**Web Storage: easier, more powerful client-side data storage**

**Introduction**

Web applications are getting more advanced each day, with more elaborate uses of JavaScript as well as upcoming standards and technologies. We increasingly rely on these applications, many of them becoming a part of our daily lives. One area in which Web application development has been lacking is the ability to store data on the client-side. That is, until now.

[Web Storage](http://dev.w3.org/html5/webstorage/) is a W3C specification that provides functionality for storing data on the client side until the end of a session (Session Storage), or beyond (Local Storage). It is much more powerful than traditional cookies, and easier to work with. In this article we will look at why this is, and how to use it.

**The current problem: cookies can crumble**

Before we go on further, let’s take a very brief look on why the current way of storing data on the client-side — cookies — is a problem:

* Low size: Cookies generally have a maximum size of around 4KB, which is not much good for storing any kind of complex data
* It’s difficult for cookies to keep track of two or more transactions on the same site, which might be happening in two or more different tabs
* Cookies can be exploited using techniques such as cross site scripting, resulting in security breaches

Other (less popular) alternatives to cookies include techniques involving query strings, hidden form fields, flash based local shared objects, etc. Each with their own set of problems related to security, ease of use, size restrictions etc. So up until now we have been using pretty bad ways of storing data on the user’s end. We need a better way, which is where Web Storage comes in.

**Web Storage**

[The W3C Web Storage specification](http://dev.w3.org/html5/webstorage/) was designed as a better way of storing data on the client-side. It has two different types of storage: Session Storage and Local Storage.

Both Session and Local Storage will typically be able to store around 5Mb of data per domain, which is significantly more than cookies. As we’ll read on, we’ll find out more about them, and what makes Web Storage a better storage mechanism

**Session Storage**

Session Storage has one purpose: To remember all the data in your session, and forget it as soon you close the tab (or window) you are working in.

**Setting and retrieving data**

To set a key value pair in Session Storage, you just need to write a line like this:

sessionStorage.setItem(yourkey, yourvalue);

To retrieve the data again, you would write:

var item = sessionStorage.getItem(yourkey);

To store the value This is a sample sentence in the Session Storage, you could write:

sessionStorage.setItem(1, 'This is a sample sentence');

Here, the key value is 1, but that does not mean that its the first value as such. It just converts the number 1 in a string '1' and uses that as key, it does not put that key value pair in the first position as such.

And to retrieve that sentence back inside a JavaScript alert, you’d write:

var item = sessionStorage.getItem(1);

alert(item);

Another example of setItem() could be:

sessionStorage.setItem('name', 'john');

and you could retrieve it using

var name = sessionStorage.getItem('name');

**Removing and clearing data**

There are also methods for removing and clearing data from the Session Storage. The removeItem() method is used to remove a particular item from the list:

var item = sessionStorage.removeItem(yourkey);

Bear in mind that you can also just reference a data item’s key and remove it from the list that way:

var items = sessionStorage.removeItem(1);

The clear() method is used to clear all items in the list; you use it in the following way:

sessionStorage.clear();

And you can use the length attribute to find out of the number of key/value pairs in the storage, like this:

var no\_of\_items = sessionStorage.length;

**Local Storage**

Local Storage is used if you want the data to be stored for more than a single session. A simple use case would be to count the number of times a person has visited a Web page. When a page uses Local Storage, the page (or window) can be closed and re-opened, and still show the saved data — it provides persistent storage.

Saving and retrieving data in Local Storage works similarly to Session Storage: it uses the same function names setItem() and getItem(). To save a sentence in Local Storage you write something like this:

localStorage.setItem(1, 'This is a sample sentence');

and to retrieve it:

var data = localStorage.getItem(1);

Also just like Session Storage, Local Storage supports the length attribute, removeItem() and clear().

In both Session Storage and Local Storage, the clear() function has the same objective - to clear *all* the values in list. This means that if you call, say localStorage.clear() then it will remove all local storage from that origin. So all Local Storage data from say, (such as www.example.org, www.example.org:80, www.example.org/abc/, www.example.org/xyz/) will be cleared. However, storage for, say, abc.example.org will not be affected by it. For Session Storage though, it will only clear the storage for the current session.

**A simple example**

To illustrate Web Storage in action, I have created a simple example that uses both Local Storage and Session Storage. Check out the [Web Storage demo page](http://people.opera.com/shwetankd/external/demos/webstorage_demo.htm) to see this in action. The demo will ask you to enter two strings, one for Session Storage, the other for Local Storage. You can then open the Storage Inspector in Opera Dragonfly to access the Web Storage. You'll also notice that once you close and page, and then open it again, the data you typed in for Local Storage is preserved, whereas for Session Storage, this is not the case.

