Client-Side Storage

Cookies and Web Storage

Asst.Prof. Dr. Umaporn Supasitthimethee

ผศ.ดร.อุมาพร สุภสิทธิเมธี

https://developer.mozilla.org/en-US/docs/Learn/JavaScript/Client-side_web_APIs/Client-side_storage
https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie
https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API

Client-Side Storage

- Client-side storage works on similar principles but has different uses. It consists of JavaScript APIs that allow you to store data on the client (i.e., on the user's machine) and then retrieve it when needed. This has many distinct uses, such as:
    - Personalizing site preferences (e.g., showing a user's choice of custom widgets, color scheme, or font size).
    - Persisting previous site activity (e.g., storing the contents of a shopping cart from a previous session, remembering if a user was previously logged in).
    - Saving data and assets locally so a site will be quicker (and potentially less expensive) to download or be usable without a network connection.
    - Saving web application generated documents locally for use offline

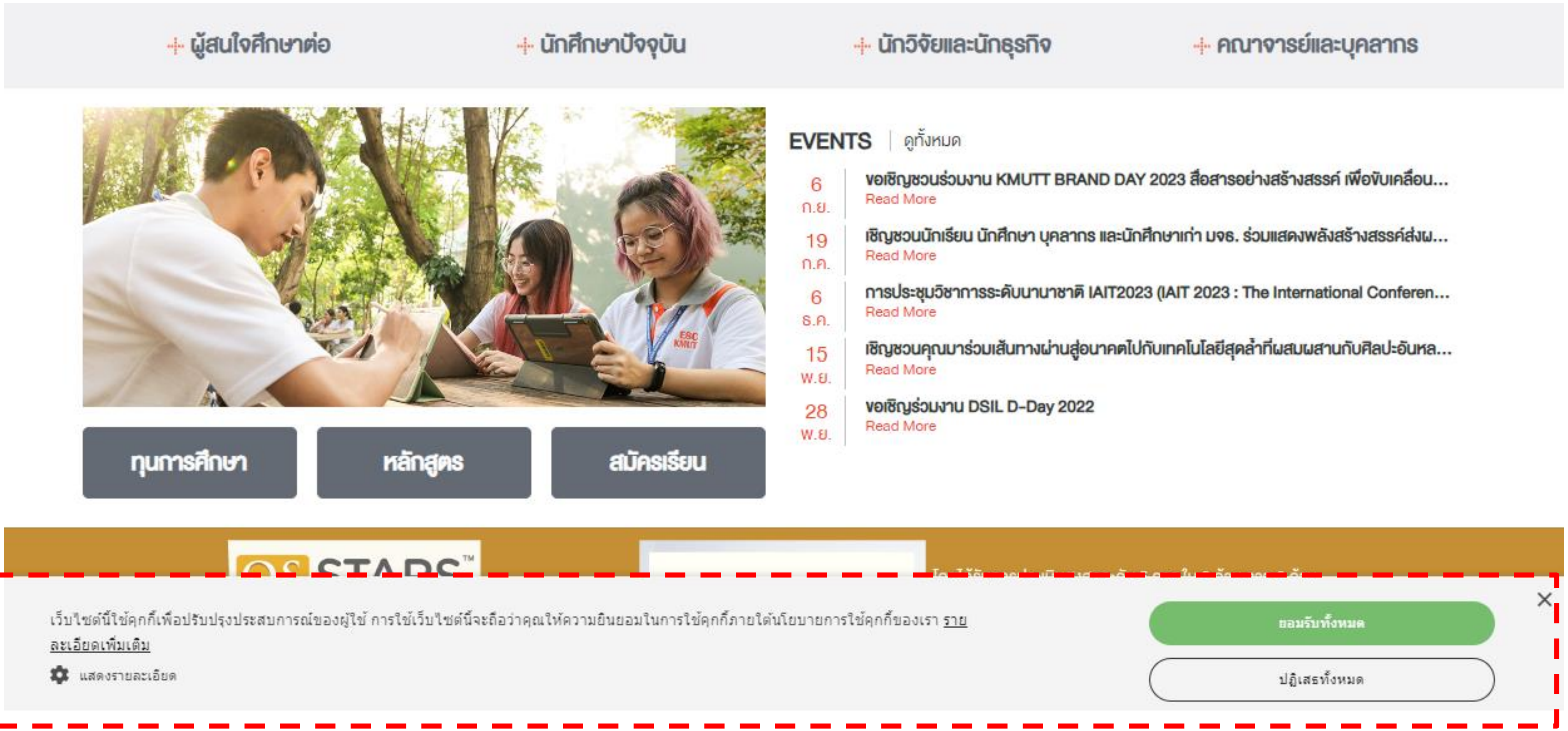
Client-Side Storage

- **HTTP Cookies**
  - They're the earliest form of client-side storage commonly used on the web.
  - They are still used commonly to store data related to user personalization and state, e.g., session IDs and access tokens.
- **Web Storages**
  - The Web Storage API provides a very simple syntax for storing and retrieving smaller, data items consisting of a name and a corresponding value.
  - This is useful when you just need to store some simple data, like the user's name, whether they are logged in, what color to use for the background of the screen, etc.
- IndexedDB
  - IndexedDB is a way for you to persistently store data inside a user's browser. It provides a solution for storing larger amounts of structured data.

Cookies vs. Session Storage vs. Local Storage

