SSRG International Journal of Computer Science and Engineering
ISSN: 2348–8387 / https://doi.org/10.14445/23488387/IJCSE-V11I7P102

Volume 11 Issue 7, 8-14, July 2024
© 2024 Seventh Sense Research Group®

Original Article

State Management Techniques and Options for Micro-frontend Web Development

Tanmaya Gaur

Principal Architect, Customer Support, T-Mobile US, Washington, USA.

Corresponding Author : tanmay.gaur@gmail.com

Received: 16 May 2024 Revised: 25 June 2024 Accepted: 12 July 2024 Published: 31 July 2024

Abstract *- The intention behind building applications as Micro Frontend is to develop the experience as a composition of features which are owned and developed completely isolated and by independent teams. These micro-experiences are strung together at run-time or build-time to deliver a single cohesive application experience to the end user. State management refers to the patterns of persisting application data across multiple components and web pages. The state was available to the client application and is used to determine end-user experience. Effective state management keeps the experiences across the entire application session in sync and predictable. Traditional application development has some well-defined options to solve state management. For applications implemented as micro-frontends, there are specific nuances and implications of that development paradigm which need to be accounted for. This paper will attempt to review traditional state management methodologies and provide an overview of considerations when using a micro-frontend style application.*

Keywords - *State Management, CRM, Web Development, Micro-frontend.*

1. Introduction

HTTP follows a classical client-server model, with a client opening a connection to make a request and then waiting until it receives a response. HTTP is traditionally a stateless protocol, meaning that the server on its own does not keep any data (state) between two requests. There are often similar data needs across various parts of an application. While it is always possible to have the UI Application trigger another API Call every time it needs access to that data, performance, load and resource optimization prescribes storing such data in a client-side state OR building an effective caching solution. Caching by itself is a large topic which we will discuss separately. For right now, let us think of State management as a synchronized cache of dynamic server data on the client side. The onus is on the client application to maintain a session state. State management refers to the overall patterns used to synchronize, store, and expose (to application code) access to the state of the application. Modern dynamic web applications often retrieve significant information across the duration of the user session. State management helps keep this state preserved on or near the client and exposes methods to allow different parts of the application to access and utilize this data. This helps applications avoid expensive API calls (refer to Figure 2) to the backend, thereby reducing back-end load as well as avoiding duplicative code. If done right, this would imply fewer lines of code, which in turn means simpler logic, which makes applications easier to understand and maintain. A good state management strategy accounts for ensuring consistency of the state and provides strategies for revocation and override if the state gets stale. Overall, state management can help code quality, operability, and maintenance as it can promote the reuse of application code across multiple components or modules of a web application. While traditional options for state management apply to micro-frontends, there are specific nuances to be aware of. These details are often missed in documentation around micro-frontend, which mostly focuses on the build and deployment strategies. In this paper, we will go through the common state management techniques one by one and discuss the unique specificities of using them in a micro-frontend. This paper will also compare and contrast these techniques from the security, performance, scalability, and usability of a micro-frontend web application. In most cases, it is recommended to place the state management responsibility within a common omnipresent container app or micro-frontend in a way that the state is available to all other micro-frontends seamlessly. This, however, gets tricky at scale, for example, if we implement micro-frontends using web components which are built for encapsulation and modularity. Also, given how we do want micro-frontends to be able to execute stand-alone with minimal inter dependency, we most often end up with having two state management solutions, one at a micro-frontend level and a global shared state. These and other considerations which are crucial in determining a solution specific to a micro-frontend use-case are also discussed in this chapter.

