The new Cookie Store API

14 April 2025

by **Fotis Papadogeorgopoulos**

Historically, in a browser, programmatic access to cookies has revolved around the document.cookie *(https://developer.mozilla.org/en-US/docs/Web/API/Document/cookie)* API, which is simply a string getter/setter. To read a single cookie from that string, you must parse the whole string into a structured format. To write a single cookie to that string, you would have the cookie in a structured format and serialise it to a string, prior to setting `document.cookie`. There are popular npm libraries to make this dance more ergonomic, such as `js-cookie` *(https://www.npmjs.com/package/js-cookie)*.

However, custom libraries still come at a performance cost. There are still parsing and serialisation costs to pay, and my understanding is that these costs increase as the cookie string becomes larger. I would also note that `document.cookie` access is *synchronous*. If it ends up in a particularly hot loop, it could impact main thread responsiveness.

Finally, there has been no single way to keep the cookie string as the source of truth for managing state, because there are no associated change events. This becomes relevant in real-world applications, that might be syncing various system-wide and app-specific user preferences via cookies.

While this has been true historically, there is a new Cookie Store API *(https://developer.mozilla.org/en-US/docs/Web/API/Cookie_Store_API)* in

browsers nowadays, that allows structured manipulation of cookies, in a performant way. I say “new”, even though it has existed in Chromium-based browsers for the past 4 years. However, with Safari 18.4 *(https://developer.apple.com/documentation/safari-release-notes/safari-18_4-release-notes#:~:text=Added%20support%20for%20the%20Cookie%20Store%20API)* landing support at the end of March 2025, there are now two browsers shaping this API, making me more comfortable using it as a progressive enhancement.

Firefox 137 *(https://github.com/mdn/browser-compat-data/issues/26374)* also seems to support this API, but I was not able to use `cookieStore` in my tests on Firefox 137.0.1 (aarch64). It might be landing in Firefox 138 instead.

Here is the caniuse support table, for reference:

Can I Use cookie-store-api? Data on support for the cookie-store-api feature across the major browsers. (Embed Loading)

I see three main benefits to such an API:

1. performance (asynchronous access, less serialisation back-and-forth, smaller bundle size)
2. using the cookie store directly for state management, avoiding synchronisation bugs
3. standardisation as a way to build shared (cookie) SDKs with less per-library variance.

Let's dive into each of these!

How to use it

MDN has the goods when it comes to documentation *(https://developer.mozilla.org/en-US/docs/Web/API/Cookie_Store_API)*, but here is a brief example of using cookieStore *(https://developer.mozilla.org/en-US/docs/Web/API/CookieStore)*:

```
async function setDarkModePreference() {
  try {
    await cookieStore.set('theme', 'dark');
  } catch (err) {
    // a nice benefit of the Cookie Store API is immediate
errors,
    // instead of needing to round-trip writing and reading
  }
}

async function resetThemePreference() {
```

```
  return cookieStore.delete('theme');
}
```

Note that all the methods of the `CookieStore` interface are asynchronous.

Part 1: Performance

The string-based interface of `document.cookie` is awkward to work with, with a fair amount of back-and-forth from strings to more structured representations. Parsing the cookie string is not trivial, but we can assume that shared ecosystem libraries follow the spec, and are well-tested and fuzzed.

Libraries can make optimisations, such as exiting early when they find the relevant cookie in the string. I am a fan of the `js-cookie` library, plus their source code makes for a nice read *(https://github.com/js-cookie/js-cookie/blob/main/src/api.mjs)*.

Even with optimisations, access to `document.cookie` is synchronous. If it ends up in a particularly hot loop, it could impact main thread responsiveness.

There is also the impact to the bundle size of an application, though `js-cookie` specifically is tiny, at 780B min+gzip *(https://bundlephobia.com/package/js-cookie@3.0.5)*.

Overall, a native, async-first way to manipulate cookies would make for a nice performance improvement, even compared to optimised ecosystem libraries.

Part 2: Cookie State Management

While I am a fan of the asynchronous access and the reduced bundle size, I am an even bigger fan of the 'change' even that Cookie Store provides:

```
cookieStore.addEventListener('change', (event) => {
  console.log(event);
});
```

Here is the reference for CookieChangeEvent *(https://developer.mozilla.org/en-US/docs/Web/API/CookieChangeEvent/CookieChangeEvent)*.