Cookies	Session Storage	Local Storage
Has different expiration dates depending on set up expiration date	Data is gone when you close the browser or tab	Has no expiration date
Client and Server	Client only	Client only
Data are transferred on each HTTP request	Data are not transferred on each HTTP request	Data are not transferred on each HTTP request
4 kb limit	5 mb limit	5 mb limit
can be accessed by third parties		

Cookies



EVENTS | ดูทั้งหมด

6 ก.ย.	ขอเชิญชวนร่วมงาน KMUTT BRAND DAY 2023 สื่อสารอย่างสร้างสรรค์ เพื่อขับเคลื่อน... Read More
19 ก.ค.	เชิญชวนนักเรียน นักศึกษา บุคลากร และนักศึกษาเก่า มจธ. ร่วมแสดงพลังสร้างสรรค์ส่งผ... Read More
6 ธ.ค.	การประชุมวิชาการระดับนานาชาติ IAIT2023 (IAIT 2023 : The International Conferen... Read More
15 พ.ย.	เชิญชวนคุณมาร่วมเส้นทางผ่านสู่อนาคตไปกับเทคโนโลยีสุดล้ำที่ผสมผสานกับศิลปะอันหล... Read More
28 พ.ย.	ขอเชิญร่วมงาน DSIL D-Day 2022 Read More

เว็บไซต์นี้ใช้คุกกี้เพื่อปรับปรุงประสบการณ์ของผู้ใช้ การใช้เว็บไซต์นี้จะถือว่าคุณให้ความยินยอมในการใช้คุกกี้ภายใต้นโยบายการใช้คุกกี้ของเรา รายละเอียดเพิ่มเติม

แสดงรายละเอียด

Introduction to Cookies

- **HTTP cookies (web cookies, browser cookies)**, commonly just called **cookies**, is a small piece of data that a server sends to the user's web browser.
- **Cookies** are a general mechanism which server-side connections can use to **both store and retrieve information** on the client side of the connection.
- The browser may store it and send it back with later requests to the same server. Cookies are sent with every request, so they can **worsen performance** (especially for mobile data connections).
- Typically, it's used to tell if two requests came from the same browser — keeping a user logged-in, for example. It remembers stateful information for the stateless HTTP protocol.
- The addition of a simple, persistent, client-side state significantly extends the capabilities of Web-based client/server applications.