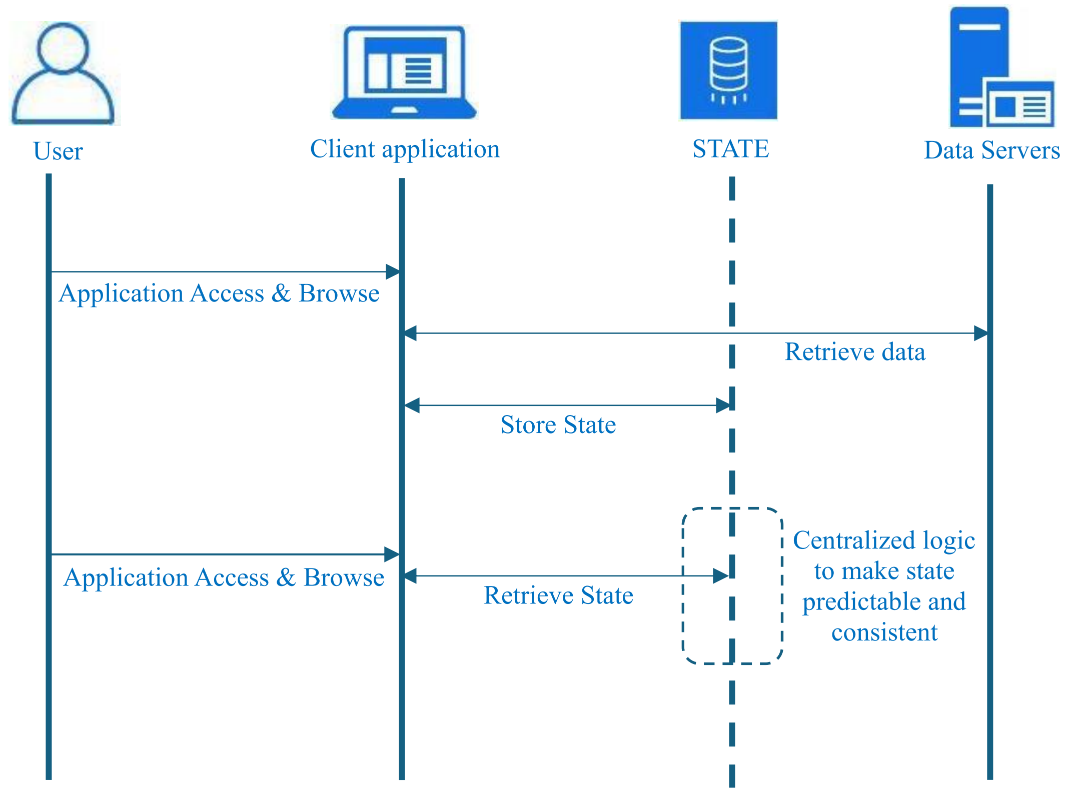
User
Client application
STATE
Data Servers
Application Access & Browse
Retrieve data
Store State
Application Access & Browse
Retrieve State
Centralized logic to make state predictable and consistent

