SSRG International Journal of Computer Science and Engineering
ISSN: 2348–8387 / https://doi.org/10.14445/23488387/IJCSE-V11I8P103

Volume 11 Issue 8, 17-25, August 2024

Original Article

Session 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: 20 June 2024 Revised: 29 July 2024 Accepted: 17 August 2024 Published: 31 August 2024

Abstract *- Micro-frontends extend the concept of micro-services to the world of UI. The idea behind Micro Frontends is to develop applications as a composition of features which are owned and developed completely isolated and by independent teams. These experiences are strung together either at run-time or build-time to deliver a single cohesive application experience to the end user. Session management, which is a crucial aspect of traditional web development, is also a requisite for micro-frontend applications as it enables the application to maintain the state of a user's session. This could mean things like preferences, actions, and authentication status across different types of applications and user types. Without session management, the backends would treat each request as a new one, and the user would have to re-enter their credentials, preferences, and data every time they interact with the web application. Traditional application development has some well-defined options to enable session management. For applications implemented as micro-frontends, there are implications specific to that architecture which need to be accounted for. This paper will attempt to review traditional session management methodologies and provide an overview of considerations when using them in micro-frontend style applications.*

Keywords - *Session 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 does not keep any data (state) between two requests. A web session could be thought of as the sequence of network HTTP request and response transactions associated with a user session (refer to Figure 1). The onus is on the client application to maintain a sessionID (or token), often created after verifying the user in a way that the sessionID can be used to back up that claim. Sessions should be unique per user and difficult for a bad actor to impersonate.

Modern and complex web applications require retaining information or status about the user across the duration of multiple requests. Sessions provide the ability to establish variables – such as access rights and localization settings – which will apply to each interaction a user has with the web application for the duration of the session.

Session management may also apply to anonymous users depending on the application functionality, e.g. scenarios where an anonymous user is following a purchase path transaction and transactional activity needs to be coherent across the commerce to order capture flow (refer to figure 2).

While there are multiple options available to implement micro-frontend style apps, in most cases, it generally makes sense to optimize developing global concerns like session management within a container app so that these are always available to all other micro-frontends. It is recommended that the session management responsibility be placed within the common omnipresent container app in such a way that the sessionID is available to micro-frontends as they get appended and removed from the browser Document object model.

There are multiple techniques for implementing session management in web development, each with its own advantages and disadvantages. While significant documentation and guides on the internet talk about session management for traditional web apps, there is a lack of similar insights that are specific to the micro frontend style of web development. This paper will go through the common techniques one by one and discuss the unique specificities of using them in a micro-frontend. This paper will compare these techniques and discuss the implications for security, performance, scalability, and usability of a micro-frontend web application.

