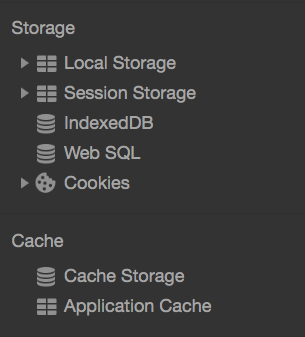
The Types of storage in Browser

There are lots of storage types that the browser offers, it makes it easier when we know how to use, when to use and why to use those.

The mostly used types of storages are Web Storage API (local storage, session storage), Cookies (persistent, session cookies) and Indexed DB.



The other types of storages are WebSQL (not implemented in Firefox, Edge, IE) and Cache API (part of Service worker, which we could discuss in another post)

Web SQL is still not a recognized standard and not recommended to be used. Also, Application Cache is deprecated.

Let’s go through the differences between the types of storage, mentioned above.

# **Client-side Storage Options and When to Use Them**

Storing and manipulating data in the browser — also known as client-side storage — is useful when it’s not necessary or practical to send it to the web server.

Situations for storing and manipulating data in the browser include:

* retaining the state of a client-side application — such as the current screen, entered data, user preferences, etc.
* utilities which access local data or files and have strict privacy requirements
* progressive web apps (PWAs) which work offline

Here are ten options for storing browser data:

1. JavaScript variables
2. DOM node storage
3. Web Storage (localStorage and sessionStorage)
4. IndexedDB
5. Cache API (don’t use AppCache!)
6. File System Access API
7. File and Directory Entries API
8. cookies
9. window.name
10. WebSQL

This article investigates these ten different ways to store data in the browser, covering their limits, pros, cons, and the best uses of each technique.

Before we browse the options, a quick note about data persistence …

# **Data Persistence**

In general, data you store will either be:

1. persistent: it remains until your code chooses to delete it, or
2. volatile: it remains until the browser session ends, typically when the user closes the tab

The reality is more nuanced.

Persistent data can be blocked or deleted by the user, operating system, browser, or plugins at any point. The browser can decide to delete older or larger items as it approaches the capacity allocated to that storage type.

Browsers also record page state. You can navigate away from a site and click back or close and re-open a tab; the page should look identical. Variables and data regarded as session-only are still available.

# **1 JavaScript Variables**

| **metric** | **comment** |
| --- | --- |
| capacity | no strict limit but browser slowdowns or crashes could occur as you fill memory |
| read/write speed | the fastest option |
| persistence | poor: data is wiped by a browser refresh |

Storing state in JavaScript variables is the quickest and easiest option. I’m sure you don’t need an example, but …

const

a = 1,

b = 'two',

state = {

msg: 'Hello',

name: 'Craig'

};

**Advantages**:

* easy to use
* fast
* no need for serialization or de-serialization

**Disadvantages**:

* fragile: refreshing or closing the tab wipes everything
* third-party scripts can examine or overwrite global (window) values

You’re already using variables. You could consider permanently storing variable state when the page unloads.

# **2 DOM Node Storage**

| metric | comment |
| --- | --- |
| capacity | no strict limit but not ideal for lots of data |
| read/write speed | fast |
| persistence | poor: data can be wiped by other scripts or a refresh |

Most DOM elements, either on the page or in-memory, can store values in named attributes. It’s safer to use attribute names prefixed with data-:

1. the attribute will never have associated HTML functionality
2. you can access values via a dataset property rather than the longer .setAttribute() and .getAttribute() methods.

Values are stored as strings so serialization and de-serialization may be required. For example:

// locate <main> element

const main = document.querySelector('main');

// store values

main.dataset.value1 = 1;

main.dataset.state = JSON.stringify({ a:1, b:2 });

// retreive values

console.log( main.dataset.value1 ); // "1"

console.log( JSON.parse(main.dataset.state).a ); // 1

**Advantages**:

* you can define values in JavaScript or HTML — such as <main data-value1="1">
* useful for storing the state of a specific component
* the DOM is fast! (contrary to popular opinion)

**Disadvantages**:

* fragile: refreshing or closing the tab wipes everything (unless a value is server-rendered into the HTML)
* strings only: requires serialization and de-serialization
* a larger DOM affects performance
* third-party scripts can examine or overwrite values

DOM node storage is slower than variables. Use it sparingly in situations where it’s

practical to store a component’s state in HTML.