Fig. 1 State across a user Session

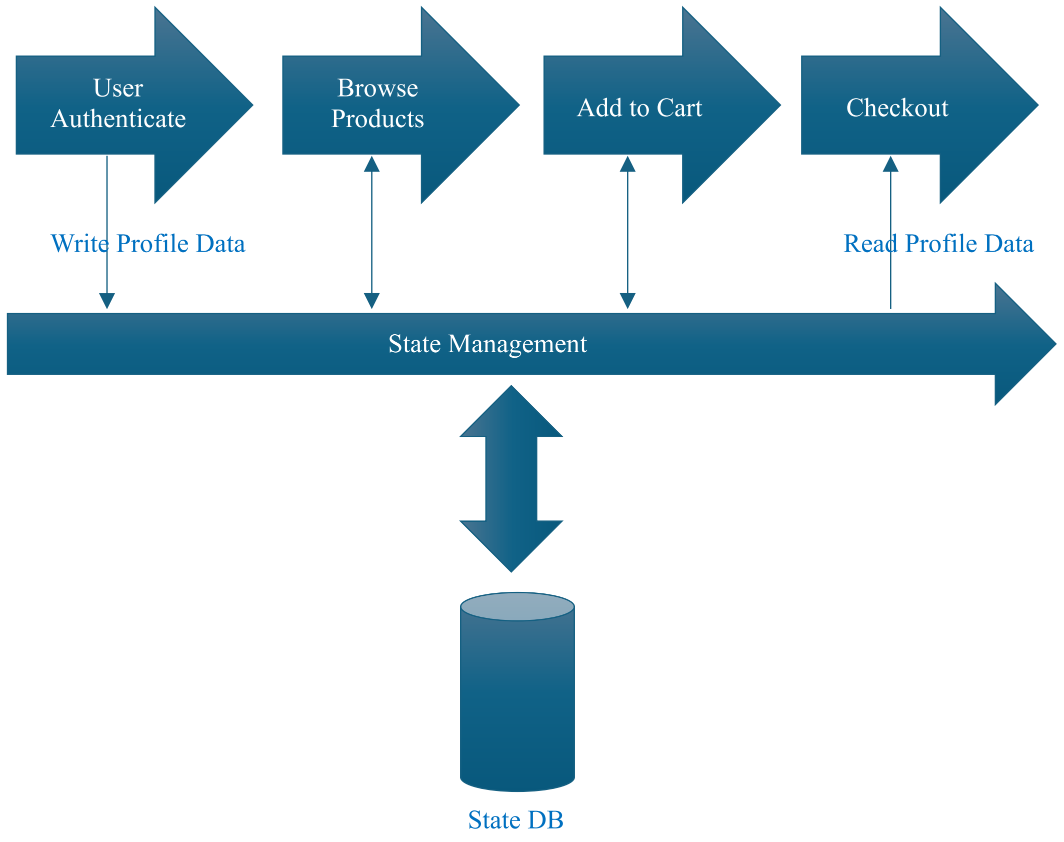

Fig. 2 State sharing across a flow can avoid duplicative API calls

2. Implementation Concerns

There are multiple options available to implement state Management. This section lists the architectural considerations to help decide which ones work best for your specific use case. It is always important to understand your use case and determine the best option for your specific needs. Irrespective of the option you end up with, it makes sense to isolate state management into its own app or within the common container app. We will touch on this in the subsequent section.

2.1. Your Choice of Micro-Frontend

Just like session management, your choice of micro-frontend pattern impacts your implementation of state management. Most monolith apps run in the execution context of the tab and hence share Cookies, HTML5 Storage, Global objects, etc.With micro-frontend, depending on the implementation, this may not always be the case. E.g. a micro-frontend built using Iframe will not have access to the Global objects of the parent container. Irrespective of the underlying storage solution, State management should expose well defined state interfaces available to your micro-frontends. It is ideal to expose methods to query and mutate the state as well as have an observable pattern over important state data as it helps abstract the micro-frontends from the actual state storage. Such abstraction lets developers update the underlying state implementation without impacting every single micro-frontend.

State management is responsible for the consistency of experience across the application as the state of an application evolves. This becomes slightly more taxing as all micro-frontends are coded and designed to be independent. Also, micro-frontends concurrently loaded on a webpage may all be interested in user activity interacting with any one of them.

Instead of all micro-frontends chatting with each other, it is a recommended pattern to have them communicate via methods and events exposed by state management (refer to Figure 3). e.g. in a commerce app, if a user adds a product to the cart, the banner and other apps may need to update upsell and cross-sell offerings appropriately.