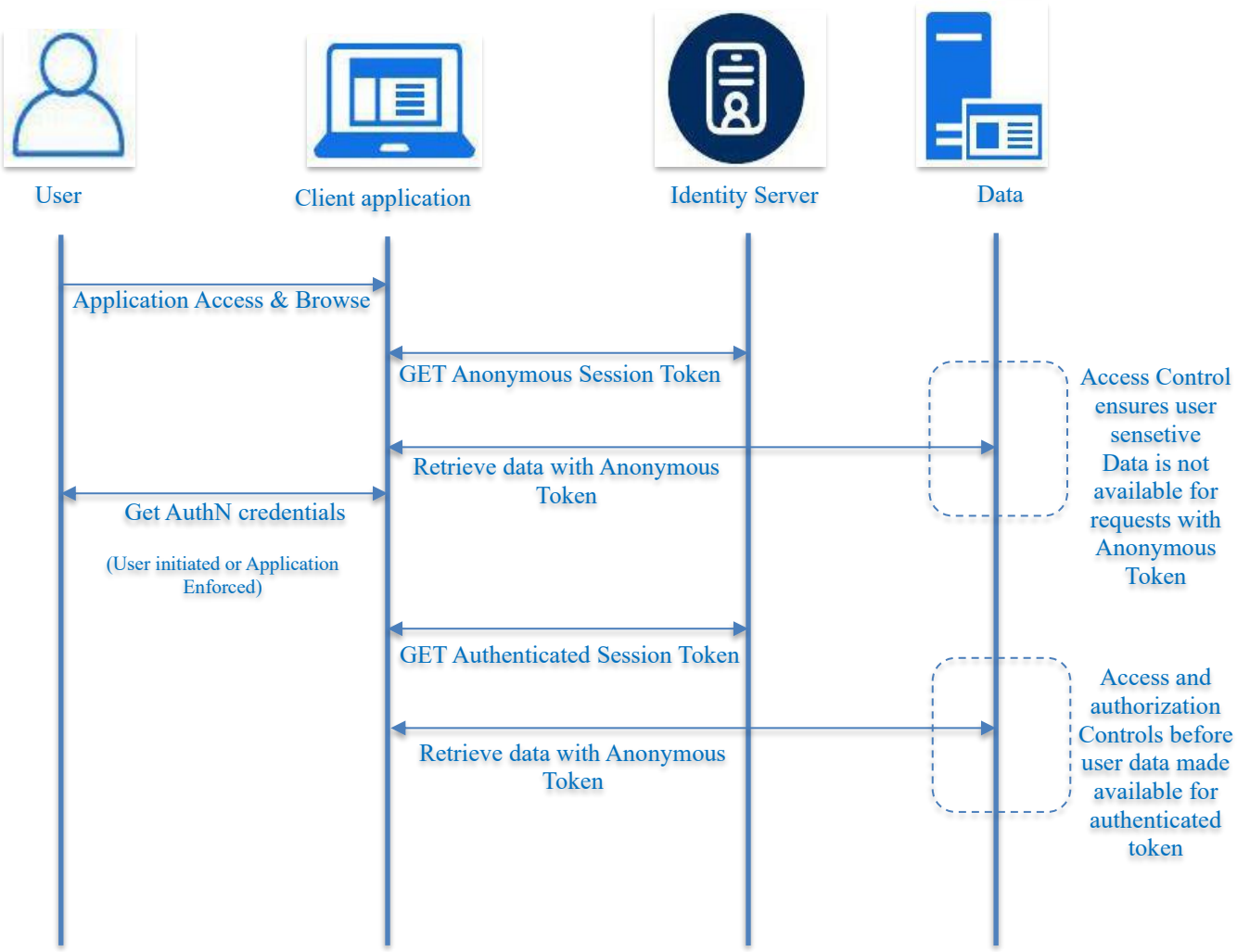

Fig. 1 Session across a sequence of HTTP invocations

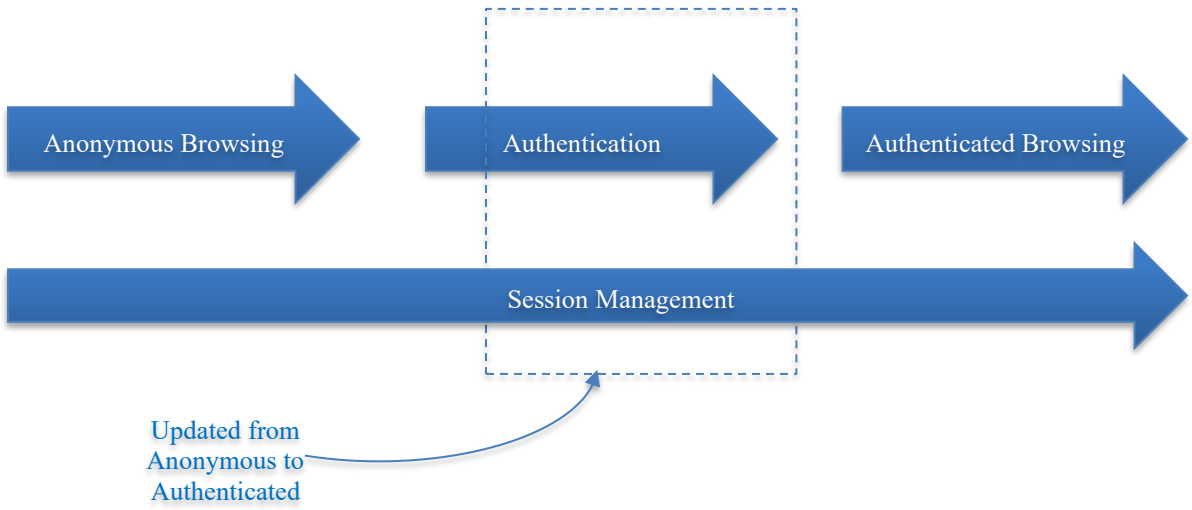

Fig. 2 Session management extends across anonymous and authenticated session

2. Implementation Concerns

There are multiple options available to implement session Management. This section lists some of the architectural considerations for deciding which one works best for you. It is always important to understand the development use-case and determine the best option that meets specific needs. Irrespective of the option you choose, it makes sense to separate the session management concerns of retrieving, persisting, and exposing the session ID into its own app or within the container app. This paper will touch on this in the next section of the post by going over the various options.