**3** **Web Storage (localStorage and sessionStorage)**

| **metric** | **comment** |
| --- | --- |
| capacity | 5MB per domain |
| read/write speed | synchronous operation: can be slow |
| persistence | data remains until cleared |

This infamous storage was introduced to us by HTML5. Unlike cookies, Web Storage does not depend on Http requests and is stored in user’s local disc, so it can be used for a lot of other purposes in the browser’s front-end itself.

Web Storage provides two similar APIs to define name/value pairs. Use:

1. window.localStorage to store persistent data, and
2. window.sessionStorage to retain session-only data while the browser tab remains open (but see Data Persistence)

Store or update named items with .setItem():

localStorage.setItem('value1', 123);

localStorage.setItem('value2', 'abc');

localStorage.setItem('state', JSON.stringify({ a:1, b:2, c:3 }));

Retrieve them with .getItem():

const state = JSON.parse( localStorage.getItem('state') );

And delete them with .removeItem():

localStorage.removeItem('state')

Other properties and methods include:

* .length: the number of items stored
* .key(N): the name of the Nth key
* .clear(): delete all stored items

Changing any value raises a storage event in other browser tabs/windows connected to the same domain. Your application can respond accordingly:

window.addEventListener('storage', s => {

console.log(`item changed: ${ s.key }`);

console.log(`from value : ${ s.oldValue }`);

console.log(`to new value: ${ s.newValue }`);

});

**Advantages**:

* simple name/value pair API
* session and persistent storage options
* good browser support

**Disadvantages**:

* strings only: requires serialization and de-serialization
* unstructured data with no transactions, indexing, or search
* synchronous access will affect the performance of large datasets

Web Storage is ideal for simpler, smaller, ad-hoc values. It’s less practical for storing large volumes of structured information, but you may be able to avoid performance issues by writing data when the page unloads.

**USAGE**

* It can be used to store intermediate data and used for sending requests in areas having poor connectivity.
* Can be used to store data that has to be used across tabs, to store user’s information like items in cart, reminders, etc,.
* It is indeed a wardrobe or toolset of all things we need and can be used selectively only at the time we would want to.

**DATA TYPES**

 Any date type like strings, objects, arrays, etc. can be stored with ease.

**SIZE**

Safari allows 5mb of local storage safari and unlimited session storage. While Chrome and Firefox allow 10mb of local and session storage.

**A Types**

## **A1 Session storage**

* Session storage is cleared when the tab is closed.
* session storage is specific to the tab that set it
* Session storage is tab specific, and scoped to the lifetime of the tab.
* Session storage may be useful for storing small amounts of session specific information, for example an IndexedDB key.
* Session storage should be used with caution because it is synchronous and will block the main thread.
* Session storage is limited to about 5MB and can contain only strings.
* Because Session Storage is tab specific, it is not accessible from web workers or service workers.
* In Session Storage, the information is put away in the program's memory for that particular session. This means until you close the program window.
* In contrast to cookies, the information in session Storage is never moved to the sever while making an organization demand. The capacity furthest reaches of SessionStorage is additionally extremely high when contrasted with cookies.

**Window.sessionStorage**

The read-only **sessionStorage** property accesses a session **Storage** object for the current **origin**. sessionStorage is similar to **localStorage**; the difference is that while data in **localStorage** doesn't expire, data in **sessionStorage** is cleared when the page session ends.