Imagine we have a web app that allows users to store their site-specific theme preference: `light`, `dark` and `auto` (meaning that it defers to system settings). When the user selects their preference, we want to persist it in a cookie, so that subsequent visits from the server get the correct theme by default. At the same time, different parts of the application might consult the theme mode in JS. How would we ensure that we have a single source of truth?

Since different parts of the application are interested in the theme information, that state has to be kept somewhere centrally. Ideally, we would have the browser manage the state, since the browser keeps the cookies in the first place. However, the `document.cookie` API does not expose a change event, so any part of the UI other than the one making the change would not be able to react to it.

The next best thing we could do is to manage a mirror of the state ourselves, and make changes through a single blessed setter. In React

terms, we could keep a custom Context, either for all cookie values or for a specific one. Here is a sketch for a theme cookie specifically:

```
import Cookie from 'js-cookie';
import type { PropsWithChildren } from 'react';
import {
  useCallback,
  createContext,
  useContext,
  useEffect,
  useMemo,
  useState,
} from 'react';

const THEME_COOKIE_NAME = 'theme';

type Theme = 'light' | 'dark' | 'auto';

// Like a setState tuple
type ContextValue = [Theme, (theme: Theme) => void];

const ThemeContext = createContext<ContextValue | undefined>
(undefined);

export function useThemeState() {
  const ctx = useContext(ThemeContext);
  if (!ctx) {
    throw new Error('useThemeState must be used in a
ThemeManager tree');
  }
  return ctx;
}
```

```
export function ThemeManager({ children }: PropsWithChildren)
{
  const [theme, setStoredTheme] = useState<Theme>();

  useEffect(() => {
    // NOTE: this is incomplete; in reality you might use a
one-off useSyncExternalStore and should also do validation
    setStoredTheme(Cookie.get(THEME_COOKIE_NAME));
  }, [setStoredTheme]);

  const setTheme = useCallback(
    (theme: Theme) => {
      Cookie.set(THEME_COOKIE_NAME, theme);
      setStoredTheme(theme);
    },
    [setStoredTheme]
  );

  const memoValue = useMemo(() => [theme, setTheme], [theme,
setTheme]);

  return (
    <ThemeContext.Provider value={memoValue}>{children}
</ThemeContext.Provider>
  );
}
```

Modulo some implementation details, this works! After the initial setup, I do not find the mirroring to be a big deal (the React state has to live *somewhere* after all), but the opportunity for state to go out of sync worries me from time to time.

We can now do a bit better, by using the change event. Since there is a complete set of APIs to read, write and be notified of changes, we could use React's built-in syncing mechanisms:

```
import { useCallback, useEffect, useState } from 'react';

const THEME_COOKIE_NAME = 'theme';

type Theme = 'light' | 'dark' | 'auto';

export function useThemeState() {
  // An SSR/hydration snapshot is left as an exercise for the
reader
  const [theme, setStoredTheme] = useState<Theme | undefined>
(undefined);

  const onChange = useCallback(
    (ev: CookieChangeEvent) => {
      const deleted = ev.deleted.find((c) => c.name ===
THEME_COOKIE_NAME);

      if (deleted) {
        setStoredTheme(undefined);
        return;
      }

      const changed = ev.changed.find((c) => c.name ===
THEME_COOKIE_NAME);

      if (changed) {
        setStoredTheme(changed.value);
        return;
```

```
      }
    },
    [setStoredTheme]
  );

  useEffect(() => {
    cookieStore.addEventListener('change', onChange);
    return () => {
      cookieStore.removeEventListener('change', onChange);
    };
  }, [onChange]);

  const setTheme = useCallback((theme: Theme) => {
    cookieStore.set(THEME_COOKIE_NAME, theme);
  }, []);

  return [theme, setTheme];
}
```

This allows us to get rid of the context, using cookieStore directly. I initially wanted to use useSyncExternalStore *(https://react.dev/reference/react/useSyncExternalStore)*, as that would allow ditching the extra `useState` altogether. However, `useSyncExternalStore` would not take the new value from the change event, but rather from a separate `cookieStore.get` call. But since the latter is asynchronous, this does not match the sync requirement of the snapshot function that React expects.

(A harebrained idea would be to use the synchronous `document.cookie` API for the `useSyncExternalStore` snapshot, but ehh, I'm not a big fan of mixing the two APIs, given the interop and performance points.)

Intermission: Adopting CookieStore API Today

You might be wondering how you can use this API if it is not supported everywhere yet.