The specification of the minimum number of cookie

- Cookies are stored on the client computer, In general, if you use the following approximate limits, you will run into no problems across all browser traffic:
  - 300 cookies total
  - 4096 bytes per cookie
  - 20 cookies per domain
  - 81920 bytes per domain
- For best cross-browser compatibility, it's best to keep the total cookie size to **4095 bytes or less**. The size limit applies to all cookies for a domain.
- When a cookie larger than 4 kilobytes is encountered the cookie should be trimmed to fit, but the name should remain intact as long as it is less than 4 kilobytes.
- Servers should not expect clients to be able to exceed these limits. When limit is exceeded, **clients should delete the least recently used cookie.**

Syntax of the Set-Cookie HTTP Response Header

```
set-Cookie: NAME=VALUE; expires=DATE; path=PATH; domain=DOMAIN_NAME; secure; HttpOnly
```

- ***NAME=VALUE***: This string is a sequence of characters excluding semi-colon, comma and white space
- **expires=*DATE***: The expires attribute specifies a date string that defines the valid lifetime of that cookie. The date string is formatted as: `Wdy, DD-Mon-YYYY HH:MM:SS GMT.` The default will expire at the end of session (when browser is closed).
- **domain=*DOMAIN_NAME***: specifies which hosts are allowed to receive the cookie. (e.g., 'example.com' or 'subdomain.example.com'). If not specified, this defaults is on current access domain, not including subdomain.
- **path=*PATH:*** (e.g., '/', '`/mydir`') The path `"/foo"` would match `"/foo/bar.html".` The path `"/"` is the most general path. By default, a cookie is available to all web pages in the same directory or any subdirectories of that directory.
- **secure:** a cookie is sent to the server only with an encrypted request over the HTTPS protocol, never with unsecured HTTP (except on localhost). By default, a cookie also appears on secure and unsecure site
- **HttpOnly:** If the HttpOnly flag is included in the HTTP response header, the cookie cannot be accessed through the client-side script. We can’t see such a cookie or manipulate it using `document.cookie`.

Cookie on HTTP Examples

1. HTTP Response:
set-cookie: CUSTOMER=Somchai; path=/; expires=Wed, 14-Dec-2023 23:12:40 GMT

2.When client requests a URL in path "/" on this server, it sends:
cookie: CUSTOMER= Somchai;

3. HTTP Response:
set-cookie: PART_NUMBER=COMP_001; path=/

4. When client requests a URL in path "/" on this server, it sends:
cookie: CUSTOMER= Somchai; PART_NUMBER=COMP_001

5. HTTP Response:
set-cookie: SHIPPING=FEDEX; path=/foo

6. When client requests a URL in path "/" on this server, it sends:
cookie: CUSTOMER= Somchai; PART_NUMBER=COMP_001

7.When client requests a URL in path "/foo" on this server, it sends:
cookie: CUSTOMER= Somchai; PART_NUMBER=COMP_001; SHIPPING=FEDEX

Syntax of the `set-cookie` HTTP Response Header

- The `set-cookie` HTTP response header sends cookies from the server to the user agent. For instance, **the headers of a server response** may look like this:

```
HTTP/2.0 200 OK
Content-Type: text/html
set-cookie: school=KMUTT
set-cookie: program=IT

[page content]
```

```
HTTP/1.1 200 OK
Content-type: text/html
set-Cookie: name=value;
expires=Mon, 22-Dec-2023 00:00:00 GMT;
domain=example.com
[other-header: other-header-value]
```

- This HTTP response sets a cookie with the name of "*name*" and a value of "*value*".

```
set-cookie: cookie-name=cookie-value
```

Syntax of the Cookie HTTP Request Header

- Browsers store such session information and **send it back to the server** via the **Cookie HTTP header** for every request after that point, such as the following:

```
GET /sample_page.html
Host: example.com
cookie: school=KMUTT; program=IT
[other-header: other-header-value]
```

- This extra information being sent back to the server can be used to **uniquely identify the client** from which the request was sent.