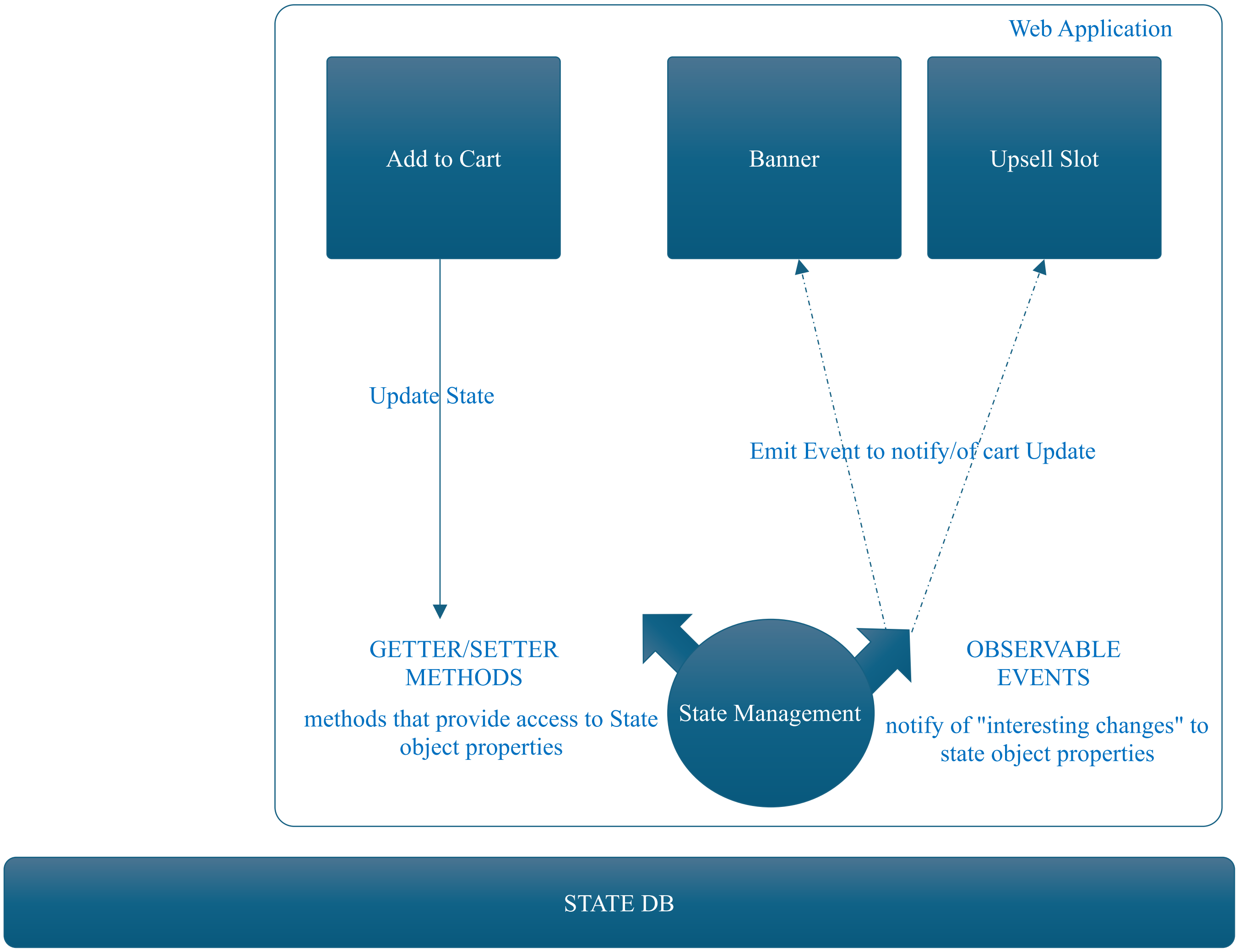
Web Application
Add to Cart
Banner
Upsell Slot
Update State
Emit Event to notify/of cart Update
GETTER/SETTER METHODS
methods that provide access to State object properties
State Management
OBSERVABLE EVENTS
notify of "interesting changes" to state object properties
STATE DB

Fig. 3 State management abstracts micro-frontends concurrently running in a web application

2.2. Session v/s State Management

While sometimes being used in overlapping contexts, there exists a difference between state and session management. The difference often depends on your application and the specific use case.

- Session management refers to the approaches to keep sessions consistent between client and server by persisting a secure token on the client app, which is sent with all HTTP traffic and is well understood by the server.
- In certain scenarios, applications may want to store more data on the client than just the session tokens. This allows the application to reduce network traffic and, at times, simplify the client application. This may avoid code duplication and assist performance at times by avoiding http roundtrips. This is an example of an application state where the data is specific features in the application or stores details of the current user. There could also be data related to the user's interaction with the application, the status of various events and location data like which page the user is currently on and his navigation history.

2.3. Client-Side v/s Server-Side state

Client-side state refers to patterns storing the state on the client device itself. This is the more commonly used pattern for web applications storing state. There are various options like cookies, Web-Storage API, Global Variables and indexedDB, which will be discussed in the next section. Storing state on the server side means it will be stored on a server outside the client device. This makes it easier to share that data across multiple devices and to manipulate it without approval or access to the device itself. There are specific use cases where server-side state storage makes more sense than the client side.

2.4. Security Risks

Almost all browser data storage options are susceptible to XSS. Some options provide an additional level of data isolation, like HttpOnly cookies, private closures, Web Workers, etc. Still, none of these options are foolproof from a security perspective. State data Security is not just dependent on your choice of data storage, and there is also the need to make sure that JavaScript that can execute on the page is secure. As such, its best to consider all browser data unsecure and never trust it where it can have security connotations.

