20th August 2019

LocalStorage, sessionStorage

Web storage objects `localStorage` and `sessionStorage` allow to save key/value pairs in the browser.

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

We already have cookies. Why additional objects?

- Unlike cookies, web storage objects are not sent to server with each request. Because of that, we can store much more. Most browsers allow at least 2 megabytes of data (or more) and have settings to configure that.
- Also unlike cookies, the server can't manipulate storage objects via HTTP headers. Everything's done in JavaScript.
- The storage is bound to the origin (domain/protocol/port triplet). That is, different protocols or subdomains infer different storage objects, they can't access data from each other.

Both storage objects provide same methods and properties:

- `setItem(key, value)` – store key/value pair.
- `getItem(key)` – get the value by key.
- `removeItem(key)` – remove the key with its value.
- `clear()` – delete everything.
- `key(index)` – get the key on a given position.
- `length` – the number of stored items.

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

Let's see how it works.

localStorage demo

The main features of `localStorage` are:

- Shared between all tabs and windows from the same origin.
- The data does not expire. It remains after the browser restart and even OS reboot.

For instance, if you run this code…

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

…And close/open the browser or just open the same page in a different window, then you can get it like this:

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

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

The `localStorage` is shared between all windows with the same origin, so if we set the data in one window, the change becomes visible in another one.

Object-like access

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

```
// set key
localStorage.test = 2;

// get key
alert( localStorage.test ); // 2

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

That's allowed for historical reasons, and mostly works, but generally not recommended, because:

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. We'll see that later in this chapter.

Looping over keys

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

Unfortunately, storage objects are not iterable.

One way is to loop over them as over an array:

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

Another way is to use `for key in localStorage` loop, just as we do with regular objects.

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

```
// bad try
for(let key in localStorage) {
  alert(key); // shows getItem, setItem and other built-in stuff
}
```

…So we need either 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.

Strings only

Please note that both key and value must be strings.

If were any other type, like a number, or an object, it gets converted to string automatically:

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

We can use `JSON` to store objects though:

```
sessionStorage.user = JSON.stringify({name: "John"});

// sometime later
let user = JSON.parse( sessionStorage.user );
alert( user.name ); // John
```

Also it is possible to stringify the whole storage object, e.g. for debugging purposes:

```
// added formatting options to JSON.stringify to make the object look nice
alert( JSON.stringify(localStorage, null, 2) );
```

sessionStorage

The `sessionStorage` object is used much less often than `localStorage`.

Properties and methods are the same, but it's much more limited:

- The `sessionStorage` exists only within the current browser tab.
  - Another tab with the same page will have a different storage.
  - But 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.

Let's see that in action.

Run this code…

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

…Then refresh the page. Now you can still get the data:

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

…But 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 sparingly.

Storage event

When the data gets updated in `localStorage` or `sessionStorage`, storage event triggers, with properties:

- `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 is: the event triggers on all `window` objects where the storage is accessible, except the one that caused it.

Let's elaborate.

Imagine, you 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` allow to store key/value in the browser.

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

`localStorage`	`sessionStorage`
Shared between all tabs and windows with the same origin	Visible within a browser tab, including iframes from the same origin
Survives browser restart	Survives page refresh (but not tab close)

API:

- `setItem(key, value)` – store key/value pair.
- `getItem(key)` – get the value by key.
- `removeItem(key)` – remove the key with its value.
- `clear()` – delete everything.
- `key(index)` – get the key number `index` .
- `length` – the number of stored items.
- Use `Object.keys` to get all keys.
- We access keys as object properties, in that case `storage` event isn't triggered.

Storage event:

- Triggers on `setItem` , `removeItem` , `clear` calls.
- Contains all the data about the operation (`key/oldValue/newValue`), the document `url` and the storage object `storageArea` .
- Triggers on all `window` objects that have access to the storage except the one that generated it (within a tab for `sessionStorage` , globally for `localStorage`).

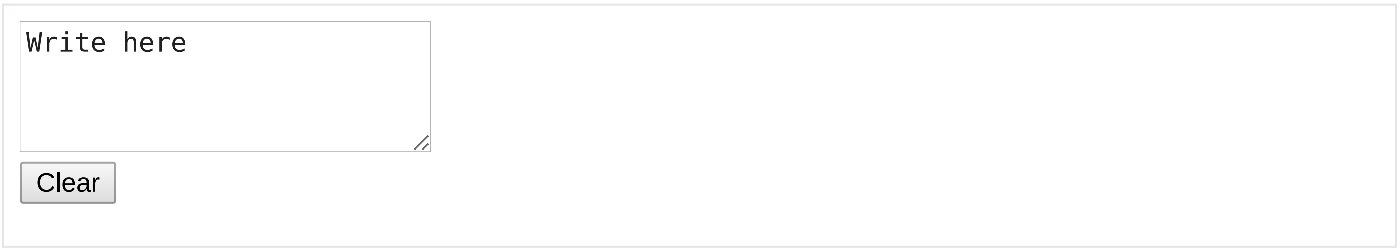
Tasks

Autosave a form field

Create a `textarea` field that “autosaves” its value on every change.

So, if the user accidentally closes the page, and opens it again, he’ll find his unfinished input at place.

Like this:



Open a sandbox for the task.

solution

Comments

- If you have suggestions what to improve - please submit a GitHub issue or a pull request instead of commenting.
- If you can't understand something in the article – please elaborate.
- To insert a few words of code, use the `<code>` tag, for several lines – use `<pre>` , for more than 10 lines – use a sandbox (plnkr, JSBin, codepen…)