2.1. Choice of Micro-Frontend

The choice of micro-frontend technology may limit the options for session management. Most monolith apps run in the same execution context of the browser and hence share Cookies, HTML5 Storage, Windows objects etc. With micro-frontend, depending on the implementation, this may not always be the case. For example, a micro-frontend built using Iframe will not have access to the Windows JS object of the parent container. Even access to cookies is often restricted based on the user agent.

Another impact is the need to build clean interfaces. Since a micro-frontend implementation needs to build stand-alone and independent micro-frontends, developers must create clean API abstractions (refer to Figure 3) in front of methods made available to retrieve the session token and data. Other aspects of session management, like requesting a session refresh, authentication step-up, etc., need to be handled either by a common layer or exposed via clean interfaces.

An aspect specific to micro-frontend is the sharing of tokens across all the micro-frontend apps. Assuming traditional OIDC, the scopes of the session tokens determine what API and backend data these tokens provide access to.

Assuming a micro-frontend may span multiple different functions and needs, it is easy to stuff all required scopes into a single token and run into “super” access tokens. There are ways to avoid this scenario by providing some ways of token binding or exchange mechanism. This is a significant topic on its own.

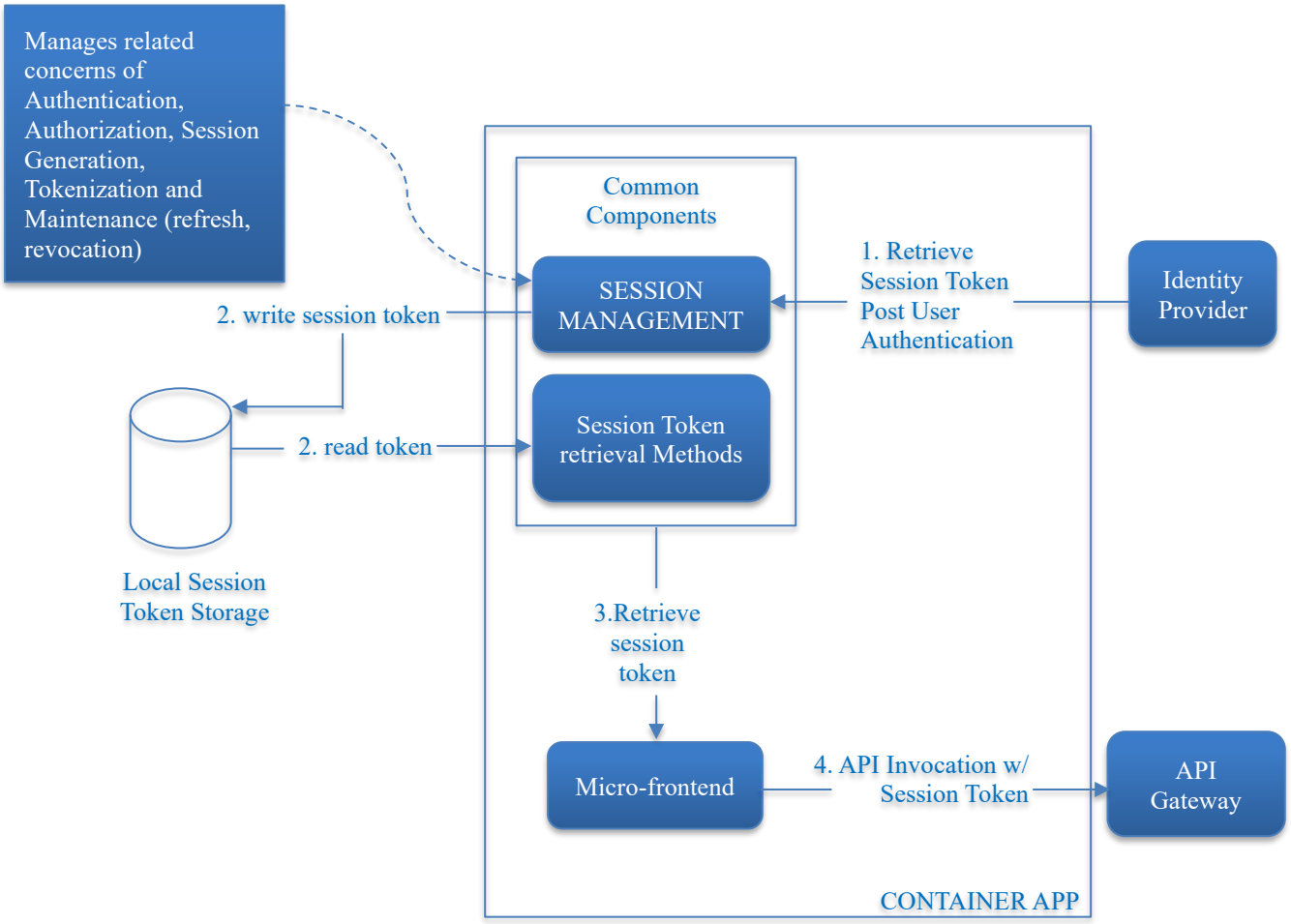

Fig. 3 Session Management within a micro-frontend on the browser

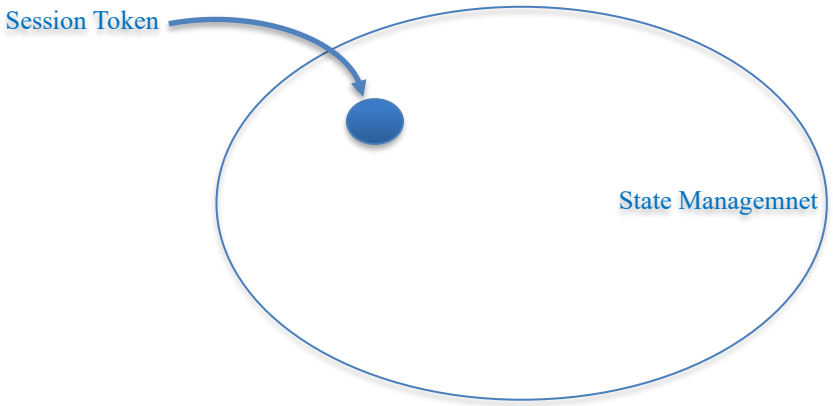

Fig. 4 Session Management a subset of state management

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 the application being developed 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. There are security considerations regarding what data is suitable to be stored on the client side and for how long. This state is confined to the client device.