**Using the Storage event**

The specification also provides for the [storage event](http://dev.w3.org/html5/webstorage/#the-storage-event) to be fired whenever the storage area changes. This provides various usefull attributes such as:

* **storageArea**: Tells which kind of storage it is (Session or Local)
* **key**: The key which is being changed.
* **oldValue**: The old value of the key.
* **newValue**: The new value of the key.
* **url**: The URL of the page whose key is changed.

If a clear() method is called, then key, oldValue and newValue attributes are set to null. Here is a [modified version of the demo page](http://people.opera.com/shwetankd/external/demos/webstorage_demo2.htm) mentioned previously, this time using storage events to let the user know the change in values. If you enter a value, and then change it once again, then you'll see an alert mentioning the old value and the new value.

**Where do I get detailed access to Web Storage data on my browser?**

In Opera 10.50+, there are a few ways you can do this. You can enter <opera:webstorage>, as well as <opera:config#PersistentStorage> in the address bar to access high level details of Web Storage (what is the storage limit, where it is saved, etc), but there is an even better way for developers to get detailed information on the Web Storage for a particular page — through the Storage Inspector in Opera Dragonfly, which provides you with much more detailed information.

Opera 10.50+ has a new and improved [Opera Dragonfly](http://www.opera.com/dragonfly/) debugging tool included (which we recently released as an [open source project](http://bitbucket.org/scope/)). Amongst the fixes, improvements and new features is the Storage Inspector. This gives developers a separate tab to access information regarding cookies and the Local and Session Storage of a page. Open Opera Dragonfly and click on the *Storage* tab to access it.

**Important things to keep in mind regarding Web Storage**

* **Storage per origin**: All storage from the same origin will share the same storage space. An origin is a tuple of scheme/host/port (or a globally unique identifier). For example, http://www.example.org and http://abc.example.org are two separate origins, as are http://example.org and https://example.org as well as http://example.org:80 and http://example.org:8000
* **Storage limit**: As of now, most browsers that have implemented Web Storage, including Opera, have placed the storage limit at 5 Mb per domain. You can change this storage limit on a per-domain basis by saving some data from a domain in Session or Local Storage, then going to <opera:webstorage>. That domain will then appear in the list there, and you can click it to access statistics and options, including size of data saved for that domain, what the storage limit is, and what the browser will do when that limit is exceeded.
* **Security considerations and associated best practices**: Storage is assigned on a per-origin basis. Someone might use DNS Spoofing to make themselves look like a particular domain when in fact they aren’t, thereby gaining access to the storage area of that domain on a user’s computer. SSL can be used in order to prevent this from happening, so users can be absolutely sure that the site they are viewing is from the same domain name.
* **Where not to use it**: If two different users are using different pathnames on a single domain, they can access the storage area of the whole origin and therefore each other’s data. Hence, it is advisable for people on free hosts who have their sites on different directories of the same domain (for example, freehostingspace.org/user1/ and freehostingspace.org/user2/), to not use Web Storage on their pages for the time being.
* **Web Storage is not part of the HTML5 spec**: It is a [whole specification in itself](http://dev.w3.org/html5/webstorage/).

**Running your web applications offline with HTML5 AppCache**

**Introduction**

Web applications have become a major part of many people’s lives, so much so that many of us use them all the time. Wouldn't it be great if we could use them even when offline? Until recently, there wasn’t any viable way to do this – however, with the introduction of the [W3C HTML5 application cache](http://dev.w3.org/html5/spec/offline.html) feature, it is possible to make your web applications run offline as well as online. Let us see how...

**Why have your app run offline?**

Web applications are becoming more complex and capable every day. There are many examples of web applications doing the same job as desktop applications in various fields (think about Google Docs, Picasa, etc.). However, one major disadvantage is that they cannot work when the user is not connected to the Internet.

This is where HTML5's new offline storage comes in. It tries to remove that disadvantage by defining a way to store files in a cache, so that when the user is offline, the browser still has access to the necessary files. These can be HTML, CSS or JavaScript files, or any other assets the site needs to run.

**Saving your files for offline use with the application cache**

HTML5 has a feature for offline web applications called application cache, or AppCache for short. Files stored in this AppCache are available to the application even when the user is offline. You can specify which files you want to store in the AppCache using a manifest file.

**How is the application cache different from the normal browser cache?**

There a number of ways in which AppCache is different from the browser's normal cache. First of all, is the intention behind the two. AppCache is intended for proper web applications whereas the browser's cache is for normal web pages in general. The normal cache will cache pretty much any page whereas the AppCache will only cache pages which are specifically included in the manifest file. Plus, the normal cache is unreliable, as we dont know which pages (and which resources within those pages) will be available for sure.

