным в качестве условия, число 1 будет преобразовано в значение true, а число 0 - в false.) Так, логически умножив значение свойства Attributes на 4, мы проверим, установлен ли у файла атрибут "системный".  
  
if (oF.Attributes & 1024) {  
//Файл является ярлыком или жёсткой ссылкой NTFS  
}  
  
Свойство Attributes доступно не только для чтения, но и для записи. Записав в него новое значение, мы можем изменить атрибуты файла.  
  
if (oF.Attributes & 2)  
oF.Attributes = oF.Attributes - 2;  
  
Это выражение проверяет, установлен ли у файла атрибут "скрытый", и, если установлен, сбрасывает его.  
  
Поскольку значение свойства Attributes представляет собой сумму чисел, указывающих на установленные атрибуты (см. список выше), то, чтобы сбросить нужный нам атрибут, нам достаточно вычесть из значения этого свойства число, обозначающее данный атрибут в установленном состоянии. Так, в предыдущем примере, чтобы сбросить у файла атрибут "скрытый", мы вычли из значения свойства Attributes число 2. Аналогично, чтобы установить какой-либо атрибут, мы прибавим обозначающее его число к значению свойства Attributes.  
  
if (!(oF.Attributes & 32))  
oF.Attributes = oF.Attributes + 32;  
  
Это выражение проверяет, сброшен ли у файла атрибут "архивный", и, если сброшен, устанавливает его.  
  
**Примечание:**  
Указанным выше способом мы сможем установить или сбросить только атрибуты "только для чтения", "скрытый", "системный" и "архивный". Превратить файл в папку или сжатый файл в несжатый мы не сможем - при попытке сделать это возникнет сообщение об ошибке.  
  
Кроме того, мы можем получить версию исполняемого файла. Для этого предназначен метод GetFileVersion объекта FileSystemObject. В качестве единственного параметра он принимает строку с полным путём к файлу и возвращает строку с номером его версии или пустую строку, если файл не содержит информации о его версии (например, если это не исполняемый файл).  
  
var sVersion = oFSO.GetFileVersion("c:\\Work\\testfile.exe");  
  
Получаем в переменной sVersion номер версии файла testfile.exe.  
  
var sVersion = oFSO.GetFileVersion("c:\\Work\\testfile.txt");  
  
Получаем в переменной sVersion пустую строку, так как текстовые файлы не содержат информации о версии.  
  
var sVersion = oFSO.GetFileVersion(oF.Path);  
  
Чтобы не указывать путь к полученному ранее файлу дважды, мы можем извлечь его из свойства Path. (Впрочем, файл oF - текстовый, а текстовые файлы не содержат версии... Но ведь это только пример.)  
  
**2.3. Копирование, перемещение, переименование и удаление полученного ранее файла**  
Теперь рассмотрим, как выполняется копирование, перемещение, переименование и удаление файла, полученного ранее с помощью метода GetFile объекта FileSystemObject.  
  
Для копирования файла применяется метод Copy объекта File. Формат его вызова таков:  
  
<файл>.Copy(  
<путь к файлу назначения>[,  
<перезаписывать ли файл, если он уже существует>]  
);  
  
Первый параметр задаёт полный путь к файлу назначения, который будет получен в результате копирования, в виде строки. Указать просто путь к папке назначения мы в этом случае не можем.  
  
**Примечание:**  
Если мы зададим путь к файлу назначения, содержащий несуществующие папки, то при выполнении Web-сценария получим сообщение об ошибке.  
  
Второй, необязательный, параметр позволяет указать, что веб-сценарий будет делать, если файл назначения, путь которого мы указали первым параметром, уже существует на диске. Значение true указывает перезаписать такой файл, а значение false - не перезаписывать. Значение по умолчанию - true (то есть файл будет перезаписан).  
  
**Примечание:**  
Если файл назначения уже существует, и мы указали, чтобы он не перезаписывался, то при выполнении веб-сценария возникнет ошибка.  
  
Метод Copy не возвращает результата.  
  
oF.Copy("c:\\Work\\Temp\\testfile.txt");  
  
Копируем полученный ранее файл testfile.txt в папку Work\Temp диска C.  
  
oF.Copy("c:\\Work\\Temp\\" + oF.Name);  
  
Чтобы в пути к файлу назначения самим не указывать собственно имя файла, мы можем поступить так - взять это имя из свойства Name.  
  
oF.Copy("c:\\Work\\testfile2.txt");  
  
Создаём копию файла testfile.txt под именем testfile2.txt в той же папке Work.  
  
Для перемещения файла в другую папку используется метод Move объекта File. Единственный параметр этого метода задаёт путь к файлу назначения, который будет получен в результате перемещения, в виде строки. Результата этот метод не возвращает.  
  
**Примечание:**  
Если мы зададим путь к файлу назначения, содержащий несуществующие папки, то при выполнении веб-сценария получим сообщение об ошибке. То же самое случится, если файл назначения уже существует.  
  
oF.Move("c:\\Work\\Temp\\testfile.txt");  
  
Перемещаем полученный ранее файл testfile.txt в папку Work\Temp диска C.  
  
oF.Move("c:\\Work\\testfile2.txt");  
  
Перемещаем файл testfile.txt в ту же папку Work, но под именем testfile2.txt. Фактически мы выполняем переименование данного файла.  
  
Хотя переименование удобнее выполнять, присвоив новое имя файла уже знакомому нам свойству Name объекта File (см. параграф 2.2).  
  
**Примечание:**  
Если мы зададим имя файла, который уже существует, то при выполнении веб-сценария получим сообщение об ошибке.  
  
oF.Name = "testfile2.txt";  
  
Для удаления файла применяется метод Delete объекта File. Формат его вызова таков:  
  
<файл>.Delete(  
[<удалить ли файл с установленным атрибутом "только для чтения">]  
);  
  
Единственный, да и тот необязательный, параметр указывает, что делать в случае, если удаляемый файл имеет установленный атрибут "только для чтения". Значение true указывает, что такой файл всё равно следует удалить, а значение false - не делать этого. Значение по умолчанию - false (файлы с установленным атрибутом "только для чтения" удаляться не будут).  
  
Метод Delete не возвращает результата.  
  
**Примечание:**  
При вызове метода Delete файл будет удалён безвозвратно, без помещения в системную Корзину. При попытке удаления несуществующего файла возникнет ошибка.  
  
oF.Delete();  
  
**2.4. Открытие полученного ранее файла**  
Ещё мы можем открыть полученный ранее вызовом метода GetFile объекта FileSystemObject файл, чтобы прочитать его содержимое или записать в него новое. При этом файл обязательно должен быть текстовым - работа с двоичными файлами не поддерживается - об этом уже говорилось в предыдущей статье цикла.  
  