For example, for an amazon.com commerce flow, the session management token encapsulates the trusted user identity based on authentication credentials. State management, in this case, may include all the items that the user has in their cart, their click and navigation history during the session, etc.

So, what is the overlap? If state management is client-side, then session management may use the state management approach to store and persist the session tokens.

Think of state management as the larger data bucket (refer to figure 4) and session tokens as one of the data elements being stored amongst that data. You do, however, want to consider the other implementation concerns called out in this paper before deciding to reuse the pattern.

2.3. JWT v/s Opaque Tokens

With the understanding that the session tokens are representative of secure session information between the client and server and play a role in the authentication and authorization of the user. Let us dive deeper into the two types of session tokens.

- Opaque tokens have a longer history with web development and are usually an alphanumeric string that identifies some information in the issuer's database.
- JWT Tokens, on the other hand, are JSON strings that contain all the claims and information they represent and are certified by a signature from the issuer. By default, it is unencrypted, but it can be encrypted via the JSON Web Encryption (JWE) standard.

Given the different nature of these tokens, there are nuances which developers must consider. Opaque tokens are essentially unique random strings and, hence, useful for transmitting sensitive information that should not be available to the client. They also have significant advantages when it comes to revoking the tokens and payload size since the data is all stored on the server side.

JWT Tokens, on the other hand, have advantages for highly distributed systems. An application does not have to repeatedly query the authorization server to retrieve token details, as JWT tokens are signed and can be validated locally. This could be crucial when you are building applications for performance and scale. This makes JWT a shiny alternative in case the session data does not contain sensitive information and tokens do not need to be revoked.

