LocalStorage vs sessionStorage

Web storage objects `localStorage` and `sessionStorage` allows to save Data in the Browser.

Isha Jauhari
Jan 13 · 5 min read ★

What's interesting about them is that the data survives a page refresh (for `sessionStorage`) and even a full browser restart (for `localStorage`).

We already have cookies. Why there is need for additional objects?

- The data is saved locally only and can't be read by the server, which eliminates the security issue that cookies present. Everything's done in JavaScript.
- It allows for much more data to be saved (10Mb for most browsers).

- The storage is bound to the origin (domain/protocol/port triplet). That is, different protocols or sub-domains infer different storage objects, they can't access data from each other.

Both the storage objects provide same methods and properties:

- `setItem(key, value)` – store item as key/value pair.
- `getItem(key)` – get item's value by key.
- `removeItem(key)` – remove item using key.
- `clear()` – clear or delete everything.
- `key(index)` – get the Item key based on index.
- `length` – returns total number of stored items.

As you have noticed, it's like a `Map` collection (`setItem/getItem/removeItem`), which also keeps elements order and allows to access by index with `key(index)` .

localStorage

Main features:

- Data is shared between all tabs and windows from the same origin.
- The data will not expire. It will remain even after browser restart and survive OS reboot too.

For example, if you execute this code…

```
localStorage.setItem('localStorage', 1);
```

And close/open the browser or just open the same page in a different window, then the result would be:

```
alert( localStorage.getItem('localStorage') ); // 1
```

We only needs to be on the same origin (domain/port/protocol), the url path can be different.

Since it is shared between all windows with the same origin, hence if we set the data in one window, the change will be visible in another window too.

Work with Strings Only

Both *'key'* and *'value'* values should be string. If we specify any other type, like a number, or an object, it will automatically gets converted to string :

```
sessionStorage.user = {name: "Batman"};
alert(sessionStorage.user); // [object Object]
```

We can use `JSON` to store objects though:

```
sessionStorage.user = JSON.stringify({name: "Batman"});// sometime
later
let user = JSON.parse( sessionStorage.user );
alert( user.name ); // Batman
```

We can also stringify the whole storage object, e.g. for debugging purposes:

```
// used JSON.stringify to make the object look nicer
alert( JSON.stringify(localStorage, null, 2) );
```

Accessing it like Object

We can also use a plain object way of getting/setting keys, like this:

```
// set key
localStorage.object = 2;// get key
alert( localStorage.object); // 2

// remove key
delete localStorage.object;
```

That's allowed for historical reasons, and mostly works. However it is not recommended, due to below reasons:

1. If the key is user-generated, it can be anything, like `length` or `toString`, or another built-in method of `localStorage`. In that case `getItem/setItem` work fine, while object-like access fails:

```
let key = 'length';
localStorage[key] = 5; // Error, can't assign length
```

2. There's a `storage` event, it triggers when we modify the data. That event does not happen for object-like access.

Looping Over Keys

As we've seen, the methods provide "get/set/remove by key" functionality. But how we can get all saved values or keys?

However we can't iterate storage objects. We can either loop over them as an array:

```
for(let i=0; i<localStorage.length; i++) {
  let key = localStorage.key(i);
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

Or we can use `for key in localStorage` loop, just like regular objects.

It iterates over keys, but also outputs few built-in fields that we don't need:

```
// Not gonna work as intended
for(let key in localStorage) {
  alert(key); // shows getItem, setItem and other built-in stuff
}
```

So we wither need to filter fields from the prototype with `hasOwnProperty` check:

```
for(let key in localStorage) {
  if (!localStorage.hasOwnProperty(key)) {
```

```
    continue; // skip keys like "setItem", "getItem" etc
  }
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

Or just get the “own” keys with `Object.keys` and then loop over them if needed:

```
let keys = Object.keys(localStorage);
for(let key of keys) {
  alert(`${key}: ${localStorage.getItem(key)}`);
}
```

The latter works, because `Object.keys` only returns the keys that belong to the object, ignoring the prototype.

sessionStorage

Usage of `sessionStorage` object is much less than `localStorage`.

Properties and methods are the same, however it’s functionality is much more limited:

- The `sessionStorage` exists only within the current browser tab. Another tab with the same page will have a different session storage.
- However it is shared between iframes in the same tab (assuming they come from the same origin).
- The data survives page refresh, but not closing/opening the tab.

for example…

```
sessionStorage.setItem('sessionStorage', 1);
```

After refreshing the page, you can still get the data:

```
alert( sessionStorage.getItem('sessionStorage') );
// after refresh: 1
```

However if you open the same page in another tab, and try again there, the code above returns `null`, meaning "nothing found".

That's exactly because `sessionStorage` is bound not only to the origin, but also to the browser tab. For that reason, `sessionStorage` is used less.

Storage Event

Storage event triggers(with properties), whenever there is any updates in the data in `localStorage` or `sessionStorage`.

- `key` – the key that was changed (`null` if `.clear()` is called).
- `oldValue` – the old value (`null` if the key is newly added).
- `newValue` – the new value (`null` if the key is removed).
- `url` – the url of the document where the update happened.
- `storageArea` – either `localStorage` or `sessionStorage` object where the update happened.

The important thing to note is: the event triggers on all `window` objects where the storage is accessible, except the one that caused it.

Let's see this in more detail.

Suppose, we have two windows with the same site in each. So `localStorage` is shared between them. You might want to open this page in two browser windows to test the code below.

If both windows are listening for `window.onstorage`, then each one will react on updates that happened in the other one.

```
// triggers on updates made to the same storage from other documents
window.onstorage = event => {
  if (event.key != 'now') return;
  alert(event.key + ':' + event.newValue + " at " + event.url);
};

localStorage.setItem('now', Date.now());
```

Please note that the event also contains: `event.url` – the url of the document where the data was updated.

Also, `event.storageArea` contains the storage object – the event is the same for both `sessionStorage` and `localStorage`, so `event.storageArea` references the one that was modified. We may even want to set something back in it, to "respond" to a change.

That allows different windows from the same origin to exchange messages.

Modern browsers also support Broadcast channel API, the special API for same-origin inter-window communication, it's more full featured, but less supported. There are libraries that polyfill that API, based on `localStorage`, that make it available everywhere.

Summary

Web storage objects `localStorage` and `sessionStorage` allows us to store key/value in the browser.

- Both `key` and `value` must be strings.
- The limit is 2mb+, it depends on the browser.
- They do not expire.
- The data is bound to the origin (domain/port/protocol).

LocalStorage()	SessionStorage()	Cookies
5MB/10MB Storage	5MB Storage	4KB Storage
It's not session based ,need to deleted via JavaScript by using clear method of local storage For ex:- localStorage.clear();	It's session based i.e. working per window or tab	Expiry depends upon setting and working per window and also We can set the expiration time for each cookie
only client side reading support	only client side reading	client side reading and server side reading support
less order browser support	less order browser support	more order browser support

JavaScript Web Development Coding Development Programming