AppCache is exciting because now the developer has much more programmatic control over the cache, which means much more certainty and control over how web applications could behave offline. Also to note, is that you can have multiple pages share the same AppCache. Also, with AppCache, you can make use of the API to determine what is the state of the AppCache, and then even have it update itself.

**The manifest file**

This file resides on the server and dictates which files should be stored client-side in the browser's AppCache, in readiness for the user going offline. Let's delve a bit deeper in into how it works.

You can give the manifest file any name you want, but best practice dictates that you give it an extension of .manifest. Every manifest file has to start with CACHE MANIFEST, after which you list the files you want to be stored and made available for offline use. Comments can be made by putting # at the beginning of a line. A very simple manifest file looks like so:

CACHE MANIFEST

#You can also use the CACHE: section header to explicitly declare the following three files.

style.css

script.js

index.htm

The manifest file has to have the correct MIME Type, which is text/cache-manifest. To deal with these, you could have an extension .manifest for the manifest file, and add the following in the .htaccess file on your server:

AddType text/cache-manifest .manifest

**Linking the HTML file with the manifest file**

Now that you have created the manifest file telling which files need to be cached in your application, you need tell the HTML page to use that cache. To do this you have to link the page to the manifest file by including the manifest attribute in the <HTML> tag of your page. For example:

<html manifest=”demo.manifest”>

If your web application has more than one page, make sure that all the pages link to the manifest file in this way, otherwise they won't be part of the AppCache, and won't work offline.

**Using section headers for better control over the AppCache**

So far, we've seen a very basic example of how to use the manifest file. With the use of section headers, we can actually specify exactly how a certain file is to be cached, or not.

**Explicitly defining the files to be cached**

You can use the CACHE: section header to explicitly declare which files need to be cached. For example, the previous example of the manifest file can be written like so, and it will function in exactly the same way:

CACHE MANIFEST

CACHE:

style.css

script.js

index.htm

The only difference is that in this example we have explicitly declared that all those files will be part of the application cache. Here is a very simple [example page that uses the CACHE: section header](http://people.opera.com/shwetankd/demos/2/index.htm).

It is important to note that the path mentioned for the files should be relative to the location of the manifest file. In the examples here, we are assuming that the files mentioned are in the same directory as the manifest file. You can use relative as well as absolute URLs when stating the files in the manifest file.

The files specified as part of the CACHE: will load from AppCache (not the server) even if you are online, provided that there is no change in the manifest file. If, however, the browser finds an updated manifest file, then the new cache will once again be downloaded according to the what new manifest file says. So AppCache may not be suitable for sites with fast moving content like news blogs, for example, but can be very useful for web applications which do a particular thing and want to work offline (for example, a calendar app, or a to-do list, etc.).

**What if I want a file to bypass the cache and load directly from the server?**

If a page is associated with a manifest file, then only those files mentioned in the manifest file will try to load regardless of whether the user is online or offline. Now, there may be situations where you might want some file to bypass this cache when the user is online, so that it connects and downloads fresh from the server instead of the cache (For example, some dynamic content from a CGI Script).

Basically, if a page is associated with a manifest file, then all network traffic for its files is blocked, and files either have to be loaded from the AppCache, or fail to load. The NETWORK: section header gives exceptions for this rule. You can use the NETWORK: section header to declare which files should NOT to be cached, so that they are downloaded from the server, and never be part of the application cache. The NETWORK: section header respects the browser's normal cache header. So if a file is supposed to be cached by the browser's normal cache, then it will still be cached by it (just like any other file not specified in the AppCache), even if it's specified under the NETWORK: section header.

CACHE MANIFEST

CACHE:

style.css

script.js

index.htm

NETWORK:

style2.css

In the above example, style2.css will always be downloaded from the server and never be part of the application cache. Keep in mind that it may be the case that you have too many files to list which need to bypass the cache, making writing them all down under the NETWORK: section header cumbersome. In this case you can use the asterisk character (\*), so that all urls are allowed to go online if you are online.

Check out my [example that uses a manifest file employing a NETWORK: section header](http://people.opera.com/shwetankd/demos/3/index.htm). You will notice what when you are offline and re-load the page, the page does reload, but the background styling disappears. This is because the background styling in this example is in the file style2.css, which is under the NETWORK: section header, meaning that it is not cached and will only load when you are online and re-load the page.