Открыть файл можно вызовом метода OpenAsTextStream объекта File. Вот формат его вызова:  
  
<файл>.OpenAsTextStream(  
<режим открытия>[,  
<кодировка файла>]  
);  
  
Первым параметром указывается режим открытия файла. Значение 1 указывает открыть файл для чтения, значение 2 - для записи, а значение 8 - для дозаписи.  
  
**Примечание:**  
В документации по WSH на Web-сайте MSDN сказано, что первый параметр метода OpenAsTextStream является необязательным, но не указано его значение по умолчанию. Поэтому автор рекомендует всегда указывать первый параметр.  
  
Второй, необязательный, параметр указывает кодировку, в которой будет прочитано содержимое открываемого файла. Отметим, что это кодировка, в которой мы будем читать содержимое файла, а не кодировка, в которой оно реально сохранено. Числовое значение 0 задаёт кодировку ASCII, значение -1 - кодировку Unicode, а значение -2 - системную кодировку по умолчанию. Значение по умолчанию - 0 (то есть указывается кодировка ASCII).  
  
Метод OpenAsTextStream возвращает экземпляр объекта TextStream. Подробнее об этом объекте и вообще о чтении из файлов и записи в них читайте в предыдущей статье цикла.  
  
**Примечание:**  
Разумеется, для открытия файла мы также можем использовать метод OpenTextFile объекта FileSystemObject, описанный в предыдущей статье цикла.  
  
**2.5. Копирование, перемещение и удаление произвольного файла или файлов**  
Ранее мы узнали, как можно скопировать, переместить и удалить файл, полученный вызовом метода GetFile объекта FileSystemObject. Однако в этом случае, перед тем как собственно скопировать, переместить или удалить файл, нам следует его получить. То есть выполнить операцию, которая во многих случаев будет лишней...  
  
Поэтому, если нам нужно только скопировать, переместить или удалить файл, не выполняя больше никаких манипуляций с ним, или выполнить эти действия сразу над несколькими файлами, мы можем воспользоваться способами, описанными в этом параграфе.  
  
Так, скопировать произвольный файл мы можем, вызвав метод CopyFile объекта FileSystemObject в таком формате:  
  
<экземпляр объекта FileSystemObject>.CopyFile(  
<путь к копируемому файлу>,  
<путь к файлу или папке назначения>[,  
<перезаписывать ли файл, если он уже существует>]  
);  
  
Первый параметр задаёт полный путь к копируемому файлу в виде строки. В имени собственно файла допускается использовать символы-маски \* и ?, так что мы можем скопировать сразу несколько файлов, удовлетворяющих определённому критерию.  
  
Второй параметр задаёт либо путь к файлу назначения, либо путь к папке назначения в виде строки. Правило здесь очень простое: если значение второго параметра заканчивается символом обратного слеша (\) или если путь к копируемому файлу (значение первого параметра) содержит символы-маски, то предполагается, что второй параметр задаёт путь к папке назначения; в противном случае он задаёт путь к файлу назначения.  
  
**Примечание:**  
В случае указания путей к несуществующим файлам или папкам при выполнении веб-сценария возникнет ошибка. Также ошибка возникнет, если путь к копируемому файлу, содержащий символы-маски, не указывает ни на один файл.  
  
Третий, необязательный, параметр позволяет указать, что веб-сценарий будет делать, если в папке назначения уже присутствует файл с тем же именем, что и копируемый файл. Значение true указывает перезаписать такой файл, а значение false - не перезаписывать. Значение по умолчанию - true (то есть файл будет перезаписан).  
  
**Примечание:**  
Если в папке назначения уже существует файл с тем же именем, что и копируемый, а мы отменили перезапись существующих файлов, то при выполнении веб-сценария возникнет ошибка.  
  
Метод CopyFile не возвращает результата.  
  
oFSO.CopyFile("c:\\Work\\\*.exe", "c:\\Dists");  
  
Копируем все исполняемые файлы из папки Work в папку Dists диска C. Поскольку мы указали в пути к копируемому файлу символ-маску \*, второй параметр метода CopyFile задаст путь к папке назначения.  
  
oFSO.CopyFile("c:\\Work\\testfile.txt", "c:\\Work\\Temp\\");  
  
Копируем файл testfile.txt из папки Work в папку Work\Temp диска C. Обратим внимание, что мы завершили путь к папке назначения символом обратного слеша, тем самым указав, что это именно путь к папке, а не к файлу.  
  
oFSO.CopyFile("c:\\Work\\testfile.txt", "c:\\Work\\testfile2.txt");  
  
Копируем файл testfile.txt в ту же папку под именем testfile2.txt.  
  
Для перемещения произвольного файла в другую папку предназначен метод MoveFile объекта FileSystemObject. Формат его вызова таков:  
  
<экземпляр объекта FileSystemObject>.MoveFile(  
<путь к перемещаемому файлу>,  
<путь к файлу или папке назначения>  
);  
  
Параметры этого метода аналогичны параметрам уже знакомого нам метода CopyFile, и ведёт он себя так же.  
  
oFSO.MoveFile("c:\\Work\\\*.exe", "c:\\Dists");  
  
Перемещаем все исполняемые файлы из папки Work в папку Dists диска C.  
  
oFSO.MoveFile("c:\\Work\\testfile.txt", "c:\\Work\\Temp\\");  
  
Перемещаем файл testfile.txt из папки Work в папку Work\Temp диска C.  
  
oFSO.MoveFile("c:\\Work\\testfile.txt", "c:\\Work\\Temp\\testfile2.txt");  
  
Перемещаем файл testfile.txt из папки Work в папку Work\Temp диска C под именем testfile2.txt, тем самым фактически его переименовывая.  
  
Удалить файл с диска можно вызовом метода DeleteFile объекта FileSystemObject. Формат его вызова таков:  
  
<экземпляр объекта FileSystemObject>.DeleteFile(  
<путь к удаляемому файлу>[,  
<удалить ли файл с установленным атрибутом "только для чтения">]  
);  
  
Первым, обязательным, параметром этому методу передаётся путь к удаляемому файлу. Он должен быть представлен в виде строки. В имени файла, присутствующем в этом пути, можно использовать символы-маски \* и ?; то есть мы сможем удалить сразу несколько файлов.  
  