2.5. The Data itself

There are various limitations to the different data storage options. Data stored server side comes with upload/download costs. The client-side options have restrictions with regard to the size and data types supported. As an example, web storage limits data stored to a maximum of 5MB while cookies are limited to 4 KB. These considerations are a topic for the next section.

2.6. User-Agent Compatibility Required

Your use cases and browsers may also influence the state strategy you need to support. E.g. Certain browsers do not support indexed DB in private browsing mode.

2.7. Data Availability

Does the state need to persist across multiple tabs? Does the state need to persist across user sessions? Does the state need to persist across a browser crash? These are important considerations when deciding your state management solution.

2.8. Data Fetching and Caching

There is an overlap between the ideas of "managing state" and "caching fetched data from the server". For example, you can use a state management tool like Mobx or Redux to track the loading state and cache the fetched data, although they are not purpose-built for that use case. There are also tools that are specifically designed to abstract the use case of fetching data, caching it, and managing the loading state and cached data internally without needing to write that code yourself. Depending on your specific needs, these choices at times, may make more sense than writing the solution from scratch.

2.9. State Categories

Another way to think of the state is to think of the different needs at different levels.

- Micro-frontend State: These state variables can be accessed and/or modified within the micro-frontend implementation. The actual scope will depend on the specificity of the implementation.
- Page State: These variables are limited to being accessed and/or modified within a page or route. All micro-frontends loaded on that specific page will have access to this data. The data is cleaned up at specific events, traditionally as you navigate away from the page. An example would be page reload.
- App State: These variables can be accessed and modified across the application. i.e. all micro-frontends should be able to access and update this data.

3. Implementation Options

With the basic understanding in mind, this section details some of the key state management considerations for micro-frontends, working through the different implementation options. The paper will focus on client-side application state solutions only.

3.1. Cookies

Cookies or HTTP Cookies are small files of information sent by web servers to web browsers over the HTTP protocol. Web Browsers store the cookies they receive for a specific period, as specified in the cookie metadata returned by the server. There are other properties associated with cookies that help manage data security. In micro-frontends, cookies can be used to communicate between different micro-frontends, but data can only be shared if the micro-frontends are under the same sub-domain.

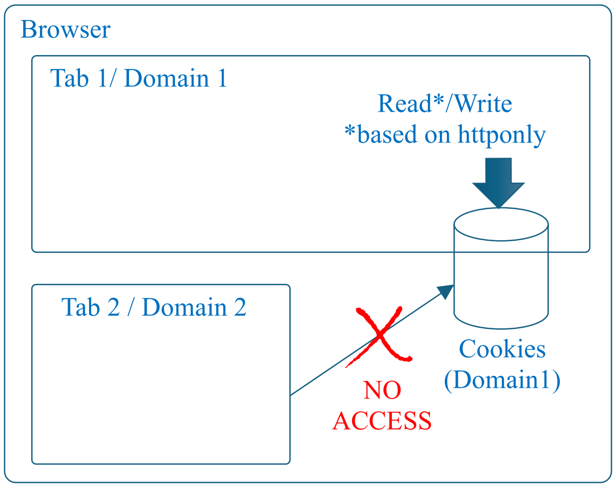

Fig. 5 State data in cookies

Cookies can be used to store state data (Refer to Figure 5) and are automatically transmitted as part of the headers unless the request is to a cross-domain server. They can also be available to both the client and server, depending on how they are configured. They are also widely supported across all browser types. The downside of cookies is that they are limited in the amount and type of data they can store. Cookies support a maximum of 4096 bytes of data, which is not enough for the data needs of modern applications.