Restrictions

- Cookies are, by nature, **tied to a specific domain**.
- When a cookie is set, it is sent along **with requests to the same domain** from which it was created.
- This restriction ensures that information stored in cookies is available only to approved recipients and **cannot be accessed by other domains.**

Cookie Types

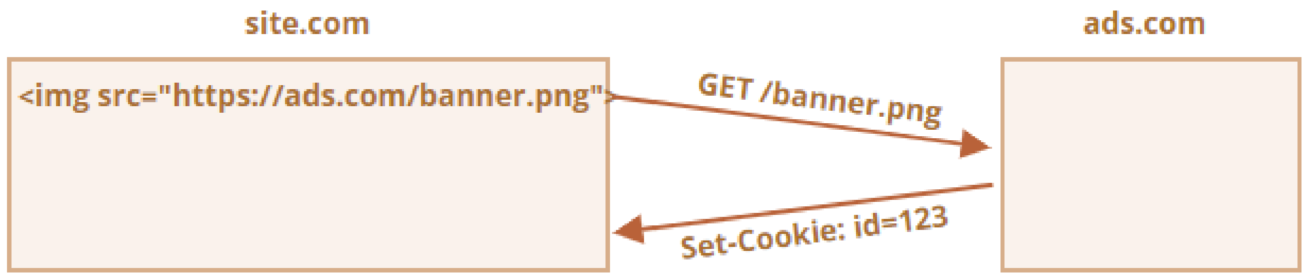
- **First-party cookies** are created by **the host domain** – the domain the user is visiting. These types of cookies are generally considered good; Most browsers accept first-party cookies by default, as their primary role is to allow customization and improve user experience.
- **Third-party cookies** are created **by domains other than the one you are visiting directly,** hence the name third-party. They are used for cross-site tracking, Ad-Retargeting Services, analytics and tracking services.