Второй, необязательный, параметр указывает, что делать в случае, если удаляемый файл имеет установленный атрибут "только для чтения". Значение true указывает, что такой файл всё равно следует удалить, а значение false - не делать этого. Значение по умолчанию - false (файлы с установленным атрибутом "только для чтения" удаляться не будут).  
  
Метод DeleteFile не возвращает результата.  
  
**Примечание:**  
При вызове метода DeleteFile файл будет удалён безвозвратно, без помещения в системную Корзину. При попытке удаления несуществующего файла возникнет ошибка. Также ошибка возникнет, если путь к удаляемому файлу, содержащий символы-маски, не указывает ни на один файл.  
  
oFSO.DeleteFile("c:\\Work\\testfile.txt");  
  
Удаляем файл testfile.txt из папки Work диска C.  
  
oFSO.DeleteFile("c:\\Work\\\*.txt");  
  
Удаляем все текстовые файлы из той же папки.  
  
**2.6. Проверка существования файла**  
Часто бывает нужно проверить, существует ли на диске файл с указанным путём, перед его открытием или созданием. Для этого мы можем использовать метод FileExists объекта FileSystemObject. В качестве единственного параметра он принимает путь к файлу в виде строки и возвращает true, если файл с этим путём существует, и false в противном случае.  
  
var sFilePath = "c:\\Work\\testfile.txt";  
if (oFSO.FileExists(sFilePath))  
oFSO.DeleteFile(sFilePath);  
  
**2.7. Получение имени временного файла**  
Другая операция, которую время от времени приходится выполнять, - создание временного файла для хранения каких-либо рабочих данных. Такие временные файлы должны иметь уникальные имена, исключающие конфликты с именами уже существующих на диске файлов. Как правило, имена временных файлов генерируются случайным образом.  
  
Получить такие имя для создания временного файла можно вызовом метода GetTempName объекта FileSystemObject. Этот метод не принимает параметров и возвращает строку со сгенерированным именем файла. Отметим, что, во-первых, оно уже будет включать расширение tmp, а во-вторых, не будет содержать путь, так что временный файл можно будет создать в любой папке.  
  
var sFileName = oFSO.GetTempName();  
var oTS = oFSO.CreateTextFile("c:\\Work\\" + sFileName, true);  
oTS.WriteLine("Test!!!");  
oTS.Close();  
  
Здесь мы получаем имя временного файла, создаём файл с таким именем в папке Work диска C и записываем в этот файл строку "Test!!!". Подробнее о создании текстовых файлов и записи в них рассказывалось в предыдущей статье цикла, посвящённой файловому вводу-выводу.  
  
  
**Дополнительные материалы**

* [Статья об HTML-приложениях](http://www.thevista.ru/link.php?url=http://msdn.microsoft.com/en-us/library/ms536496.aspx)
* [Статья об объекте FileSystemObject](http://www.thevista.ru/link.php?url=http://msdn.microsoft.com/en-us/library/z9ty6h50.aspx)

*Продолжение следует...*

**Веб-сайт состоит из множества файлов: текстового контента, кода, стилей, медиа-контента, и так далее. Когда вы создаете веб-сайт, вы должны собрать эти файлы в рациональную структуру на вашем локальном компьютере, убедитесь, что они могут общаться друг с другом, и весь ваш контент выглядит правильно, прежде чем вы, в конечном итоге**[**загрузите их на сервер**](https://developer.mozilla.org/ru/Learn/Getting_started_with_the_web/Publishing_your_website)**. Работая с файлами, обсуждайте некоторые вопросы, о которых вы должны быть в курсе, чтобы вы могли рационально настроить файловую структуру для вашего веб-сайта.**

## Где ваш веб-сайт должен располагаться на вашем компьютере?[EDIT](https://developer.mozilla.org/ru/docs/Learn/Getting_started_with_the_web/Dealing_with_files$edit#Где_ваш_веб-сайт_должен_располагаться_на_вашем_компьютере)

Когда вы работаете на веб-сайте локально на вашем компьютере, вы должны держать все связанные файлы в одной папке, которая отражает файловую структуру опубликованного веб-сайта на сервере. Эта папка может располагаться где угодно, но вы должны положить её туда, где вы сможете легко её найти, может быть, на вашем рабочем столе, в домашней папке или в корне вашего жесткого диска.

1. Выберите место для хранения проектов веб-сайта. Здесь, создайте новую папку с именем web-projects (или аналогичной). Это то место, где будут располагаться все ваши проекты сайтов.
2. Внутри этой первой папки, создайте другую папку для хранения вашего первого веб-сайта. Назовите ее test-site (или как-то более творчески).

## Небольшое отступление о регистре и пробелах[EDIT](https://developer.mozilla.org/ru/docs/Learn/Getting_started_with_the_web/Dealing_with_files$edit#Небольшое_отступление_о_регистре_и_пробелах)

Вы заметите, что в этой статье, мы просим вас называть папки и файлы полностью в нижнем регистре без пробелов. Это потому что:

1. Многие компьютеры, в частности веб-серверы, чувствительны к регистру. Так, например, если вы положили изображение на свой веб-сайт в test-site/MyImage.jpg, а затем в другом файле вы пытаетесь вызвать изображение как test-site/myimage.jpg, он может не сработать.
2. Браузеры, веб-серверы и языки программирования не обрабатывают пробелы последовательно. Например, если вы используете пробелы в имени файла, некоторые системы могут отнестись к имени файла как к двум именам файлов. Некоторые серверы заменяют пробелы в вашем имени файла на "%20" (символьный код для пробелов в URI), нарушая все ваши ссылки. Лучше разделять слова с помощью тире и нижнего подчеркивания: my-file.htmlили my\_file.html.

По этим причинам, лучше всего приобрести привычку писать названия ваших папок и файлов в нижнем регистре и без пробелов, по крайней мере, пока вы не поймете, зачем это нужно. Так вы столкнетесь с меньшим количеством проблем.