**Providing fallback content**

The FALLBACK: section header is used to define fallbacks to be used in place of files that fail to load (or load incompletely):

CACHE MANIFEST

CACHE:

style.css

script.js

index.htm

NETWORK:

style2.css

FALLBACK:

main\_image.jpg backup\_image.jpg

The fallback content is supposed to be cached and only used in case the main content does not load. In the above example, backup\_image.jpg is cached by AppCache so if main\_image.jpg cannot load, backup\_image.jpg will load in its place. Check out my [manifest backup example](http://people.opera.com/shwetankd/demos/4/index.htm) — if you go to this page, and disconnect from the internet and then re-load the page, the browser will try to load the image, but since you're not online (and the image is not cached) it will not load, and hence the fallback content will load in its place instead. (The browser will first take a little time trying to load the main content, and only then load the fallback content...so be patient!)

This manifest file is utilized in my [example that provides fallback content for a number of images](http://people.opera.com/shwetankd/demos/5/index.htm).

**Using the application cache API and events to make sure the latest files are used in your cache**

One of the great things about application cache is that now you, the programmer, have access to how the cache could behave. You have access to events which can tell you about the current state of the application cache, and have functions to asynchronously update the it too. For exampple, you can use window.applicationCache to find out if the browser supports application cache or not. Let's take a look at some other ways in which you can gain programmatic control over the application cache.

**Statuses**

You can check the current status of the application cache using window.applicationCache.status, which returns a numeric value mapped to the following states:

0 - uncached

If the page is not linked to the application cache. Also, the the very first time the application cache is being downloaded, then during the time it is being downloaded, the AppCache will have a status of uncached.

1 - idle

When the browser has the latest version of the AppCache, and there aren no updated versions to download, then the status is set to Idle.

2 - checking

The duration of when the page is checking for an updated manifest file, then the status is set to Checking.

3 - downloading

The duration of when the page is actually downloading a new cache (if an updated manifest file has been detected), the status is set to downloading

4 - updateready

Once the browser finishes downloading that new cache, it is ready to be used (but still not being used yet). During this time, the status is set as updateready

5 - obsolete

In case the manifest file cannot be found, then the status is set to obsolete and the application cache gets deleted. It is important to know, that in case the manifest file (or any file mentioned in the manifest file except those which have a fallback) fail to load, then it will be counted as an error and the old application cache will continue to be used.

**Events**

Certain events also get fired, depending on what going on with the AppCache at the moment.

checking

This gets fired when browser is checking for attempting to download the manifest for the first time, or is checking if there is an updated version of the manifest file.

noupdate

If there is no updated version of the manifest file on the server, then noupdate is fired.

downloading

If the browser is downloading the cache for the first time, or if is downloading an updated cache, then this is fired.

progress

This is fired for each and every file which is downloaded as part of the AppCache.

cached

This is fired when all the resources have finished downloading, and application is cached.

updateready

Once resources have finished re-downloading for an updated cached file, then updateready is called. Once this happens, then we can use swapCache() (as explained later in the article) to make the browser to use this newly updated cache.

obsolete

This is fired if the manifest file cannot be found (404 error or 410 error).

error

This can be fired for a number of reasons. If the manifest file can't be found, then the application cache download process has to be aborted, in which case this event can be fired. It can also be fired in case the manifest file is present, but any of the files mentioned in the manifest file can't be downloaded properly. It can even be fired in case the manifest file changes while the update is being run (in which case the browser will wait a while before trying again), or in any other case where there is a fatal error.

The event handlers for these events are all prefixed by 'on'. For example, onchecking, onupdateready, onerror, etc.

The application cache API has a few things worth noting:

* window.applicationCache.update(): This will trigger the application cache download process, which is nearly the same as reloading the page. It simply checks if the manifest has changed, and if so downloads a fresh version of all the content in the cache (respecting any cache headers). Note that even though a new cache is created with this, the page will continue to use the old cache. To make the page use the new cache you have just downloaded, you must use the swapCache() function.
* window.applicationCache.swapCache(): This function tells the browser to start using the new cache data if it is available. It is important to note that even if there is a new manifest file, the application will still continue using the old cache (as specified in the old manifest file) untill swapCache() is called. Once swapCache() is called, then the cache will be used as specified from the new manifest file.

Normally you won’t need to use the update() function, as the browser should automatically do this when reloading a page. Most commonly the swapCache() function will be used in conjunction with the onupdateready event.

In the following example, if you change the manifest file and reload the page, the browser will download the new files in the cache, and then switch to the new cache (as the swapcache() function is called):

<html manifest="demo.manifest">

<head>

<script type="text/javascript">

window.applicationCache.addEventListener('updateready', function(){

window.applicationCache.swapCache();

}, false);

</script>

</head>

<body>

...