* Whenever a document is loaded in a particular tab in the browser, a unique page session gets created and assigned to that particular tab. That page session is valid only for that particular tab.
* A page session lasts as long as the tab or the browser is open, and survives over page reloads and restores.
* Opening a page in a new tab or window creates a new session with the value of the top-level browsing context, which differs from how session cookies work.
* Opening multiple tabs/windows with the same URL creates sessionStorage for each tab/window.
* Duplicating a tab copies the tab's sessionStorage into the new tab.
* Closing a tab/window ends the session and clears objects in sessionStorage.
* Data stored in sessionStorage is specific to the protocol of the page. In particular, data stored by a script on a site accessed with HTTP (e.g., http://example.com) is put in a different sessionStorage object from the same site accessed with HTTPS (e.g., <https://example.com>).
* The keys and the values are always in the UTF-16 string format, which uses two bytes per character. As with objects, integer keys are automatically converted to strings.

**Value**

A Storage object which can be used to access the current origin's session storage space.

**Exceptions**

**SecurityError**

The request violates a policy decision, or the origin is not a valid scheme/host/port tuple (this can happen if the origin uses the file: or data: scheme, for example). For example, the user may have their browser configured to deny permission to persist data for the specified origin.

**Examples**

**Basic usage**

// Save data to sessionStorage

sessionStorage.setItem('key', 'value');

// Get saved data from sessionStorage

let data = sessionStorage.getItem('key');

// Remove saved data from sessionStorage

sessionStorage.removeItem('key');

// Remove all saved data from sessionStorage

sessionStorage.clear();

**Saving text between refreshes**

The following example autosaves the contents of a text field, and if the browser is refreshed, restores the text field content so that no writing is lost.

// Get the text field that we're going to track

let field = document.getElementById("field");

// See if we have an autosave value

// (this will only happen if the page is accidentally refreshed)

if (sessionStorage.getItem("autosave")) {

// Restore the contents of the text field

field.value = sessionStorage.getItem("autosave");

}

// Listen for changes in the text field

field.addEventListener("change", () => {

// And save the results into the session storage object

sessionStorage.setItem("autosave", field.value);

});

**Pros**:

* It is favored over cookies since it is considerably more natural than utilizing cookies to store things.
* Less to and from to the server
* A cookie is size restricted to 4kb and utilizing sessionStorage you can have substantially more information spared per session
* Quicker page loads

**Cons**:

* Just equipped for sparing strings
* It's eradicated after the "session" or program is shut
* Not secure enough to store usernames or passwords
* The information isn't steady for example it will be lost once the window/tab is shut. Like local Storage, it deals with same-inception strategy. In this way, information put away may be accessible on a similar inception.

## **A2 Local storage**

* Local Storage can be cleared by the developer.
* It exists until it is cleared by the website.
* Local storage is shared by all tabs of the same domain
* Local Storage should be avoided because it is synchronous and will block the main thread. It is limited to about 5MB and can contain only strings. LocalStorage is not accessible from web workers or service workers.

**Window.localStorage**

* The local Storage read-only property of the window interface allows you to access a Storage object for the Document's origin; the stored data is saved across browser sessions.
* Local Storage is similar to session Storage, except that while local Storage data has no expiration time, sessionStorage data gets cleared when the page session ends — that is, when the page is closed.

(local Storage data for a document loaded in a "private browsing" or "incognito" session is cleared when the last "private" tab is closed.)

* The keys and the values stored with localStorage are always in the UTF-16 string format, which uses two bytes per character. As with objects, integer keys are automatically converted to strings.
* localStorage data is specific to the protocol of the document. In particular, for a site loaded over HTTP (e.g., http://example.com), localStorage returns a different object than localStorage for the corresponding site loaded over HTTPS (e.g., https://example.com).
* For documents loaded from file: URLs (that is, files opened in the browser directly from the user's local filesystem, rather than being served from a web server) the requirements for localStorage behavior are undefined and may vary among different browsers.
* In all current browsers, localStorage seems to return a different object for each file: URL. In other words, each file: URL seems to have its own unique
* local-storage area. But there are no guarantees about that behavior, so you shouldn't rely on it because, as mentioned above, the requirements for file: URLs remains undefined. So it's possible that browsers may change their file: URL handling for localStorage at any time. In fact some browsers have changed their handling for it over time.
* LocalStorage is one kind of the Web Storage API.
* In localStorage the information on the client program remains for a lifetime except if the client erases or flush the localStorage information from its program, it doesn't make a difference if the client shuts the window or tab the information will stay in the program except if the program erases the memory of the program is cleared.