## Какую структуру должен иметь ваш веб-сайт?[EDIT](https://developer.mozilla.org/ru/docs/Learn/Getting_started_with_the_web/Dealing_with_files$edit#Какую_структуру_должен_иметь_ваш_веб-сайт)

Далее давайте взглянем на то, какую структуру должен иметь наш тестовый сайт. Наиболее распространенные вещи, которые есть в любом сайте, создаваемом вами: индексный файл HTML, папки, содержащие изображения, файлы стилей и файлы скриптов. Давайте создадим их сейчас:

1. **index.html**: Этот файл обычно содержит контент домашней страницы, то есть текст и изображения, которые люди видят, когда они впервые попадают на ваш сайт. Используя ваш текстовый редактор, создайте новый файл с именемindex.html и сохраните его прямо внутри вашей папки test-site.
2. **Папка images**: Эта папка обычно содержит все изображения, которые вы используете на вашем сайте. Создайте папку с именем images внутри вашей папки test-site.
3. **Папка styles**: Эта папка обычно содержит CSS код, используемый для стилизации вашего контента (например, настройка текста и цвета фона). Создайте папку с именем styles внутри вашей папки test-site.
4. **Папка scripts**: Эта папка обычно содержит весь JavaScript код, используемый для добавления интерактивных функций на вашем сайте (например, кнопки которые загружают данные при клике). Создайте папку с именем scriptsвнутри вашей папки test-site.

**Примечание**: На компьютерах под управлением Windows у вас могут возникнуть проблемы с отображением имен файлов, поскольку у Windows есть надоедливая настройка с названием **Скрывать расширения для известных типов файлов**включенную по умолчанию. Обычно вы можете отключить ее, перейдя в проводник, выбрать вариант **Свойства папки...** и снять флажок **Скрывать расширения для зарегистрированных типов файлов**, затем щёлкнуть **OK**. Для получения более точной информации, охватывающей вашу версию Windows, выполните поиск в Интернете.

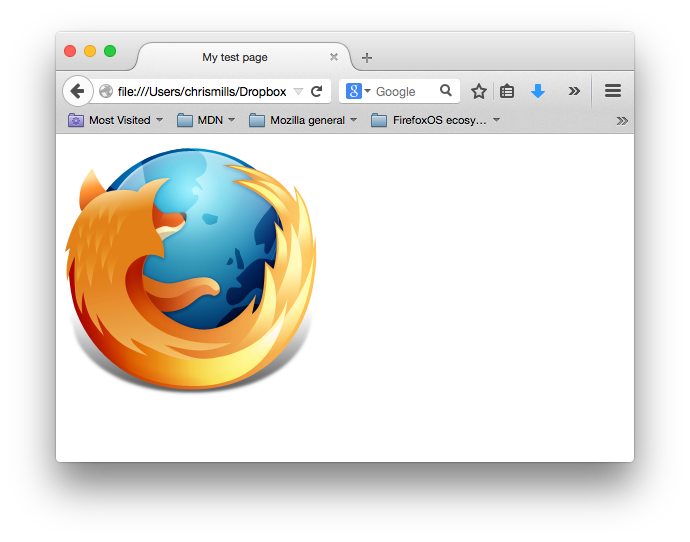
## Путь к файлам[EDIT](https://developer.mozilla.org/ru/docs/Learn/Getting_started_with_the_web/Dealing_with_files$edit#Путь_к_файлам)

Для того, чтобы файлы общались друг с другом, вы должны указать файлам путь между ними - обычно один файл знает, где находится другой. Чтобы продемонстрировать это, мы вставим немного HTML в наш файл index.html и научим его отображать изображение, которое вы выбрали в статье ["Каким должен быть ваш веб-сайт?"](https://developer.mozilla.org/ru/Learn/Getting_started_with_the_web/What_should_your_web_site_be_like)

1. Скопируйте изображение, которое вы выбрали ранее, в папку images.
2. Откройте ваш файл index.html и вставьте следующий код в файл именно в таком виде. Прямо сейчас не беспокойтесь о том, что все это значитс - позже в этом руководстве мы рассмотрим структуру более подробно.
3. <!DOCTYPE html>
4. <html>
5. <head>
6. <meta charset="utf-8">
7. <title>My test page</title>
8. </head>
9. <body>
10. <img src="" alt="My test image">
11. </body>

</html>

1. Строка <img src="" alt="My test image"> - это HTML код, который вставляет изображение на страницу. Мы должны сказать HTML, где находится изображение. Изображение находится внутри папки images, которая находится в той же директории что и index.html. Пройдя вниз по файловой структуре от index.html до нашего изображения, получим путь к файлу, который нам нужен. Он выглядит как images/your-image-filename. Например наше изображение, названное firefox-icon.png, имеет такой путь к файлу: images/firefox-icon.png.
2. Вставьте путь к файлу в ваш HTML код между двойными кавычками src="".
3. Сохраните ваш HTML файл, а затем загрузите его в вашем браузере (двойной щелчок по файлу). Вы должны увидеть вашу новую веб-страницу, отображающую ваше изображение!



Несколько правил о путях к файлам:

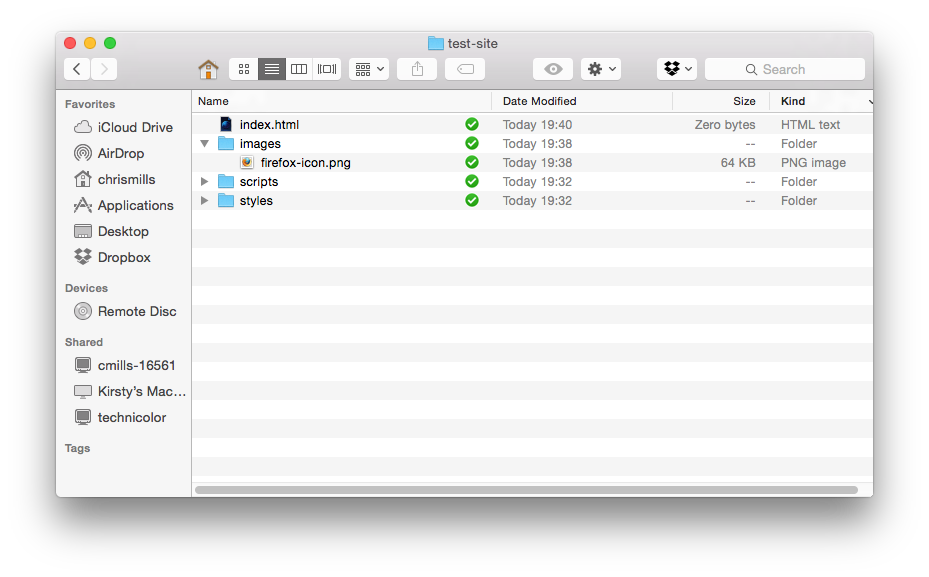
* Для ссылки на целевой файл в той же директории, что и вызывающий HTML файл, просто используйте имя файла, например, my-image.jpg.
* Для ссылки на файл в поддиректории, напишите имя директории в начале пути, плюс косую черту (forward slash, слеш), например: subdirectory/my-image.jpg.
* Для ссылки на целевой файл в директории **выше** вызывающего HTML файла, напишите две точки. Например, если index.html находится внутри подпапкиtest-site, а my-image.png - внутри test-site, вы можете обратиться к my-image.png из index.html, используя ../my-image.png.
* Вы можете комбинировать их так, как вам нравится, например../subdirectory/another-subdirectory/my-image.png.

