Lesson - 21

Storing Data on Client Side

SkillUp by Dobrea Vladislav

Lesson Plan

- Discuss HW
- Continue Implementation of Todo’s
- Storing data on client

Cookies, document.cookie

Cookies are small strings of data that are stored directly in the browser. They are a part of HTTP protocol, defined by RFC 6265 specification.

Cookies are usually set by a web-server using response `Set-Cookie` HTTP-header. Then the browser automatically adds them to (almost) every request to the same domain using `Cookie` HTTP-header.

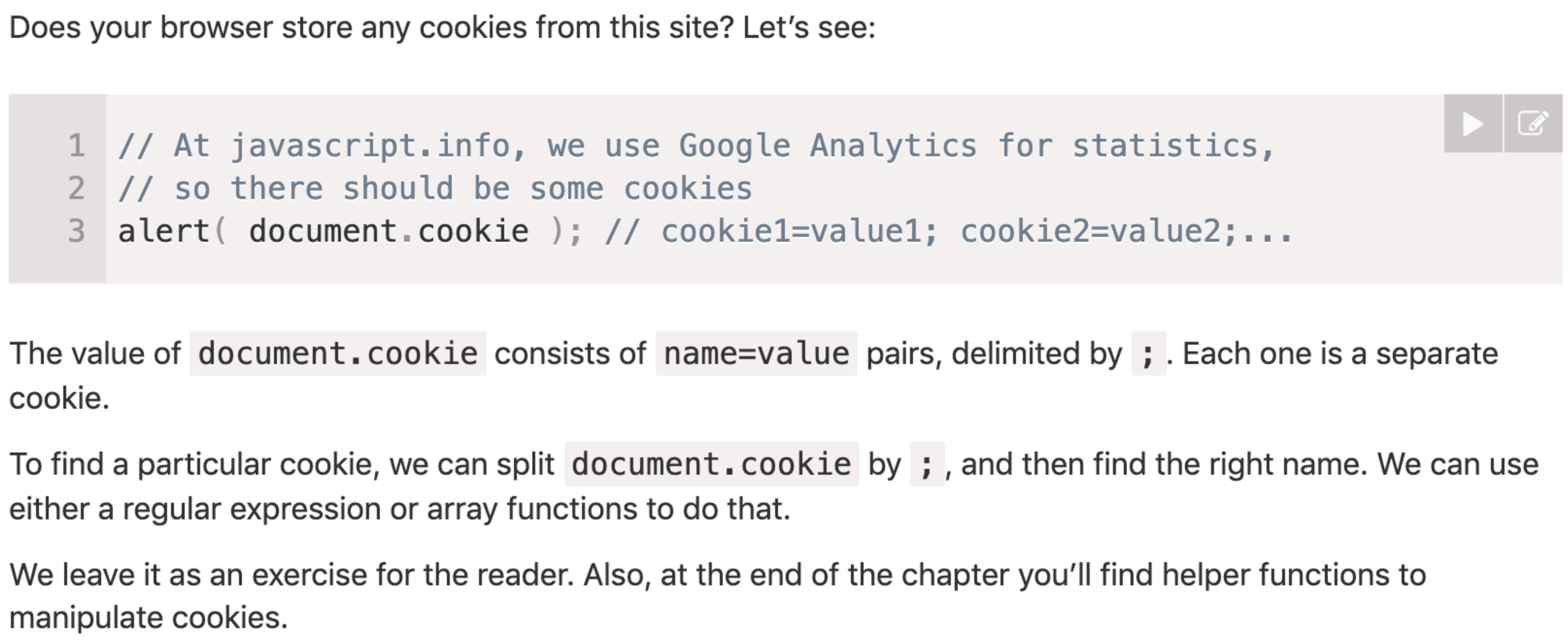
One of the most widespread use cases is authentication:

1. Upon sign in, the server uses `Set-Cookie` HTTP-header in the response to set a cookie with a unique "session identifier".
2. Next time when the request is set to the same domain, the browser sends the cookie over the net using `Cookie` HTTP-header.
3. So the server knows who made the request.

We can also access cookies from the browser, using `document.cookie` property.

There are many tricky things about cookies and their options. In this chapter we'll cover them in detail.

Reading from document.cookie



Does your browser store any cookies from this site? Let's see:

```javascript
// At javascript.info, we use Google Analytics for statistics,
// so there should be some cookies
alert( document.cookie ); // cookie1=value1; cookie2=value2;...
```

The value of `document.cookie` consists of `name=value` pairs, delimited by `;`. Each one is a separate cookie.

To find a particular cookie, we can split `document.cookie` by `;`, and then find the right name. We can use either a regular expression or array functions to do that.

We leave it as an exercise for the reader. Also, at the end of the chapter you'll find helper functions to manipulate cookies.

Writing cookies

We can write to `document.cookie`. But it's not a data property, it's an accessor (getter/setter). An assignment to it is treated specially.

A write operation to `document.cookie` updates only cookies mentioned in it, but doesn't touch other cookies.

For instance, this call sets a cookie with the name `user` and value `John`:

```javascript
document.cookie = "user=John"; // update only cookie named 'user'
alert(document.cookie); // show all cookies
```

If you run it, then probably you'll see multiple cookies. That's because `document.cookie=` operation does not overwrite all cookies. It only sets the mentioned cookie `user`.