**Value**

A Storage object which can be used to access the current origin's local storage space.

**Exceptions**

**SecurityError**

The request violates a policy decision, or the origin is not a valid scheme/host/port tuple (this can happen if the origin uses the file: or data: schemes, for example). For example, the user may have their browser configured to deny permission to persist data for the specified origin.

**Examples**

//The following snippet accesses the current domain's local Storage object and //adds a data item to it using Storage.setItem().

localStorage.setItem('myCat', 'Tom');

//The syntax for reading the localStorage item is as follows:

const cat = localStorage.getItem('myCat');

//The syntax for removing the localStorage item is as follows:

localStorage.removeItem('myCat');

//The syntax for removing all the localStorage items is as follows:

localStorage.clear();

**4 IndexedDB**

| **metric** | **comment** |
| --- | --- |
| capacity | depends on device. At least 1GB, but can be up to 60% of remaining disk space |
| read/write speed | fast |
| persistence | data remains until cleared |

* IndexedDB offers a NoSQL-like low-level API for storing large volumes of data.
* The store can be indexed, updated using transactions, and searched using asynchronous methods.
* The IndexedDB API is complex and requires some event juggling.
* Sidelined by promises of ES6, Indexed DB API has a lot more capabilities than the other two. Stores can be created, data can be stored in stores and can be deleted. It has unlimited amount of storage. It can be accessed anytime from the domain that created the DB.
* It is asynchronous and therefore it does not block the UI during any costly operation.
* Since lots of APIs have started supporting promises, this API is being less often used by the developers. Promises store the data and allows it be to used at any point by the application.
* SIZE: Unlimited.
* USAGE: It can be used when costly operation has to be done on the data before retrieving and using it in the app, without affecting the UI thread.
* var db;  
  var request = indexedDB.open("MyTestDatabase");  
  request.onsuccess = function(event) {  
   db = event.target.result;  
  };
* IndexedDB is supported in every modern browser. Its asynchronous, and will not block the main thread. Can be accessible from the window object, web workers, and service workers, making it easy to use them anywhere in your code.

**Advantages**:

* flexible data store with the largest space
* robust transactions, indexing, and search options
* good browser support

**Disadvantages**:

* a complex callback and event-based API

IndexedDB is the best option for reliably storing large quantities of data, but you’ll want to reach for a wrapper library such as idb, Dexie.js, or JsStore.

The following function opens a database connection when passed a name, version number, and optional upgrade function (called when the version number changes):

// connect

function dbConnect(dbName, version, upgrade) {

return new Promise((resolve, reject) => {

const request = indexedDB.open(dbName, version);

request.onsuccess = e => {

resolve(e.target.result);

};

request.onerror = e => {

console.error(`indexedDB error: ${ e.target.errorCode }`);

};

request.onupgradeneeded = upgrade;

});

}

The following code connects to a myDB database and initializes a todo object store (analogous to a SQL table or MongoDB collection). It then defines an auto-incrementing key named id:

(async () => {

const db = await dbConnect('myDB', 1.0, e => {

db = e.target.result;

const store = db.createObjectStore('todo', { keyPath: 'id', autoIncrement: true });

})

})()

Once the db connection is ready, you can .add new data items in a transaction:

db.transaction(['todo'], 'readwrite')

.objectStore('todo')

.add({ task: 'do something' })

.onsuccess = () => console.log( 'added' );

And you can retrieve values, such as the first item:

db.transaction(['todo'], 'readonly')

.objectStore('todo')

.get(1)

.onsuccess = data => console.log( data.target.result );

// { id: 1, task: 'do something' }

**5 Cache API**

| **metric** | **comment** |
| --- | --- |
| capacity | depends on device, but Safari limits each domain to 50MB |
| read/write speed | fast |
| persistence | data remains until cleared or after two weeks in Safari |