As shown in the theme mode example, it is likely that your application already has a centralised point that mirrors cookie state, and changes are usually tied to user actions. In those cases, you could use the `CookieStore` API where available, falling back to your favourite JS userland implementation (like `js-cookie`) plus local state management.

```
// It seems reasonable to assume that support for the
CookieStore API
// is a property of the environment, and we do not need to
react dynamically to it :)
const supportsCookieStore = 'cookieStore' in window;

const ThemeManagerUserland = ({ children }) => {
  // Does internal book-keeping of the state, as shown
previously
  return <ThemeProvider value={memoValue}>{children}
</ThemeProvider>;
};

const ThemeManagerNative = ({ children }) => {
  // Keeps state backed by cookieStore change events, and
passes on the setter
  return <ThemeProvider value={memoValue}>{children}
</ThemeProvider>;
};

export const ThemeManager = supportsCookieStore
```

```
    ? ThemeManagerNative
    : ThemeManagerUserland;
```

You might decide to split the hook or fork the logic internally, instead of splitting the providers. It really is up to you and your preferences / what is idiomatic in your application.

For one-off manipulation of cookies, you can still fall back to a userland implementation, but bear in mind that your call-sites might have to change to be asynchronous, given that the CookieStore API is asynchronous all the way,

```
import { setCookie as setCookieFallback } from 'my-favorite-
cookie-library';

export async function setCookie(cookie: string, value:
string) {
  if ('cookieStore' in window) {
    return cookieStore.set(cookie, value);
  }
  setCookieFallback(cookie, value);
  return;
}
```

Feel free to make this interface whatever you wish; I made it `(string, string)` for the sake of a simple example. The full Cookie Store API also provides a more structured way to set cookies than just a key and a value, via an option argument *(https://developer.mozilla.org/en-US/docs/Web/API/CookieStore/set#options)*:

```
cookieStore.set({
  name: 'theme',
  value: 'dark',
  path: '/',
  partitioned: false,
  sameSite: 'strict',
});
```

You would tweak your fallback function interface to account for this form as well, if you need it.

There are no TypeScript types for the Cookie Store API yet. TypeScript only adds an API to the shared dom types once it is supported by all major browsers, which I find reasonable. Now that Chrome, Safari and Firefox support the API, I'm hoping for the types to be added soon. A good opportunity for an issue and a PR, perhaps...

Part 3: Standardisation

Finally, one of the last benefits of standardisation, is the process of standardising itself! You might have a family of websites under a shared domain, that all manipulate the same set of cookies. You might decide to make an SDK to share that logic, for example to always validate the cookies values.

If you use a userland library for this task, you will have to make decisions about the encoding of values (or have the library make them for you). In turn, this might hurt interop with other libraries. For example, `js-cookie` spells this out:

> This project is RFC 6265 *(http://tools.ietf.org/html/rfc6265#section-4.1.1)* compliant. All special characters that are not allowed in the cookie-name or cookie-value are encoded with each one's UTF-8 Hex equivalent using percent-encoding *(http://en.wikipedia.org/wiki/Percent-encoding)*.
>
> The only character in cookie-name or cookie-value that is allowed and still encoded is the percent `%` character, it is escaped in order to interpret percent input as literal.
>
> Please note that the default encoding/decoding strategy is meant to be interoperable only between cookies that are read/written by js-cookie *(https://github.com/js-cookie/js-cookie/pull/200#discussion_r63270778)*. To override the default encoding/decoding strategy you need to use a converter.

By using the standard API, you would use the agreed-upon behaviour, whichever it might be. Callers could rely upon that behavior with confidence. I'll admit that I have not tested how exactly the Cookie Store API does value encoding (yet).

Depending on your state management and the scope of such an SDK, you might expose specific hooks to notify your application of cookie changes. Instead, you could rely on the standard `CookieChangeEvent` to notify the application, simplifying the integration with the SDK.

It's good to imagine the future, though I recognise that until adoption is reliably wide, such SDKs will likely paper over the `document.cookie` and Cookie Store APIs internally, and not rely on it externally.

Wrapping up

We looked at some of the current ergonomic and performance issues of the `document.cookie` API, and introduced the Cookie Store API as a contemporary alternative. We looked at some semi-realistic examples of how cookie manipulation fits with state management in applications, and also imagined what a standard Cookie Store API could mean for the future.

I hope this gives you enough to experiment with; please get in touch if you have thoughts! I am especially interested to hear use-cases in your applications, and how adoption might look for what you are building.