https://clearcode.cc/blog/difference-between-first-party-third-party-cookies/

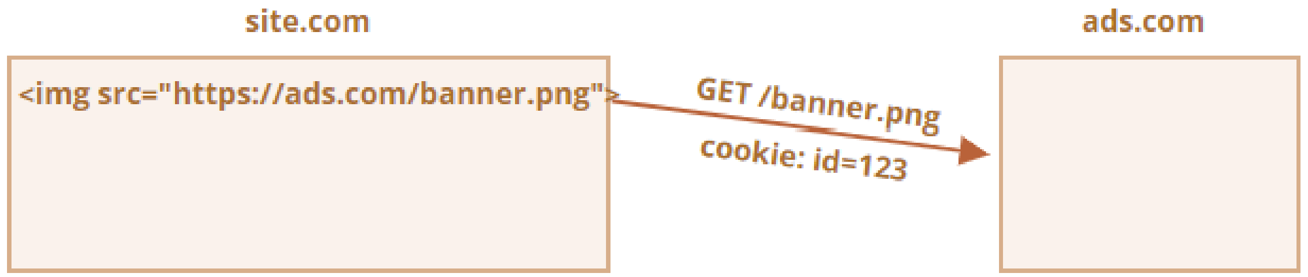
Third-Party Cookies Example

1. A page at **site.com** loads a banner from **another site: <img src="https://ads.com/banner.png">**.

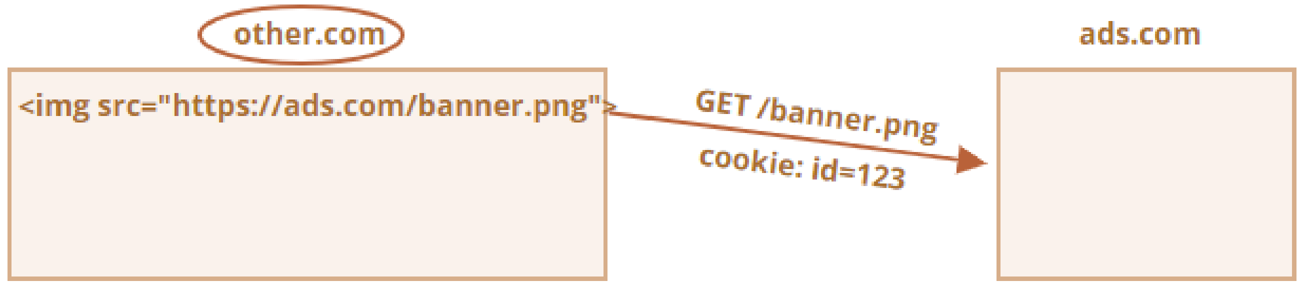
2. Along with the banner, the remote server at **ads.com may set the Set-Cookie header** with a cookie like id=1234.
Such a cookie originates from the ads.com domain, and will only be visible at ads.com:



3. Next time when **ads.com** is accessed, the remote server **gets the id cookie and recognizes the user**:



4. when the **user moves from site.com to another site other.com**, which also **has a banner, then ads.com gets the cookie,** as it belongs to ads.com, thus recognizing **the visitor and tracking him as he moves between sites**:



https://javascript.info/cookie

Cookies in JavaScript

- The document cookie, `document.cookie`, lets you read and write cookies associated with the document.
- Dealing with cookies in JavaScript is a little complicated because of a notoriously poor interface.
- When used to retrieve the property value, `document.cookie` returns a string of all cookies available to the page (based on the domain, path, expiration, and security settings of the cookies) as a series of name-value pairs separated by semicolons, as in the following example:

```
name1=value1;name2=value2;name3=value3
```

Cookies in JavaScript

- A writing cookie to `document.cookie` will update only cookie mentioned in it, but does not affect other cookies.
- Setting `document.cookie` does not overwrite any cookies **unless the name of the cookie being set is already in use**.
- You can access existing cookies from JavaScript as well if the `HttpOnly` flag isn't set.
- The format to set a cookie is as follows, and is the same format used by the set-cookie header:

only the cookie's name and value are required

```
name=value; expires=expiration_time (or max-age=n seconds); path=domain_path;
domain=domain_name; secure; HttpOnly
```

```
document.cookie = "username=Umaporn;domain=example.com;path=/"
```

```
document.cookie = 'member=Umaporn;max-age=60'
```

`encodeURIComponent()`

- The cookie value string can use `encodeURIComponent()` to ensure that the string does not contain any commas, semicolons, or whitespace (which are disallowed in cookie values).
- Although there are no characters that need to be encoded in either the name or the value, it's a best practice to always use `encodeURIComponent()`

```
document.cookie = encodeURIComponent("username") + "=" +
encodeURIComponent("Umaporn") + "; domain=example.com; path=/"
```

`encodeURIComponent()`

- The `encodeURIComponent()` function encodes a URI component by representing the UTF-8 encoding of the character.
- This function encodes special characters `; , / ? : @ & = + $ #`
- The characters `A-Z a-z 0-9 - _ . ! ~ * ' ( )` are not escaped.
- Use the `decodeURIComponent()` function to decode an encoded URI component.

```
// encodes characters such as ?,=,/,&,:
console.log(`?x=${encodeURIComponent("test?/&")}`)
// expected output: ?x=test%3F%2F%26
console.log(`?x=${decodeURIComponent("test?/&")}`)
// expected output:?x=test?/&

const url = "https://www.sit.kmutt.ac.th"
console.log(`${encodeURIComponent(url)}`)
//https%3A%2F%2Fwww.sit.kmutt.ac.th
console.log(`${decodeURIComponent(url)}`)
// expected output: https://www.sit.kmutt.ac.th
```