* The Cache API provides storage for HTTP request and response object pairs. You can create any number of named caches for storing any number of network data items.
* The API is typically used in service workers to cache network responses for progressive web apps. When a device disconnects from the network, cached assets can be re-served so a web app can function offline.
* For the network resources necessary to load your app and file-based content, use the Cache Storage API (part of service workers).
* Cache Storage API is supported in every modern browser. its asynchronous, and will not block the main thread. They're accessible from the window object, web workers, and service workers, making it easy to use them anywhere in your code.

**Advantages**:

* stores any network response
* can improve web application performance
* allows a web application to function offline
* a modern Promise-based API

**Disadvantages**:

* not practical for storing application state
* possibly less useful outside progressive web apps
* Apple has not been kind to PWAs and the Cache API

The Cache API is the best option for storing files and data retrieved from the network. You could probably use it to store application state, but it’s not designed for that purpose and there are better options.

The following code stores a network response in a cache named myCache:

// cache name

const cacheName = 'myCache';

(async () => {

// cache network response

const stored = await cacheStore('/service.json') );

console.log(stored ? 'stored OK' : 'store failed');

})();

// store request

async function cacheStore( url ) {

try {

// open cache

const cache = await caches.open( cacheName );

// fetch and store response

await cache.add( url );

return true;

}

catch(err) {

return undefined; // store failed

}

}

A similar function can retrieve an item from the cache. In this example, it returns the response body text:

(async () => {

// fetch text from cached response

const text = await cacheGet('/service.json') );

console.log( text );

})();

async function cacheGet( url ) {

try {

const

// open cache

cache = await caches.open( cacheName ),

// fetch stored response

resp = await cache.match(url);

// return body text

return await resp.text();

}

catch(err) {

return undefined; // cache get failed

}

}

**5B AppCache**

AppCache was the defunct predecessor to the Cache API. This isn’t the storage solution you’re looking for. Nothing to see here. Please move along.

**6 File System Access API**

| **metric** | **comment** |
| --- | --- |
| capacity | depends on remaining disk space |
| read/write speed | depends on file system |
| persistence | data remains until cleared |

* The File System Access API allows a browser to read, write, modify, and delete files from your local file system.
* Browsers run in a sandboxed environment so the user must grant permission to a specific file or directory.
* This returns a FileSystemHandle so a web application can read or write data like a desktop app.
* The File System Access API was designed to make it easy for users to read and edit files on their local file system.
* The user must grant permission before a page can read or write to any local file, and permissions are not persisted across sessions.

**Advantages**:

* web apps can securely read and write to the local file system
* less need to upload files or process data on a server
* a great feature for progressive web apps

**Disadvantages**:

* minimal browser support (Chrome only)
* the API may change

This storage option excites me the most, but you’ll need to wait a couple of years before it becomes viable for production use.

The following function saves a Blob to a local file:

async function save( blob ) {

// create handle to a local file chosen by the user

const handle = await window.showSaveFilePicker();

// create writable stream

const stream = await handle.createWritable();

// write the data

await stream.write(blob);

// save and close the file

await stream.close();

}

**7 File and Directory Entries API**

| **metric** | **comment** |
| --- | --- |
| capacity | depends on remaining disk space |
| read/write speed | unknown |
| persistence | data remains until cleared |

* The File and Directory Entries API provides a sandboxed file system available to a domain which can create, write, read, and delete directories and files.

**Advantages**:

* could have some interesting uses

**Disadvantages**:

* non-standard, incompatibilities between implementations, and behavior may change.

MDN explicitly states: do not use this on production sites. Widespread support is several years away at best.