</body>

</html>

If the page you build is unlikely to be reloaded by the user for a while, then you could periodically call the update() function to check for new updates to the manifest file, and if so, call the swapcache() function on an updateready event to download and switch to the new cache:

setInterval(function () { window.applicationCache.update(); }, 3600000); // Check for an updated manifest file every 60 minutes. If it's updated, download a new cache as defined by the new manifest file.

window.applicationCache.addEventListener('updateready', function(){ // when an updated cache is downloaded and ready to be used

window.applicationCache.swapCache(); //swap to the newest version of the cache

}, false);

This code will check for an updated version of the manifest file every 60 minutes. If it finds a different version of the manifest file on the server than it previously encountered, it will download a new cache based on this new manifest. Once that happens, an updateready event will be fired, stating that an updated copy of the cache has finished downloading and is ready to be used. We can then explicitly use the swapCache() function to swap the old cache with the new one we just downloaded.

In this way, you can ensure that the user's cache will stay updated.

# Web Storage, Application Cache и WebSQL

Чтобы делать приложения, которые могут работать в полностью автономном режиме, нам нужно познакомиться со следующими технологиями: HTML5 Application Cache, Web Storage и WebSQL.  
Мной уже были опубликованы вводные статьи, касающиеся [Web Storage](http://dev.opera.com/articles/view/web-storage/) и [HTML5 Application Cache](http://dev.opera.com/articles/view/offline-applications-html5-appcache/). Рекомендую их к прочтению если вы еще не знакомы с основными понятиями. В данной статье будут пересмотрены эти технологии, в том числе и WebSQL, и описаны варианты их совместного эффективного использования. Все эти технологии поддерживаются настольной версией браузера Opera 11.10, Opera Mobile 11, браузерами на движке WebKit (в iOS и Google Android).

|  |
| --- |
| **Примечание:** Все примеры, описанные ниже, доступны в [этом архиве](http://dev.opera.com/articles/view/taking-your-web-apps-offline-web-storage-appcache-websql/offline-web-apps.zip). Вы можете скачать архив и смотреть примеры, попутно читая статью (естественно, находясь в офлайне). |

# Зачем вообще нужен «офлайн-режим»?

В настоящее время веб-страницы представляют собой не просто текст. Интернет получает всё более широкое распространение, и веб-приложения, соответственно, тоже, становясь всё более сложными и совершенными. И наша зависимость от них только растёт. Мы можем видеть всё больше и больше примеров, когда веб-приложения успешно заменяют традиционные настольные приложения, чьим основным преимуществом всегда была их возможность спокойно работать в автономном режиме. Дать такую же возможность веб-приложениям не позволяло отсутствие необходимых технологий и инструментов.  
~~Но всё меняется, когда приходят они...~~  
Но теперь это не проблема! С появлением таких технологий, как HTML5 Application Cache, Web Storage и WebSQL мы наконец получили способ заставить веб-приложения работать в офлайне:

* Application Cache позволяет держать копию HTML, CSS и других элементов нашего веб-приложения в автономном режиме, которые будут использоваться, когда сеть будет недоступна.
* Web Storage основывается на механизмах хранения, подобных cookies, представляя собой их более гибкую и более мощную реализацию.
* WebSQL реализует полномерную SQL-базу данных внутри вашего браузера, которая может хранить копии данных веб-приложения для автономной работы, позволяя пользователям продолжить работу с данными даже при потере соединения с сетью. Данные синхронизируются с сервером при последующем подключении к сети.

Существует множество причин потери соединения с сетью: отключение питания, проблемы сетевым оборудованием, отсутствующий (или слабый) сигнал в случае работы через сотовый телефон. Как разработчики, мы хотим быть уверенными, что даже в подобных ситуациях наши веб-приложения будут работать корректно (по крайней мере, в разумных пределах).

# Кэш: грузимся и при отсутствии сети

В общем случае, если мы не в сети, то при загрузке или перезагрузке страниц сайта мы получим ошибку. Первое, что нужно сделать — убедиться, что наши пользователи могу видеть страницы и работать с веб-приложением даже при отсутствующем подключении, т.е. все картинки, CSS, JavaScript и сама HTML-страница должны корректно загружаться в этом режиме.  
Достигается это путем использования технологии Application Cache (называемую также AppCache). Чтобы её использовать, нужно для начала объявить manifest-файл, в котором указать имена файлов, которые необходимы приложению для работы офлайн.  
Пример файла *demo.manifest*:

CACHE MANIFEST

CACHE:

logo.png

style.css

script.js

jquery.js

index.htm

Какие бы ссылки ни содержал этот manifest-файл, соответствующее содержимое помещается в кэш, и впоследствии к этому содержимому уже будет доступ. Объявить manifest-файл очень просто — нужно использоваться атрибут *manifest* элемента *<html>*:

<html manifest="demo.manifest">

Все файлы, объявленные в этом файле, будут помещены в кэш вашего браузера. Даже если человек не в сети и пытается загрузить страницу, то все ресурсы, указанные в этом файле, загрузятся браузером.  
Более подробную информацию об этом вы можете найти в статье про [HTML5 Application Cache](http://dev.opera.com/articles/view/offline-applications-html5-appcache).