На данный момент это все, что вам нужно знать

**Примечание**: Файловая система Windows стремится использовать обратный слеш (backslash), а не косую черту, например C:\windows. Это не имеет значения, даже если вы разрабатываете веб-сайт на Windows, вы все равно должны использовать обычные слеши в вашем коде.

## Что должно быть сделано?[EDIT](https://developer.mozilla.org/ru/docs/Learn/Getting_started_with_the_web/Dealing_with_files$edit#Что_должно_быть_сделано)

К настоящему моменту структура вашей папки должна выглядеть примерно так:



HTML5 Files API - чтение файлов

[Веб-программирование](https://professorweb.ru/my/html/html5/level1/web_info.php) --- [HTML5](https://professorweb.ru/my/html/html5/level1/html5_index.php) --- Чтение файлов

Возможность веб-хранилища поддерживается в HTML5 на хорошем уровне. Но это не единственный способ получения информации веб-страницами. В эту область понемногу вводится несколько других стандартов, направленных на выполнение разных типов задач хранения данных. Одним из таких стандартов является File API, который технически не входит в HTML5, но имеет хороший уровень поддержки на всех современных браузерах.

Судя по расплывчатому названию этого стандарта, может показаться, что это всеохватывающий стандарт для чтения и записи файлов на жесткий диск клиента. Но этот стандарт не настолько амбициозный или мощный. Он просто разрешает посетителю веб-сайта выбрать файл на своем жестком диске и передать его непосредственно коду JavaScript, исполняющемуся на просматриваемой веб-странице. Код может открыть этот файл и работать с его данными, будто это простой текст или что-то более сложное. Здесь ключевым аспектом является то обстоятельство, что файл передается непосредственно коду JavaScript. В отличие от обычной выгрузки файла, он никогда не отправляется на веб-сервер.

Также важно знать, что File API не может делать. Самое важное, что он не может изменять файлы или создавать новые файлы. Чтобы сохранить какие-либо данные, нужно прибегать к другому механизму, например данные можно отправить на веб-сервер посредством запроса XMLHttpRequest или же поместить их в локальное хранилище.

Судя по этому, можно подумать, что интерфейс File API менее полезен, чем локальное хранилище, и для большинства веб-сайтов это будет правильный вывод. Но этот стандарт приоткрывает дверь в область, в которую HTML раньше не входил, по крайней мере без помощи модулей расширений.

В настоящее время интерфейс File API является необходимой функциональностью для определенных типов специализированных приложений, но в будущем его возможности могут быть расширены, и важность его значительно возрастет. Например, будущие версии интерфейса могут позволять веб-страницам сохранять файлы на жесткий диск клиента при условии, что пользователь контролирует имя файла и место его сохранения, используя диалоговое окно "Сохранить как". Модули расширения, наподобие Flash, уже оснащены такой способностью.

Поддержка браузерами интерфейса File API

Интерфейс File API не имеет такой широкой поддержки, как веб-хранилище. Текущая браузерная поддержка этого интерфейса приводится в таблице ниже:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| Поддержка браузерами интерфейса File API | | | | | | | |
| Браузер | IE | Firefox | Chrome | Safari | Opera | Safari iOS | Android |
| Минимальная версия | 10 | 3.6 | 8 | 6 | 11.1 | - | 3 |

Эти браузеры почти наверняка не реализуют все возможности File API, т.к. некоторые части стандарта (для работы с большими объемами двоичных данных и "вырезания" порций данных) все еще находятся в процессе разработки.

Получение файла

Прежде чем интерфейс File API сможет что-либо сделать с файлом, ему нужно этот файл получить. Эту задачу можно выполнить тремя разными способами, но все они одинаковые в одном ключевом аспекте — веб-страница может получить файл только в том случае, если посетитель явно выберет и предоставит его веб-странице.

Способы получения файла следующие:

Посредством элемента <input>

Присвоив атрибуту type значение file, мы получим стандартное окно для закачивания файла. Но с помощью небольшого сценария JavaScript и File API этот файл можно открыть локально.

Посредством скрытого элемента <input>

Элемент <input> очень непривлекательный. Чтобы не обезображивать им свою страницу, его можно скрыть и создать более прилично выглядящую кнопку. Нажатие этой кнопки активирует JavaScript-код, вызывающий метод click() скрытого элемента <input>, который открывает стандартное диалоговое окно выбора файла.

Посредством метода drag and drop

Если браузер поддерживает этот метод, файл можно перетащить с рабочего стола или окна браузера и отпустить его в определенной области веб-страницы.

В последующих разделах мы рассмотрим все эти подходы более подробно.

Чтение текстового файла

Самой простой операцией File API будет чтение содержимого простого текстового файла. На рисунке ниже показана веб-страница, которая с помощью этого интерфейса считывает разметку веб-страницы, а потом выводит этот текст в окне браузера:

Создание этого примера начинается с элемента <input type="file">, который создает текстовое поле и кнопку "Выберите файл":

<input id="fileInput" type="file" size="50" onchange="processFiles(this.files)">

<div id="fileOutput"></div>

Но в то время как элемент <input> обычно вставляется в контейнер <form>, чтобы файл можно было закачать на веб-сервер, в данном случае наш элемент <input> играет самостоятельную роль. Когда посетитель страницы выбирает файл, активируется событие onchange элемента <input>, что в свою очередь активирует функцию processFiles(). Как раз на этом этапе и открывается файл посредством самого обыкновенного кода JavaScript.

Теперь рассмотрим по частям функцию processFiles(). Сперва нам нужно взять первый файл из коллекции файлов, предоставленных элементом <input>. Если явно не разрешить выбор нескольких файлов (посредством атрибута multiple), коллекция файлов будет гарантированно содержать только один файл, размещенный в элементе 0 массива файлов:

function processFiles(files) {

var file = files[0];

...

}

Все объекты файлов обладают тремя потенциально полезными свойствами. Свойство name сообщает нам имя файла (без пути), свойство size указывает размер файла в байтах, а свойство type информирует о MIME-типе файла, если его можно определить. Эти свойства можно считывать и использовать их в дополнительной логике, например, отказаться обрабатывать файлы больше определенного размера или разрешить обрабатывать файлы только определенного типа.