**8 Cookies**

| **metric** | **comment** |
| --- | --- |
| capacity | 80Kb per domain (20 cookies with up to 4Kb in each) |
| read/write speed | fast |
| persistence | good: data remains until it’s wiped or expires |

* Cookies are domain-specific data.
* They have a reputation for tracking people, but they’re essential for any system which needs to maintain server state — such as logging on.
* Unlike other storage mechanisms, cookies are (usually) passed between the browser and server on every HTTP request and response. Both devices can examine, modify, and delete cookie data.
* Cookies have existed since 1994 as little content documents put on a clients' PC by a site, and can be gotten to simply by the site that set it.
* Every way of the area can likewise be characterized by cookies.
* It's the exemplary method of putting away basic string information inside a record.
* Normally, cookies are sent from the server to the client, which would then be able to store it, and send it back to the server on ensuing solicitations. This can be utilized for things like overseeing account meetings and following client data.
* They can be used for communication with the server
* We can set when we want the cookie to expire automatically, instead of having to manually delete
* Potential security issues
* Cookies have basic support in all major browsers.
* Cookies have their uses, but should not be used for storage.
* Cookies are sent with every HTTP request, so storing anything more than a small amount of data will significantly increase the size of every web request.
* They are synchronous, and are not accessible from web workers. Like LocalStorage and SessionStorage,
* cookies are limited to only strings.

**Example of Cookie in operational mode:**

Document. Cookie = "username = Jane Doe; expires = Thu, 18 Dec 2019 12:00:00 GMT; path = /" // set cookie

Document. Cookie = "username =; expires = Thu, 01 Jan 1970 00:00:00 GMT" // delete cookie

**Advantages**:

* a reliable way to retain state between the client and server
* limited to a domain and, optionally, a path
* automatic expiry control with **max-age** (seconds) or **Expires** (date)
* used in the current session by default (set an expiry date to persist the data beyond page refreshes and tab closing)

**Disadvantages:**

* cookies are often blocked by browsers and plugins (they’re generally converted to session cookies so sites continue to work)
* clunky JavaScript implementation (it’s best to create your own cookie handler or opt for a library such as js-cookie)
* strings only (requires serialization and de-serialization)
* limited storage space
* cookies can be examined by third-party scripts unless you restrict access
* blamed for privacy invasion (regional legislation may require you to show a warning for non-essential cookies)
* cookie data is appended to every HTTP request and response which can affect performance (storing 50Kb of cookie data, then requesting ten 1 byte files, would incur one megabyte of bandwidth)

Avoid cookies unless there’s no viable alternative.

**Using HTTP cookies**

* An HTTP cookie (web cookie, browser cookie) is a small piece of data that a server sends to a user's web browser.
* The browser may store the cookie and send it back to the same server with later requests.
* Typically, an HTTP cookie is used to tell if two requests come from the same browser—keeping a user logged in,

**example**. It remembers stateful information for the stateless HTTP protocol.

**Cookies are mainly used for three purposes:**

* **Session management**

Logins, shopping carts, game scores, or anything else the server should remember

* **Personalization**

User preferences, themes, and other settings

* **Tracking**

Recording and analyzing user behavior

**Note**

* Cookies were once used for general client-side storage. While this made sense when they were the only way to store data on the client, modern storage APIs are now recommended. Cookies are sent with every request, so they can worsen performance (especially for mobile data connections). Modern APIs for client storage are the Web Storage API (localStorage and sessionStorage) and IndexedDB.
* To see stored cookies (and other storage that a web page can use), you can enable the Storage Inspector in Developer Tools and select Cookies from the storage tree.

**Creating cookies**

After receiving an HTTP request, a server can send one or more **Set-Cookie** headers with the response. The browser usually stores the cookie and sends it with requests made to the same server inside a **Cookie** HTTP header. You can specify an expiration date or time period after which the cookie shouldn't be sent. You can also set additional restrictions to a specific domain and path to limit where the cookie is sent. For details about the header attributes mentioned below, refer to the **Set-Cookie** reference article.

The **Set-Cookie** and **Cookie** headers

The Set-Cookie HTTP response header sends cookies from the server to the user agent. A simple cookie is set like this:

Set-Cookie: <cookie-name>=<cookie-value>

This instructs the server sending headers to tell the client to store a pair of cookies:

HTTP/2.0 200 OK

Content-Type: text/html

Set-Cookie: yummy\_cookie=choco

Set-Cookie: tasty\_cookie=strawberry

[page content]

Then, with every subsequent request to the server, the browser sends all previously stored cookies back to the server using the Cookie header.