2.4. Types of Tokens

Now that the paper has discussed the intent of the session tokens, let us talk about the various types of tokens needed in enterprise applications.

- Access tokens are the most common and are generally issued by an authorization server. These can be opaque of JWT even though there is widespread misconception assuming Access tokens are always Opaque. These are traditionally issued for small durations to prevent

hijacking, a topic that will come up later in this paper. Access tokens are often used as bearer tokens, where the bearer of the token is granted access to specific API(s) and data.

- ID Tokens are part of the OIDC spec and represent the user identity and authentication metadata. These are always JWT and may contain multiple properties and claims standardized for the enterprise, like the issuer (who issued the tokens), actor (who is using the tokens), and subject (identity of the user token is being used for. Often the same but in certain cases, like a CRM, the subject and actor may be different. The actor could be a customer service agent, and the subject is the subscriber account the agent is working on) and the expiry associated with the token.
- Refresh Tokens are tokens that allow a client to invoke a refresh flow for the access tokens. The access token is purposefully given short lifetimes. The refresh tokens allow for ways to obtain a new Access token securely without having the user re-authenticate.
- Access tokens and bearer tokens can be vulnerable to being stolen and are often bound to the client/device/machine to which it was issued. This kind of token binding the Access token or requests to the client/device/machine is often known as the Proof-of-Possession token. MTLS is another strategy used to create sender-constrained tokens.

While the tokens do retain the same functions, how they are shared securely is where micro-frontends may start differing from traditional applications.

2.5. Functionality and User Agent Restrictions

The use cases and browser support may also influence token strategy. For example, suppose an application needs the tokens to be available across multiple windows and domains under the same sub-domain. In that case, cookies may be a better-suited option than using the Windows object and HTML5 session storage methods. In the next section, the paper will list these considerations where applicable to the various methods.

An example of a recent issue was with iframe-style micro-frontends. Chrome 85 came with an update where. Since Chrome 85, a web page inside an iframe on a different domain than the parent cannot read its own cookies unless they have explicitly been set using SameSite=None and Secure. This is another example of the kind of impact choice of session management pattern can have on micro-frontend implementation.

2.6. Security Concerns

Multiple security considerations drive various aspects of session management, including token generation, session ID persistence, transmission to the server, and expiration and refresh.

- Session Hijacking: This can be caused by malicious users capturing and replaying the session ID and can happen with unintentional disclosure, prediction, brute force, or fixation. This allows an attacker to impersonate a victim. There are various mitigation and prevention techniques ranging from simple hardening techniques around naming and storing session variables to complex proof of possession implementations where the session ID is cryptographically bound to the client until it is issued.
- In addition to hardening storage and token binding, it is important to ensure the session ID is unpredictable and large enough to prevent guessing attacks and brute force.
- Security at rest and in transit is crucial to protect against sessionID Hijacking. At rest, this would mean making sure the sessionID is stored in a way that it is not available outside the application's scope. At transit, the mechanism generally involves implementing HTTP Strict Transport Security (HSTS) to enforce HTTPS connections.
- Session Exchange Mechanisms: An application must only trust specific means to transmit and receive sessionID(s). For example, if a session is only available via cookies, the web application should ensure it does not accept it as a URL parameter to stop session fixation-like issues.

Session Exchange Mechanism is a tricky question when micro-frontend applications need to share data between them. Should a token exchange be treated as inter or intra-app? Another complexity is the lack of suitable oAuth methods at times (depending on the identity provider) when trying to exchange tokens inter-app.

2.7. Token Data Requirements

The size of the session-sensitive data may drive the choice of tokens as well as storage. If you want to store large amounts of session data, transmitting it roundtrip as a cookie with every request may have significant performance impacts. Specific to micro-frontends, this becomes a crucial discussion when scaling across multiple different functional concerns. Another aspect of the token data is governance around ensuring the absolute necessary session data is introduced into the token irrespective of whether it being client side or server. There are better and more suitable mechanisms available for broader state and cache management solutions to support performance and load optimization needs.

3. Implementation Options

Now that we understand some of the key session management considerations for web development in general and how they are impacted by micro-frontends, let's work our way through the different session implementation options.

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.