Basic Cookie Operations

- There are three basic cookie operations: **reading, writing, and deleting.**

```
class CookieUtil {
   static get(name) {
     let cookieName = `${encodeURIComponent(name)}=`,
     cookieStart = document.cookie.indexOf(cookieName),
     cookieValue = null
     if (cookieStart > -1){
       let cookieEnd = document.cookie.indexOf(";", cookieStart)
       if (cookieEnd === -1){
           cookieEnd = document.cookie.length
       }
     cookieValue = decodeURIComponent(document.cookie.substring(cookieStart+
     cookieName.length, cookieEnd))
     }
     return cookieValue
  }
```

Basic Cookie Operations

```
static set(name, value, expires) {
   let cookieText = `${encodeURIComponent(name)}=${encodeURIComponent(value)}`
   if (expires instanceof Date) {
     cookieText += `;expires=${expires.toISOString()}`
   }
   document.cookie = cookieText
}
```

```
//to remove existing cookies, setting the cookie again—with the same path,
domain, and secure options—and set its expiration date to some time in the past.
 static unset(name) {
//set to a blank string and the expiration date set to January 1, 1970 (the
value of a Date object initialized to 0 milliseconds).
   CookieUtil.set(name, "", new Date(0)) //or max-age=0
  }
}//ending class
```

Web Storages

Web Storages

- The **Web Storage API** provides mechanisms by which browsers can store key/value pairs, in a much more intuitive fashion than using cookies.
- The **Storage** interface of the Web Storage API provides access to a particular domain's session or local storage. It allows, for example, the addition, modification, or deletion of stored data items.
- These mechanisms are available via the **Window.sessionStorage** and **Window.localStorage** properties invoking one of these will create an instance of the Storage object, through which data items can be set, retrieved and removed.
- A different Storage object is used for the **sessionStorage** and **localStorage** for each origin — they function and are controlled separately.

Web Storage concepts and usage

- **sessionStorage** stores information in the user's browser ***for a single session***
  - Stores data only for a session, meaning that the data is stored *until the browser (or tab) is closed.* Opening a new tab of the same page will start a new browser session.
  - Data is never transferred to the server.
  - Storage limit is larger than a cookie (5 MB, check with the browser).
- **localStorage** does the same thing, but across sessions, ***persists even when the browser is closed and reopened***.
  - Stores data with **no expiration date**, and gets cleared only through JavaScript, or clearing the Browser cache / Locally Stored Data.
  - Storage limit is larger than a cookie (5MB, check with the browser).

https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API

`setItem(keyName, keyValue)` and `getItem(keyName)`

When they work differently?

localStorage vs. **sessionStorage**

```
let visit = localStorage.getItem('visit')
if (visit === null) {
  localStorage.setItem('visit', 1)
} else {
  localStorage.setItem('visit', ++visit)
}

alert(`visit: ${localStorage.getItem('visit')}`)
```

```
let visit = sessionStorage.getItem('visit')
if (visit === null) {
  sessionStorage.setItem('visit', 1)
} else {
  sessionStorage.setItem('visit', ++visit)
}

alert(`visit: ${sessionStorage.getItem('visit')}`)
```

The **setItem()** method of the Storage interface, when passed a key name and value, will add that key to the given Storage object, or update that key's value if it already exists.

The **getItem()** – A string containing the value of the key. If the key does not exist, `null` is returned.

removeItem(*keyName*) and clear()

localStorage vs. sessionStorage

```
localStorage.setItem('bgcolor', 'red’)
localStorage.setItem('font', 'Helvetica’)
localStorage.setItem('image', 'myCat.png’)
localStorage.removeItem('image’)
// localStorage.clear()
```

```
sessionStorage.setItem('bgcolor', 'red’)
sessionStorage.setItem('font', 'Helvetica’)
sessionStorage.setItem('image', 'myCat.png’)
sessionStorage.removeItem('image’)
// sessionStorage.clear()
```

The **clear()** method of the Storage interface clears all keys stored in a given Storage object.

The **removeItem()** method of the Storage interface, when passed a key name, will remove that key from the given Storage object if it exists. If there is no item associated with the given key, this method will do nothing.