GET /sample\_page.html HTTP/2.0

Host: www.example.org

Cookie: yummy\_cookie=choco; tasty\_cookie=strawberry

The Set-Cookie header in various server-side applications:

* PHP
* Node.JS
* Python
* Ruby on Rails

<https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies>

**9 window.name**

| **metric** | **comment** |
| --- | --- |
| capacity | varies, but several megabytes should be possible |
| read/write speed | fast |
| persistence | session data remains until the tab is closed |

The window.name property sets and gets the name of the window’s browsing context. You can set a single string value which persists between browser refreshes **or linking** elsewhere and clicking back. For example:

let state = { a:1, b:2, c:3 };

window.name = JSON.stringify( state );

Examine the value using:

state = JSON.parse( window.name );

console.log( state.b );

**Advantages**:

* easy to use
* can be used for session-only data

**disadvantages**:

* strings only: requires serialization and de-serialization
* pages in other domains can read, modify, or delete the data (never use it for sensitive information)

window.name was never designed for data storage. It’s a hack and there are better options.

Example code:

// create DB (name, version, description, size in bytes)

const db = openDatabase('todo', '1.0', 'my to-do list', 1024 \* 1024);

// create table and insert first item

db.transaction( t => {

t.executeSql('CREATE TABLE task (id unique, name)');

t.executeSql('INSERT INTO task (id,name) VALUES (1, "wash cat")');

});

// output array of all items

db.transaction( t => {

t.executeSql(

"SELECT \* FROM task",

[],

(t, results) => { console.log(results.rows); }

);

});

**10 WebSQL**

| **metric** | **comment** |
| --- | --- |
| capacity | 5MB per domain |
| read/write speed | sluggish |
| persistence | data remains until cleared |

* WebSQL was an effort to bring SQL-like database storage to the browser.
* Chrome and some editions of Safari support the technology, but it was opposed by Mozilla and Microsoft in favor of IndexedDB.
* Do not use WebSQL! It hasn’t been a viable option since the specification was deprecated in 2010.

**Advantages**:

* designed for robust client-side data storage and access
* familiar SQL syntax often used by server-side developers

**Disadvantages:**

* limited and buggy browser support
* inconsistent SQL syntax across browsers
* asynchronous but clunky callback-based API
* poor performance

**Scrutinizing Storage**

The **Storage API** can examine space available to Web Storage, IndexedDB, and the Cache API. All browsers except Safari and IE support the Promise-based API which offers an **.estimate()** method to calculate the **quota** (space available to the domain) and **usage** (space already used). For example:

(async () => {

if (!navigator.storage) return;

const storage = await navigator.storage.estimate();

console.log(`bytes allocated : ${ storage.quota }`);

console.log(`bytes in use : ${ storage.usage }`);

const pcUsed = Math.round((storage.usage / storage.quota) \* 100);

console.log(`storage used : ${ pcUsed }%`);

const mbRemain = Math.floor((storage.quota - storage.usage) / 1024 / 1024);

console.log(`storage remaining: ${ mbRemain } MB`);

})();

Two further asynchronous methods are available:

* **.persist()**: returns **true** if the site has permission to store persistent data, and
* .**persisted()**: returns **true** if the site has already stored persistent data

The **Application** panel in browser developer tools (named **Storage** in Firefox) allows you to view, modify, and clear localStorage, sessionStorage, IndexedDB, WebSQL, cookies, and cache storage.

You can also examine cookie data sent in the HTTP request and response headers by clicking any item in the developer tools’ **Network** panel.

**Storage Smorgasbord**

None of these storage solutions is perfect, and you’ll need to adopt several in a complex web application. That means learning more APIs. But having a choice in each situation is a good thing — assuming you can choose the appropriate option, of course!