# Что насчет данных?

AppCache решает задачу доступности некоторых элементов сайта в офлайне, но, возможно, нам захочется хранить некоторый объём пользовательских данных или, например, его последние поисковые запросы. В другом случае, может быть, вам захочется хранить более структурированные данные. В любом случае, Web Storage и WebSQL будут лучшим решением.

### Используем Web Storage

Web Storage прекрасно подходит для хранения небольших объемов информации, нежели огромных таблиц с данным, о чём кратко и поговорим в этой статье, рассмотрим примеры. Более подробную информацию вы можете узнать в отдельной [статье, посвящённой Web Storage](http://dev.opera.com/articles/view/web-storage/).  
Существует очень много мест, где частое отключение электричества — обычное явление («Чубайс, привет! :)», — от переводчика). Пользователю приходится сидеть и ждать, когда он сможет продолжить свою работу в Интернете, пока не включат электричество. А представьте, если кто-то оказался в подобной ситуации, заполняя многостраничную форму на каком-либо сайте, набирая большую статью для блога или важного электронного письма. Во время отключения питания (или сядут аккумуляторы) пользователь потеряет все эти данные. Не будет ли лучше, если после входа в сеть, ему будут доступны все несохраненные им записи, с которыми он продолжит работу?  
Давайте посмотрим, что можно сделать на странице, содержащей обычное текстовое поле *<textarea>*? Страница должна сохранять всё, что мы набираем, в локальное хранилище каждые несколько секунд, а в случае её перезагрузки или закрытия, страница должна загружать последний сохранённый в поле текст.  
Допустим, наша страница содержит поле *<textarea>* с *id* «draft»:

...

<textarea id="draft" rows="10" cols="30"></textarea>

...

Напишем простую функцию, которая будет сохранять в локальное хранилище содержимое *<textarea>*:

function saveMessage(){

var message = document.getElementById("draft");

localStorage.setItem("message", message.value)

}

Установим интервал сохранения в полсекунды:

setInterval(saveMessage, 500);

|  |
| --- |
| **Примечание:** здесь мы использовали *setInterval()* для простоты, чтобы сохранять сообщение в локальное хранилище каждые полсекунды. (Вы могли бы улучшить эту процедуру, например, сохраняя содержимое текстового поля только в том случае, если пользователь в него что-то ввёл). |

Также нужно убедиться, что каждый раз, когда страница открывается или перезагружается, в текстовое поле загружалось последнее сохранённое содержимое из локального хранилища (*localStorage*):

window.addEventListener("DOMContentLoaded", loadMessage, false);

function loadMessage() {

var textbox = document.getElementById("draft");

var message = localStorage.getItem("message");

if (!message) {

textbox.value = "";

}else {

textbox.value = message;

}

}

Посмотрите пример [работы с Web Storage](http://people.opera.com/shwetankd/webstorage/webstoragedemo.htm). Это вообще шикар  
ная вещь, если вам нужно сохранять локально небольшие кусочки информации.

### Работаем в автономном режиме

Для перехода в автономный режим нужно его, собственно, активировать (в браузере Opera: «Меню» → «Настройки» → «Работать автономно» либо «Файл» → «Работать автономно»). Свойство *navigator.onLine* имеет значение *false* в случае, если браузер находится в автономном режиме, в противном случае оно имеет значение *true*. Однако, во многих случаях лучше было бы использовать события. Когда пользователь переключается в автономный режим, срабатывает событие *offline*, когда переключается обратно — соответственно *online*. Можно воспользоваться этим для вывода небольшого сообщения о переходе в автономный режим.  
Получится что-то типа этого:

...

window.addEventListener( "offline", function(){showWarningDiv("on")}, false);

window.addEventListener( "online", function(){showWarningDiv("off")}, false);

...

function showWarningDiv(status){

var warningdiv = document.getElementById("warning");

if (status == "on"){

warningdiv.innerHTML = "<p style=\"padding:3px;\">Right now you are in offline mode. This message is saved and will be sent to the server the next time you are online.</p>";

} else {

warningdiv.innerHTML = "";

}

}

|  |
| --- |
| **Примечание:** в настоящее время поддержка автономного режима реализована только в Opera и Firefox. |