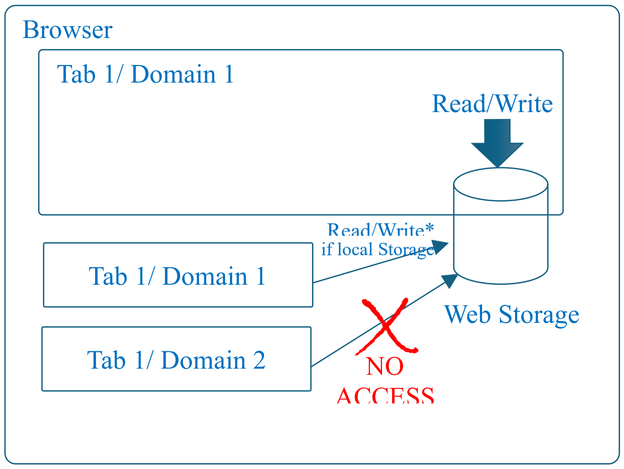

Fig. 6 State data web storage (Session or local storage)

3.2. Session or Local Storage

Session storage and local storage (Refer to Figure 6) provide application developers capability for storing name-value pairs client-side. The key aspects of this storage area:

- Scope: This defines who can access the stored data. Data stored using the localStorage API is accessible across tabs if the webpages which are loaded on these tabs are from the same origin. sessionStorage API stores data within the window context from which it was called, meaning that the tab cannot access data which was stored from Table 2.
- Duration: localStorage persists across browsing sessions, whereas sessionStorage stores data for the duration of the current browsing session only.

So why use localStorage instead of sessionStorage. localStorage is best if data is needed to be accessed across windows or tabs, across multiple sessions. SessionStorage on the other hand, allows you to store data scoped to the tab, allowing you to run multiple simultaneous tabs. Browser storage can support up to 5MB of storage per origin.

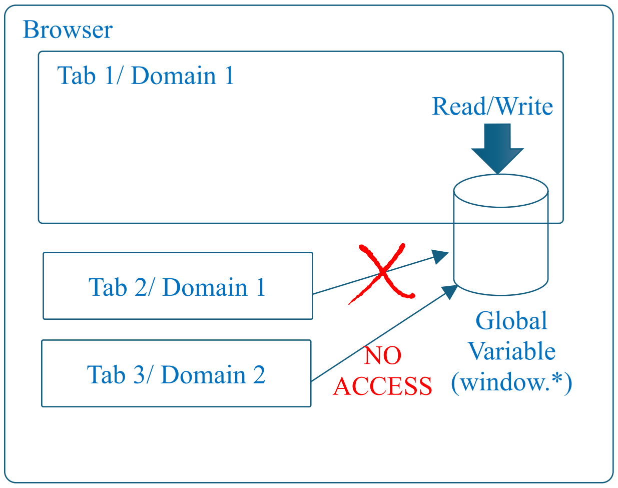

Fig. 7 State tokens in global variables

3.3. Global Variable

A custom variable is created by the application under the window.variableName means that the variable is being declared at the global scope. This means any JS code running in that tab will have access to this variable. How does this compare to sessionStorage (Refer to Figure 7)? Let us find out?

- Scope: Window variables have a global scope, while session storage variables have a session scope. This means that any function or script on the page can access window variables.
- Duration: Window variables do not persist, which means that they are lost when the page is unloaded.

Global variables are useful if you want to store a value that needs to be accessed by multiple functions or scripts on the page. Suppose that aligns with the needs of your application. These work for all micro-frontend styles except Iframe-based approaches.

3.4. Service Worker

While more complex to implement than the other methods discussed so far, service workers bring some cool capabilities to the mix, including their ability to proxy http requests. Service Workers can behave like a browser proxy server (Refer to Figure 9) that executes in an impendent context that persists even if your web app refreshes or reloads. Service workers do have the additional benefits of running in their own context and hence can be more performant than having data in memory in a global variable. Some obvious shortcomings are that you'll have to write a fallback for scenarios where service workers are not supported, as an example, you cannot use it in incognito in Firefox, and it's going to increase the complexity of the app with the message passing / asynchronosity aspect. Also, in theory, there are more points of failure as you are introducing a Service Worker into the mix. If having multiple tabs in sync is a requirement, this is a much better approach than trying to broadcast the data using post messages.