Далее создается объект FileReader для обработки файла:

// ...

var reader = new FileReader();

Теперь мы почти готовы вызвать один из методов объекта FileReader, чтобы извлечь содержимое файла. Но эти методы являются асинхронными. Это означает, что они начинают чтение файла, но не ожидают получения данных. Поэтому, чтобы получить данные, сначала нужно обработать событие onload:

reader.onload = function (e) {

// Когда это событие активируется, данные готовы.

// Вставляем их в страницу в элемент <div>

var output = document.getElementById("fileOutput");

output.textContent = e.target.result;

};

Наконец, подготовив этот обработчик события, можно вызывать метод readAsText() объекта FileReader:

reader.readAsText(file);

Этот метод сбрасывает все содержимое файла в одну длинную строку, вставляемую в свойство e.target.result, которое в свою очередь отправляется событию onload.

Метод readAsText() работает должным образом только для текстового содержимого файла, но не для двоичного. Это означает, что он идеально подходит для работы с файлами HTML, как показано на рисунке выше.

Кроме метода readAsText(), объект FileReader имеет еще несколько других методов для чтения файлов: readAsBinaryString(), readAsDataURL() и readAsArrayBuffer(), но последний метод не поддерживается в Firefox.

Метод readAsBinary() предоставляет веб-приложению возможность считывать двоичные данные, хотя он вставляет эти данные в текстовую строку несколько неуклюже, что не является особенно эффективным. А если вы еще захотите разобраться с этими данными, то для этого вам придется мучиться с исключительно запутанным кодом.

Метод readAsDataURL() предоставляет легкий способ захватывать данные изображения. Мы рассмотрим применение этого метода далее, но сначала мы выясним, как сделать нашу страницу более красивой.

Замена элемента <input>

Веб-разработчики сходятся во мнении: стандартный элемент управления <input>, применяемый для выгрузки файлов, выглядит далеко не лучшим образом. Но хотя нам не избежать использования его, совсем необязательно, чтобы пользователи его видели. Мы можем просто скрыть его с помощью следующего стилевого правила:

#fileInput {

display:none;

}

Теперь нам нужно добавить новый элемент управления, который будет инициировать процесс предоставления файла. Для этого достаточно обычной кнопки со ссылкой, которую мы можем разукрасить каким угодно образом:

<button onclick="showFileInput()">Добавить файл</button>

Последним шагом будет обработка нажатия кнопки путем инициализации вручную элемента <input> через вызов метода click() этого элемента:

function showFileInput() {

var fileInput = document.getElementById("fileInput");

fileInput.click();

}

Теперь нажатие этой кнопки запускает функцию showFileInput(), которая "нажимает" скрытую кнопку "Выберите файл" и отображает диалоговое окно для выбора файла. Это, в свою очередь, активирует событие onchange скрытого элемента <input>, которое запускает функцию processFiles() точно таким же образом, как и раньше.

Чтение файла изображения

Как мы узнали, объект FileReader обрабатывает текстовое содержимое в один простой прием. Благодаря методу readAsDataURL() он с такой же легкостью обрабатывает и изображения.

На рисунке ниже показан пример, для реализации которого используются две новые возможности - поддержка изображений и выбор файла методом drag and drop:

На этой странице изображение можно представить двумя способами: выбрать файл изображения с помощью элементов управления внизу или перетащить и отпустить нужный файл в выделенную пунктиром рамку.

Выбранное изображение используется в качестве фона элемента, хотя его с тем же успехом можно было вставить в холст и обрабатывать попиксельно в холсте. Этот метод можно применить, например, чтобы создать страницу, на которую посетитель может вставить изображение, доступное для редактирования, а затем сохранить конечный результат посредством вызова объекта XMLHttpRequest.

При создании такой страницы сначала нужно решить, какой элемент будет получать перетаскиваемые файлы. В нашем примере это <div>-элемент dropBox:

<div id="dropBox">

<div>Перетащите изображение сюда...</div>

</div>

<input id="fileInput" type="file" onchange="processFiles(this.files)">

<img id="thumbnail">

С помощью правил таблицы стилей задаем полю для перетаскивания файла желаемый размер и оформляем рамкой и фоном:

#dropBox {

margin: 15px;

width: 300px;

height: 300px;

border: 5px dashed gray;

border-radius: 8px;

background: lightyellow;

background-size: 100%;

background-repeat: no-repeat;

text-align: center;

}

#dropBox div {

margin: 100px 70px;

color: orange;

font-size: 25px;

font-family: Verdana, Arial, sans-serif;

}

Возможно, вы заметили, что в поле для перетаскивания файла изображения установлены свойства background-size и background-repeat для подготовки к следующей операции. Когда файл изображения перетаскивается в поле <div>, то изображение используется в качестве фона этого элемента. Свойство background-size обеспечивает уменьшение размеров изображения, чтобы его можно было видеть полностью. А значение no-repeat свойства background-repeat обеспечивает, что изображение не повторяется для заполнения оставшегося пространства.

Для обработки перетаскивания и отпускания файла нам требуются три события: ondragenter, ondragover и ondrop. При загрузке страницы ко всем этим событиям подключается соответствующий обработчик:

var dropBox;

window.onload = function() {

dropBox = document.getElementById("dropBox");

dropBox.ondragenter = ignoreDrag;

dropBox.ondragover = ignoreDrag;

dropBox.ondrop = drop;

}

Функция ignoreDrag() обрабатывает как событие ondragenter (которое инициализируется, когда указатель мыши с перетаскиваемым файлом входит в зону сбрасывания), так и событие ondragover (которое срабатывает постоянно во время движения курсора мыши в зоне сбрасывания). Такой подход возможен потому, что нам никак не нужно реагировать на эти действия, кроме как сообщить браузеру не предпринимать никаких действий. Код функции выглядит следующим образом:

function ignoreDrag(e) {

// Обеспечиваем, чтобы никто другой не получил это событие,

// т.к. мы выполняем операцию перетаскивания

e.stopPropagation();

e.preventDefault();

}

Событие ondrop более важное, т.к. в нем мы получаем файл и обрабатываем его. Но поскольку файл для страницы можно предоставить двумя способами, собственно для выполнения работы функция drop() вызывает функцию processFiles():

function drop(e) {

// Аннулируем это событие для всех других

e.stopPropagation();

e.preventDefault();

// Получаем перемещенные файлы

var data = e.dataTransfer;

var files = data.files;