Имеет смысле убедиться, что формы не пытаются отправить данные в то время, когда пользователь работает в автономном режиме. Чтоб проверить это, можем сделать так:

...

window.addEventListener( "submit", submitForm, false);

...

function submitForm(){

saveMessage();

if (!navigator.onLine){

return false;

}

}

При отправке данных формы срабатывает событие *submit*, которое вызывает функцию *submitForm()*. Эта функция сначала сохранит сообщение в локальном хранилище, затем, если пользователь работает в автономном режиме, данные никуда не отправятся.  
Вы можете усовершенствовать этот пример, чтобы он сохранял копию на сервер каждые несколько секунд, чтобы там она была доступна на тот случай, если пользователь случайно удалит данные у себя. Это особенно важно в тех случаях работы, например, с конфиденциальной информацией: вы, скажем, хотите, чтобы информация о вашей кредитной карте хранилась только у вас — в локальном хранилище.  
Посмотрите также [более продвинутый пример](http://people.opera.com/shwetankd/webstorage/demofolder/webstorage_advanced.zip), который использует для хранение информации *sessionStorage*. Если вы не закрываете страницу (даже в случае её перезагрузки) текст, введённый в текстовое поле, там и останется. Страница также будет отправлять содержимое этого поля на сервер каждые несколько секунд и, соответственно, обновлять время последнего сохранения. Данный подход может использоваться в блоговых движках и сервисах электронной почты для периодического сохранения «черновиков», что позволит продолжить работу в случае проблем с подключением.

### WebSQL: еще «глужбе» в офлайн

Web Storage прекрасно подходит для хранения небольших объемов информации, а что если мы захотим хранить целую базу данных? Как насчёт того, чтобы веб-приложение могло делать различные запросы к базе данных, поиск по ней?  
Здесь Web Storage уже не позволит развернуться — нужно что-то более надёжное. А именно — WebSQL. WebSQL представляет собой локальную SQLite базу данных, в которой вы можете хранить свои данные, используя комбинацию Javascript и SQL.

##### Работа с WebSQL-базами данных

Перво-наперво нужно убедиться, поддерживает ли браузер WebSQL? Сделать это можно через свойство *window.openDatabase*:

if (window.openDatabase){

//rest of your code

} else{

alert("It seems your browser does not have support for WebSQL. Please use a browser which does, otherwise parts of this application may not run as intended."); //or any other similar message

}

##### Создание и открытие базы данных

Создать и открыть базу данных можно, используя команду *openDatabase*, так:

var db = openDatabase("food\_db", "1.0", "Web SQL Storage Demo Database", 1\*1024\*1024); // creates a database called 'food\_db' with version number 1.0, description as 'Web SQL Demo Database' and a size of 1MB.

Мы только что создали базу данных *food\_db* «версии» 1.0, с описанием *«Web SQL Storage Demo Database»*, размером 1 МБ. Переменная *db* представляет собой указатель на объект базы данных, который мы и будем в дальнейшем использовать.

|  |
| --- |
| **Примечание:** размер базы данных устанавливается в байтах. Поэтому мы определили размер в формате 1\*1024\*1024, что составляет 1 МБ. Если нужно установить размер, например, 4 МБ, следует указывать величину 4\*1024\*1024 соответственно. |

##### Работа с базой данных

Мы создали и открыли базу данных. Теперь можно выполнять различные операции над ней, используя SQL-команды. Операции мы будем производить, вызывая функцию *transaction()* объекта базы данных (в нашем случае это *db*). Её вызов возвращает объект в качестве указателя, над которым мы будем выполнять различные команды, используя *executeSQL()*. Синтаксис этой команды следующий:

executeSql(sqlStatement, arguments, callback, errorCallback);

Из параметров только *sqlStatement* является обязательным, остальные — необязательные.  
Так, например, если мы хотим создать таблицу, нужно будет написать следующее:

...

db.transaction(

function(t){ // This is the callback with "t" as the transaction object

t.executeSql("CREATE TABLE IF NOT EXISTS cal\_list (food\_name TEXT PRIMARY KEY, calories REAL, servings TEXT)");

}

);

...

Этот код создаст таблицу *cal\_list* (если её не существовало) с полями *food\_name*, *calories* и *servings*.