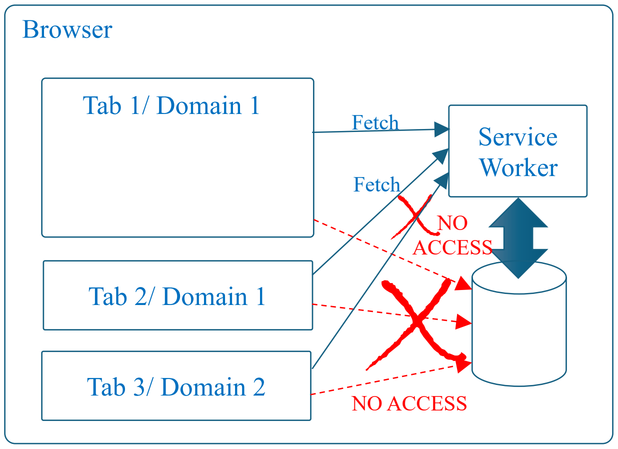

Fig. 9 State management using service worker

3.5. IndexedDB

IndexedDB (Refer to Figure 10) is a powerful client-side storage mechanism that allows web developers to store structured data, including files/Blob. The key difference from web storage API is the ability to store large amounts of data as well as it being asynchronous API and its ability to survive tabs and even a browser crash.

IndexedDB has some clear advantages over other solutions.

- In most user agents, it can store 1GB of data, which makes it suitable for larger storage needs. Also, not having these objects in memory can help with performance.
- IndexedDB can native JavaScript object data and hence avoids the need to serialize data into JSON strings.
- IndexedDB access is asynchronous, so it has minimal impact on the main JavaScript processing thread.

While indexedDB makes a lot of sense for state management and is also accessible from, it does not add significant value as a session token solution for most web applications.

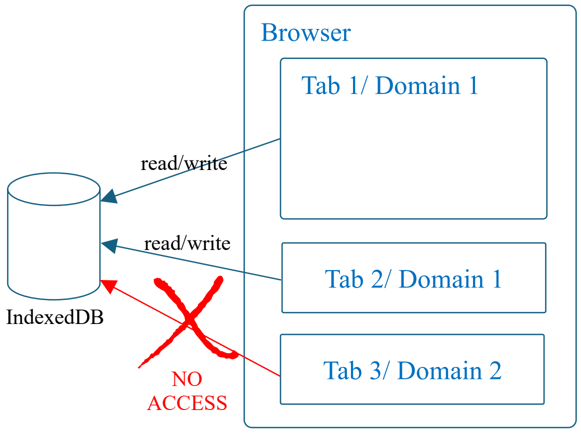

Fig. 10 State in indexedDB

4. Conclusion

Session management remains vital to web development irrespective of the architecture being a monolith or a micro-frontend. While it is entirely possible to follow traditional session management approaches when using micro-frontends, there are key architectural differences to consider, as detailed in this paper. There are some obvious shortcomings of certain approaches, as discussed. Beyond these micro-frontend nuances, the choice of the best technique or option depends on various other factors, such as the security, performance, scalability, and usability requirements of your web application, and the preferences and capabilities of the web developer.

References

[1] Session Management Cheat Sheet, Owasp Cheat Sheet Series, Owasp, 2024. [Online]. Available: https://cheatsheetseries.owasp.org/cheatsheets/Session_Management_Cheat_Sheet.html.

[2] Using the web Storage API, MDN Web Docs, Developer. Mozilla, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API.

[3] IndexedDB API, MDN Web Docs, Developer. Mozilla, 2024. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

[4] How to Store Session Tokens in a Browser (And the Impacts of Each), Ropnop Blog, 2020. [Online]. Available: https://blog.ropnop.com/storing-tokens-in-browser/#global-variable.

[5] Chanchal Aidasani, Browser Storage: A Comparative Analysis of IndexDB, Local Storage, and Session Storage, Browsee, 2023. [Online]. Available: browsee.io/blog/unleashing-the-power-a-comparative-analysis-of-indexdb-local-storage-and-session-storage/.

[6] Craig Buckler, How to Use IndexedDB to Manage State in JavaScript, StackAnatomy, Medium, 2021. [Online]. Available: medium.com/stackanatomy/how-to-use-indexeddb-to-manage-state-in-javascript-50ac358d896c.

[7] James L. Milner, Service Worker State Management, Jameslmilner, 2018. [Online]. Available: https://www.jameslmilner.com/posts/serviceworker-state-management/.

[8] State Management: Overview, React Common Tools and Practices, Netlify.app, 2022. [Online]. Available: react-community-tools-practices-cheatsheet.netlify.app/state-management/overview/.