// Передаем полученный файл функции для обработки файлов

processFiles(files);

}

Функция processFiles() является последним этапом в процессе перетаскивания файла. Она создает объект FileReader, подключает функцию обработки к событию onload и вызывает метод readAsDataURL() для преобразования данных изображения в данные URL:

function processFiles(files) {

var file = files[0];

var reader = new FileReader();

reader.onload = function (e) {

// Используем URL изображения для заполнения фона

dropBox.style.backgroundImage = "url('" + e.target.result + "')";

};

// Начинаем считывать изображение

reader.readAsDataURL(file);

}

Объект FileReader имеет еще несколько других событий, которые можно использовать при чтении файлов изображений. Событиеonprogress срабатывает периодически в процессе длинных операций, чтобы предоставить информацию об объеме загруженных данных на текущий момент. (Операцию можно аннулировать до ее завершения, вызвав метод abort() объекта FileReader.) Событие onerrorсрабатывает в случае проблем с открытием или чтением файла. А событие onloadend — при завершении операции любым способом, включая ее преждевременное завершение вследствие ошибки.

# Обзор автономных приложений

[Веб-программирование](https://professorweb.ru/my/html/html5/level1/web_info.php) --- [HTML5](https://professorweb.ru/my/html/html5/level1/html5_index.php) --- Обзор автономных приложений

Чтобы просматривать веб-сайт, нужно подключиться к интернету. На сегодняшний день все это знают. Так зачем же тогда нужно рассматривать автономные приложения? Как будто мы собираемся вернуться в прошлое столетие. В конце концов, разве на своем пути к мировому господству веб-приложения не сбросили владычество нескольких поколений автономных настольных приложений?

Осуществление множества разнообразных задач — от записей в социальной сети до покупки книг в интернет-магазине — просто невозможно без подключения к интернету. Но не стоит забывать, что даже веб-приложения не предназначены для обязательного постоянного пребывания в Сети. В них заложена возможность продолжать работу в периоды временного отсутствия подключения к интернету. Иными словами, автономное веб-приложение допускает временные сетевые отказы.

Это обстоятельство имеет особо важное значение для посетителей, пользующихся для доступа к сети портативными устройствами, такими как смартфоны и планшетные компьютеры с возможностью веб-доступа. Чтобы понять суть данной проблемы, представьте себе, что вы работаете с веб-приложением на одном из таких устройств, находясь в поезде, и в это время поезд въезжает в туннель. Шансы велики, что ваше веб-приложение выдаст сообщение о неисправимом сбое и вам придется начать все заново, когда поезд выедет из туннеля и восстановится подключение к интернету.

А вот автономное веб-приложение позволит избежать такого неблагоприятного развития событий. Хотя некоторые возможности могут стать временно недоступными, приложение в основном останется работоспособным. (Опять же, некоторые туннели длиннее других. Амбициозное автономное приложение может продолжать работу на протяжении трехчасового авиарейса или трехнедельного путешествия по Африке, если это то, что вам нужно. По сути, время нахождения в автономном режиме ничем не ограничено.)

## Кэширование файлов с помощью манифеста

Основным техническим приемом, который делает автономный режим работы возможным, является кэширование — загрузка файла с веб-сервера и сохранение его копии на жестком диске клиента. Таким образом, при потере интернет-подключения браузер сможет использовать копию страницы, сохраненную в кэше.

Для создания страницы с возможностью работы в автономном режиме нужно выполнить следующие три шага:

1. Создать файл манифеста. ***Манифест*** —- это специальный файл, в котором хранится информация, указывающая браузерам, какие файлы следует сохранять, какие не сохранять, а какие файлы заменять каким-либо другим содержимым. Этот пакет кэшируемого содержимого называется автономным приложением (offline application).
2. Модифицировать веб-страницу, чтобы она обращалась к манифесту. Таким образом, браузер будет знать, что при запросе страницы нужно загрузить файл манифеста.
3. Настроить веб-сервер. Самое важное — веб-сервер должен предоставлять файл манифеста с правильным MIME-типом.

В области веб-разработки кэширование не представляет собой ничего нового. Браузеры регулярно кэшируют содержимое, чтобы не загружать повторно одни и те же файлы. В конце концов, если несколько страниц веб-сайта используют один и тот же файл таблицы стилей, зачем загружать его для каждой страницы? Но механизм управления этим типом кэширования отличается от механизма работы автономных приложений.

Традиционное кэширование выполняется, когда вместе с запрошенным браузером файлом веб-сервер отправляет дополнительную информацию, которая называется заголовком Cache-Control. Эти заголовки предоставляют браузеру информацию о том, следует ли кэшировать данный файл и на протяжении какого времени использовать котированную копию, прежде чем запрашивать у веб-сервера, не изменился ли этот файл. Обычно, кэшированная копия содержимого веб-страниц хранится короткое время; намного дольше хранятся кэшированные копии ресурсов, используемых вебстраницами, таких как таблицы стилей, изображения и файлы сценариев.

В противовес, автономное приложение управляется отдельным файлом — манифестом, и длительность его хранения не управляется временными лимитами. Вместо временных лимитов используется следующее правило: если веб-страница является частью автономного приложения, если имеется кэшированная копия этого приложения и если определение этого приложения не претерпело никаких изменений, тогда используется кэшированная копия.

Веб-разработчик может добавить к этому правилу определенные исключения и дополнения, например, указать браузеру не кэшировать определенные файлы или заменить один файл каким-либо другим. Но ему нет надобности заботиться о сроках хранения и других, потенциально проблематичных подробностях.

## Создание манифеста

Манифест является центральной частью предоставления автономной работы HTML5. Это текстовый файл, содержащий список файлов, которые нужно кэшировать. Файл манифеста всегда начинается словами (прописными буквами):

**CACHE** MANIFEST

После этого заголовка следует список файлов, которые нужно кэшировать. Далее приведен пример файла манифеста для кэширования двух веб-страниц:

CACHE MANIFEST

MyPage1.html

MyPage2.html

Интервалы (как пустая строка в приведенном примере манифеста) не обязательны, можете вставлять их в любом месте.