Technically, name and value can have any characters, to keep the valid formatting they should be escaped using a built-in `encodeURIComponent` function:

```javascript
// special characters (spaces), need encoding
let name = "my name";
let value = "John Smith"

// encodes the cookie as my%20name=John%20Smith
document.cookie = encodeURIComponent(name) + '=' + encodeURIComponent(value);

alert(document.cookie); // ...; my%20name=John%20Smith
```

Cookies structure

Cookies have several options, many of them are important and should be set.

The options are listed after `key=value`, delimited by `;`, like this:

```
document.cookie = "user=John; path=/; expires=Tue, 19 Jan 2038 03:14:07 GMT"
```

path

- **`path=/mypath`**

The url path prefix, the cookie will be accessible for pages under that path. Must be absolute. By default, it's the current path.

If a cookie is set with `path=/admin`, it's visible at pages `/admin` and `/admin/something`, but not at `/home` or `/adminpage`.

Usually, we should set `path` to the root: `path=/` to make the cookie accessible from all website pages.

Cookies structure

domain

- `domain=site.com`

A domain where the cookie is accessible. In practice though, there are limitations. We can't set any domain.

By default, a cookie is accessible only at the domain that set it. So, if the cookie was set by `site.com`, we won't get it `other.com`.

...But what's more tricky, we also won't get the cookie at a subdomain `forum.site.com`!

```javascript
// at site.com
document.cookie = "user=John"

// at forum.site.com
alert(document.cookie); // no user
```

There's no way to let a cookie be accessible from another 2nd-level domain, so `other.com` will never receive a cookie set at `site.com`.

Cookies structure

expires, max-age

By default, if a cookie doesn't have one of these options, it disappears when the browser is closed. Such cookies are called "session cookies"

To let cookies survive browser close, we can set either `expires` or `max-age` option.

- **`expires=Tue, 19 Jan 2038 03:14:07 GMT`**

Cookie expiration date, when the browser will delete it automatically.

The date must be exactly in this format, in GMT timezone. We can use `date.toUTCString` to get it. For instance, we can set the cookie to expire in 1 day:

```
// +1 day from now
let date = new Date(Date.now() + 86400e3);
date = date.toUTCString();
document.cookie = "user=John; expires=" + date;
```

If we set `expires` to a date in the past, the cookie is deleted.

- **`max-age=3600`**

An alternative to `expires`, specifies the cookie expiration in seconds from the current moment.

If zero or negative, then the cookie is deleted:

```
// cookie will die +1 hour from now
document.cookie = "user=John; max-age=3600";

// delete cookie (let it expire right now)
document.cookie = "user=John; max-age=0";
```

Cookies functions

getCookie(name)

The shortest way to access cookie is to use a regular expression.

The function `getCookie(name)` returns the cookie with the given `name`:

```
// returns the cookie with the given name,
// or undefined if not found
function getCookie(name) {
  let matches = document.cookie.match(new RegExp(
    "(?:^|; )" + name.replace(/([\.$?*|{}\(\)\[\]\\\/\+^])/g, '\\$1') + "=([^
  ));
  return matches ? decodeURIComponent(matches[1]) : undefined;
}
```

Here `new RegExp` is generated dynamically, to match `; name=<value>`.

Please note that a cookie value is encoded, so `getCookie` uses a built-in `decodeURIComponent` function to decode it.

Cookies functions

setCookie(name, value, options)

Sets the cookie `name` to the given `value` with `path=/` by default (can be modified to add other defaults):

```
function setCookie(name, value, options = {}) {

  options = {
    path: '/',
    // add other defaults here if necessary
    ...options
  };

  if (options.expires instanceof Date) {
    options.expires = options.expires.toUTCString();
  }

  let updatedCookie = encodeURIComponent(name) + "=" + encodeURIComponent(val

  for (let optionKey in options) {
    updatedCookie += "; " + optionKey;
    let optionValue = options[optionKey];
    if (optionValue !== true) {
      updatedCookie += "=" + optionValue;
    }
  }

  document.cookie = updatedCookie;
}

// Example of use:
setCookie('user', 'John', {secure: true, 'max-age': 3600});
```

Summary

`document.cookie` provides access to cookies

- write operations modify only cookies mentioned in it.
- name/value must be encoded.
- one cookie up to 4kb, 20+ cookies per site (depends on a browser).

Cookie options:

- `path=/`, by default current path, makes the cookie visible only under that path.
- `domain=site.com`, by default a cookie is visible on current domain only, if set explicitly to the domain, makes the cookie visible on subdomains.
- `expires` or `max-age` sets cookie expiration time, without them the cookie dies when the browser is closed.
- `secure` makes the cookie HTTPS-only.
- `samesite` forbids the browser to send the cookie with requests coming from outside the site, helps to prevent XSRF attacks.

Additionally:

- Third-party cookies may be forbidden by the browser, e.g. Safari does that by default.
- When setting a tracking cookie for EU citizens, GDPR requires to ask for permission.

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.

LocalStorage, SessionStorage

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 allows access by index with `key(index)`.

LocalStorage

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.

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:

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

We can use `JSON` to store objects though:

```javascript
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:

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

Session Storage

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.

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.

Learning Resources

1. **Cookies: https://javascript.info/cookie**
2. **Local Storage and Session Storage: https://javascript.info/localstorage**