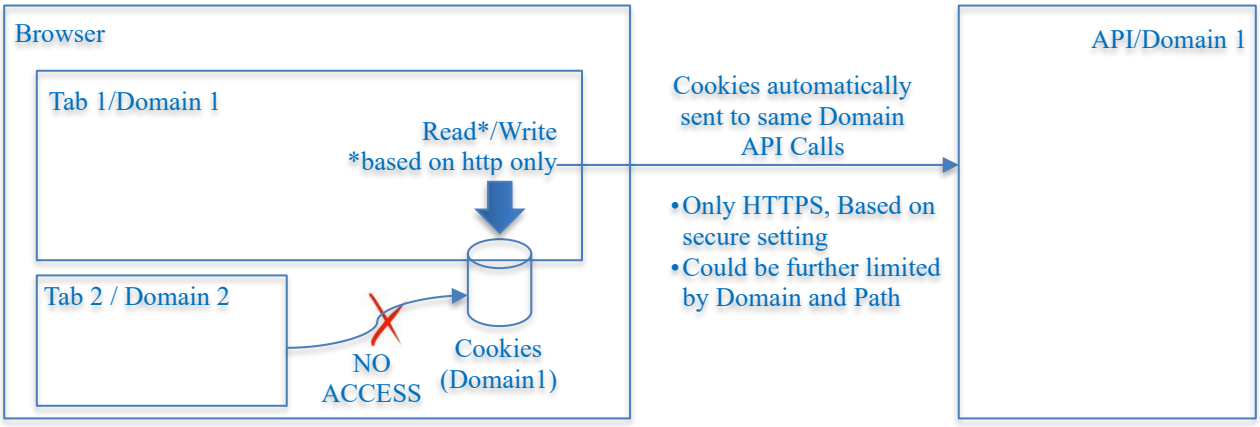

Fig. 5 Session tokens in cookies

Cookies can be used to store session tokens (Refer to Figure 5) and are automatically transmitted as part of the headers unless the request is to a cross-domain server. For sites of the same domain, this makes cookies one of the easier implementation options. They are also widely supported across all browser types.

The downside of cookies is that they are more vulnerable to theft and tampering as well as to cross-site scripting attacks. Another major downside is the impact on performance, especially if large session data is stored in the cookie. One recent issue is cookies often being used to track users and hence are now associated with privacy issues. This, at times, leads to cookies being blocked by some users or user agents, rendering the web application unstable or broken.

Some key flags to be aware of if using cookies.

- Secure: Instructs web browsers to only send the cookie through an encrypted HTTPS (SSL/TLS) connection. This is highly recommended to avoid man in the middle attacks.
- HttpOnly : Instructs web browsers not to allow scripts like JavaScript access to cookies. This is a recommended protection against XSS attacks.
- SameSite : Prevents browsers from sending a SameSite flagged cookie with cross-site requests. The main goal is to mitigate the risk of cross-site request forgery attacks. The paper did already discuss some of the recent implications of this attribute to iFrames.
- Domain : Instructs web browsers to only send the cookie to the specified domain and all subdomains. If the attribute is not set, by default the cookie will only be sent to the origin server.
- Path : Instructs web browsers to only send the cookie to the specified directory (or specific resource) within the web application. Just like Domain attribute, it is best to restrictively scope this attribute as needed.
- Expire & Max-Age : One of the most important aspects for session management, these attributes allow applications to set an expiry of the cookie. The cookie will be persisted till the specified expiry time (Max-age has preference over expires if both are specified)

3.2. Session or Local Storage

Session storage and local storage (Refer Figure 6) provide application developers capability for storing name-value pairs client-side.

Unlike HTTP cookies, the contents of localStorage and sessionStorage are not automatically shared in the request headers. The key aspects of this storage are

- 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 Tab cannot access data which was stored from Tab 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.