Для автономного приложения браузер должен кэшировать все, что может потребоваться этому приложению: веб-страницы и ресурсы, используемые этими вебстраницами (например, сценарии, графику, таблицы стилей и встроенные шрифты). Далее приведен пример более подробного манифеста со всеми типами файлов:

**CACHE** MANIFEST

*# pages*

MyPage1.html

MyPage2.html

*# styles & scripts*

MyStyle.css

MyScript.js

*# pictures # fonts*

Images/myimage.jpg

Fonts/my\_font.ttf

Обратите внимание на две новые детали в этом манифесте. Первая — несколько строчек начинаются символом #. Эти строчки являются комментариями, предоставляющими информацию относительно типа следующего за ними содержимого. Вторая — для некоторых файлов указаны пути, например Images/myimage.jpg. При условии, что эти файлы находятся на веб-сервере и браузер может иметь доступ к ним, их можно включать в пакет автономного приложения.

Для сложных веб-страниц потребуется большое количество вспомогательных файлов, что может порождать длинные и сложные файлы манифеста. Но наихудшее — это то, что одна-единственная ошибка в имени файла вызовет полный сбой автономного приложения. В перспективе основные веб-редакторы должны облегчить жизнь веб-разработчиков в этом отношении. В них будут добавлены возможности автоматического создания файлов манифеста для указанных файлов и средства для их модифицирования и сопровождения.

В некоторых случаях разработчик может не включать в автономное приложение большие или несущественные ресурсы, такие как, например, изображения большого размера или рекламные баннеры. В этом нет ничего предосудительного, но если есть подозрения, что отсутствие таких файлов способно вызвать какие-либо проблемы (например, сообщения об ошибках, странные пустые области или искаженную компоновку страницы), рассмотрите возможность применения JavaScript для настройки ваших страниц в автономном режиме с помощью метода проверки подключения (описано позже).

Созданный файл манифеста сохраняется в корневой папке веб-сайта. Ему можно присвоить любое имя, но в настоящее время наиболее популярными расширениями файла являются manifest и appcache.

Первое расширение (например, MyManifest.manifest) было бы наиболее логичным, но оно конфликтует с расширением, используемым на некоторых серверах Windows (как часть [процесса развертывания ClickOnce](https://professorweb.ru/my/WPF/documents_WPF/level33/33_2.php), используемого приложениями .NET). Второе расширение тоже логично, но менее популярно.

В любом случае, главное — нужно настроить веб-сервер, чтобы он распознавал используемое расширение файла манифеста.

### Каким может быть объем кэша?

Часто задаваемый вопрос - есть ли какие-либо ограничения на объем кэшируемой информации? Максимальный объем кэша, выделяемого для автономных приложений, варьируется, в зависимости от браузера.

Очевидно, что браузеры на мобильных устройствах находятся на нижнем пределе диапазона. Так как объем хранилища на таких устройствах ограничен, им приходится быть экономными в своем кэшировании. На момент написания этих строк версия браузера Safari для iPad и iPhone ограничивала каждое автономное приложение 5 Мбайт кэша.

Браузеры настольных компьютеров на удивление неоднородны. Firefox по умолчанию выделяет каждому автономному приложению 50 Мбайт кэша, а пользователь может еще повысить этот предел. (Пользовательские параметры кэша в Firefox можно установить на вкладке Network в окне, открываемом после выбора последовательности команд меню Tools --- Options --- Advanced.)

А вот браузер Chrome выделяет каждому автономному приложению ничтожные 5 Мбайт кэша, обойти которое можно либо с помощью [специализированного приложения Chrome](http://code.google.com/chrome/extensions/apps.html" \t "_blank), либо [изменением настроек браузера](https://groups.google.com/a/chromium.org/forum/" \l "!topic/chromium-html5/6RHxi5BdKO4" \t "_blank). Разработчики браузера Chrome планируют в будущем устранить это ограничение и предоставить пользователям явный контроль над параметрами кэша для каждого автономного приложения, но на данном этапе 5 Мбайт — это все, что они получают.

К сожалению, этот неоднородный подход к выделению кэша браузерами представляет проблему. Например, автономное приложение, требующее больше чем 5 Мбайт кэша, будет работать без проблем в браузере Firefox, но не в Chrome. Что еще хуже, при посещении вашего сайта пользователем Chrome этот браузер будет пытаться кэшировать приложение, но не сможет этого сделать по причине ограничения, после чего просто выбросит все загруженные данные. Это не только тратит впустую время и трафик, но также не предоставляет пользователю никакой возможности работать в автономном режиме. Он просто не сможет использовать ваше приложение до тех пор, пока не подключится к интернету.

Из всего этого можно сделать следующий вывод: в перспективе автономные приложение будут, скорее всего, иметь большие объемы кэша в своем распоряжении. Но в настоящее время веб-разработчики должны учитывать, что им доступен намного меньший объем — всего 5 Мбайт. Вследствие этого непрактично использовать возможность автономной работы для улучшения производительности, например, загружая большие, часто используемые файлы и сохраняя их в течение длительного времени. Но хочется надеяться, что эта ситуация в скором времени изменится.

## Браузерная поддержка автономных приложений

К этому времени вы, наверное, уже поняли, что автономные приложения поддерживаются всеми основными браузерами, за обычным исключением — Internet Explorer. Поддержка распространяется на несколько версий назад, что практически обеспечивает возможность пользователей браузеров Firefox, Chrome и Safari исполнять автономные приложения. Минимальные версии основных браузеров, поддерживающих автономные приложения, перечислены в таблице:

|  |  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- | --- |
| *Браузерная поддержка автономных приложений* | | | | | | | |
| **Браузер** | IE | Firefox | Chrome | Safari | Opera | Safari iOS | Android |
| **Минимальная версия** | - | 3.5 | 5 | 4 | 10.6 | 2.1 | 2 |

Но различные браузеры поддерживают автономные приложения по-разному. Самая большая разница состоит в объеме хранилища, выделяемого автономным приложениям для кэширования своих файлов. Это важный аспект, т.к. он определяет, какие веб-сайты будут помещены в кэш для автономного доступа, а какие — нет.

Если браузер не имеет встроенной поддержки автономных приложений (например, Internet Explorer), предоставить такую поддержку каким-либо обходным решением невозможно. Но это обстоятельство не должно останавливать вас от использования возможности работы ваших веб-сайтов в автономном режиме. В конце концов режим автономной работы — хоть и полезная, но всего лишь необязательная возможность.

Браузеры, не поддерживающие автономных приложений, все равно будут работать с такими сайтами, только будут требовать для этого подключения к интернету. А пользователи, нуждающиеся в возможности автономной работы с сайтом, например, те, кто часто путешествуют, откроют для себя, насколько это ценно пользоваться браузером иным, чем Internet Explorer.

Alexandr Erohin ✯ alexerohinzzz@gmail.com © 2011 - 2017