##### Добавление записей в таблицу

Выполнять запросы на добавление записей в таблицу — простая задача для WebSQL. Рассмотрим пример:

var food\_name = "pizza";

var amount\_of\_calories = 320;

var serving\_size = "one slice";

db.transaction(

function(t){

t.executeSql("INSERT INTO cal\_list VALUES (?, ?, ?)", [food\_name, amount\_of\_calories, serving\_size]);

}

);

Первый знак вопроса экранирует параметр *food\_name*, второй — *amount\_of\_calories*, а третий — *serving\_size*. Этот код добавляет запись в таблицу *cal\_list* со значениями: *pizza*, *320* и *one slice* соответствующие столбцы.  
Выполним другой запрос — на получение данных:

var min\_cal\_amount = 300;

...

t.executeSql("SELECT \* FROM cal\_list WHERE calories > ?", [min\_cal\_amount]);

Этот код выполнит запрос на выборку всех строк со значением *calories* больше 300: знак вопроса экранирует переменную *min\_cal\_amount*.

##### Обработка результатов запроса

Ну, вот, мы создали базу данных с таблицами, в которые записали данные, теперь мы хотим сделать запрос и вывести полученные результаты. Обычно, мы получаем кучу результатов для одного SQL-запроса и нам нужно как-то обработать эти результаты, чтобы вывести их в виде таблицы или каком-то другом структурированном виде на странице. Третий параметр функции executeSQL() определяет успешность выполнения запроса. Ниже приведён пример обработки результатов:

var list = document.getElementById("thelist");

var food;

var min\_cal\_amount = 400;

var serving\_size;

db.transaction(

function(t){

t.executeSql("SELECT food\_name AS food, calories AS amount\_of\_calories, servings as serving\_size FROM cal\_list where calories > ?" ,[min\_cal\_amount], function(t,r){

for (var i=0; i < r.rows.length; i++){

food = r.rows.item(i).food;

amount\_of\_calories = r.rows.item(i).amount\_of\_calories;

serving\_size = r.rows.item(i).serving\_size;

list.innerHTML +="<li>"+food+" has "+amount\_of\_calories+" KCAL worth of calories.</li>";

}

},

function(t,e) {alert(e.message);})

}

);

Сначала мы определяем количество элементов в результате запроса через *r.rows.length* и пробегаемся от 0 до этого значения. Каждый элемент доступен по *r.rows.item(i)*, где i — номер строки. Названия столбцов можно получить так же. Для получения получения значения столбца *food* используем *r.rows.item(i).food* и так далее для других столбцов.

# Используем всё и сразу

Не исключено, что вам захочется использовать Web Storage, Application Cache и WebSQL вместе. Это возможно, всё зависит от задач, которые нужно решить. Например, если вы хотите хранить всего лишь некоторые пользовательские настройки, будет излишеством использовать WebSQL. Для подобных целей лучше подойдёт Web Storage.  
Однако, если у вас большой объём данных, и нужно их как-то обрабатывать, то логичнее было бы использовать WebSQL.  
Посетите страницу с примером про калории, на которой [используются вместе все три технологии](http://people.opera.com/shwetankd/websql/combineddemo.htm): база данных хранится в Web Storage, так что вы сможете искать даже если закроете или перезагрузите страницу, AppCache и WebSQL используются для предоставления возможности работать в автономном режиме.

|  |
| --- |
| **Примечание:** вы можете самостоятельно получить информацию о том, какие базы данных используются в Opera и управлять ими, перейдя по ссылке **opera:webdatabases**. А получить информацию о доменах, которые хранят свои данные в браузере, используя Web Storage, и управлять ими, перейдя по ссылке **opera:webstorage**. |

# Подождите… разве спецификации WebSQL утверждены?

Ещё не окончательно. Но вы уже можете применять эту технологию на приведённом списке браузеров и устройств. Чего нельзя сказать о IndexedDB, у которой нет такой кросс-платформенной поддержки. Осталось только дождаться утверждения спецификации, что и произойдёт в скором времени.  
Важно также отметить, что некоторые настольные версии браузеров, такие как Firefox и Internet Explorer, не будут поддерживать WebSQL, они будут работать с IndexedDB. В этом случае, первое, где можно применить WebSQL — автономное хранилище данных для смартфонов: поддерживаются браузеры Opera Mobile 11, а также браузеры на движке WebKit для платфом Android и iOS.  
Хорошая новость о WebSQL заключается в том, что он предлагает удобный инструмент для создания автономной базы данных в браузере. Он лёгок в изучении и применении и позволяет быстро разработать автономное веб-приложение, которое будет сразу работать в нескольких браузерах.

# Заключение

Веб-приложения теперь получили возможность работать в полностью автономном режиме, используя такие технологии, как Application Cache, Web Storage и WebSQL базы данных. Application Cache нам нужен для кэширования файлов, чтоб исползовать их в автономном режиме; Web Storage — для хранения небольших объёмов информации, ну а WebSQL будет удобным инструментом для работы с большими объёмами данных. Разработчики в зависимости от потребностей могут использовать различное сочетание этих технологий, для создания автономных приложений.