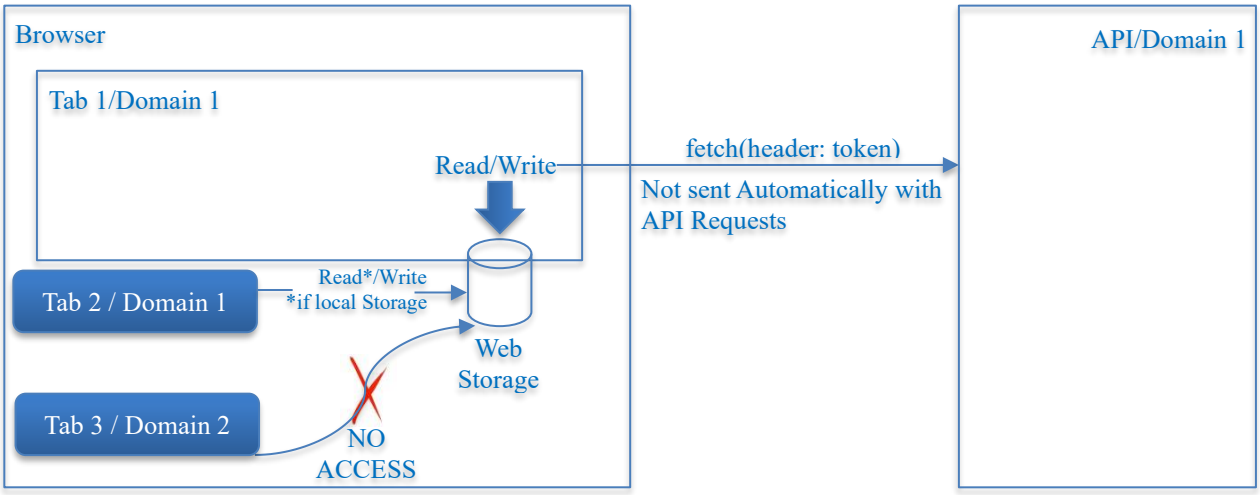

Fig. 6 Session Tokens in Web Storage (Session or Local Storage)

3.3. Global Variable

A custom variable created by the application under 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 Figure 7), let's find out?

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

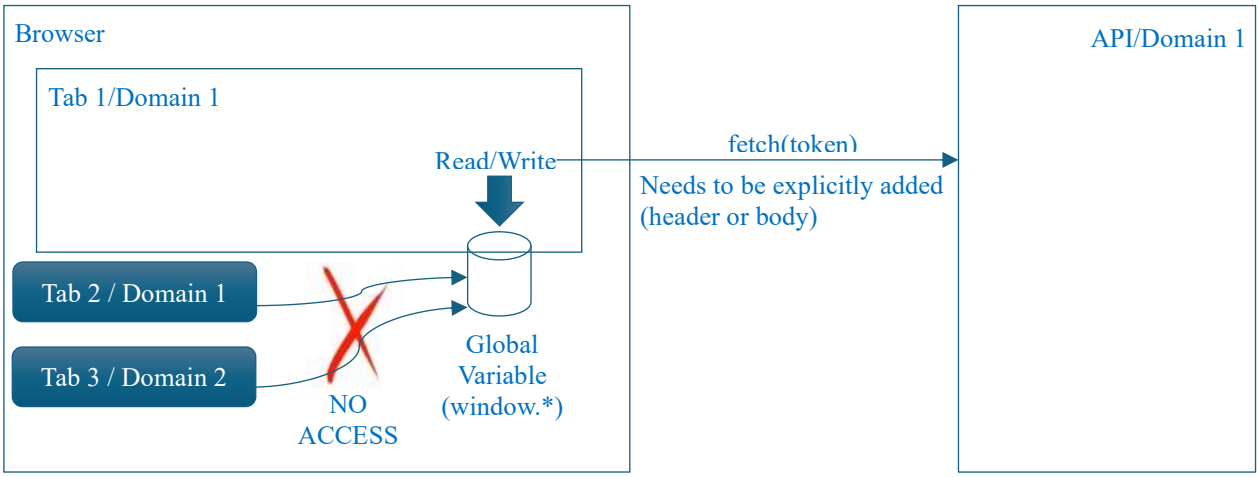

Fig. 7 Session Tokens in Global Variables

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

3.4. Closure Variables

One of the critical security issues with approaches storing the token in local or session storage session hijacking. All Java-script running on the page has access to the token. With micro-frontends especially, this allows even one compromised micro-frontend to be able to hijack the token. Closure provides us add a layer of security in such a scenario. How this works is by developing a closure (Refer Figure 8) which exposes two methods, a set method, and a fetch method. The set method allows the caller to set a session token, often invoked by the applications auth module.

The fetch function appends the token value as authorization headers to outbound API calls, if being made to pre-configured whitelist of domains. The tokens are traditionally only needed when calling API layer which is pre-configured.

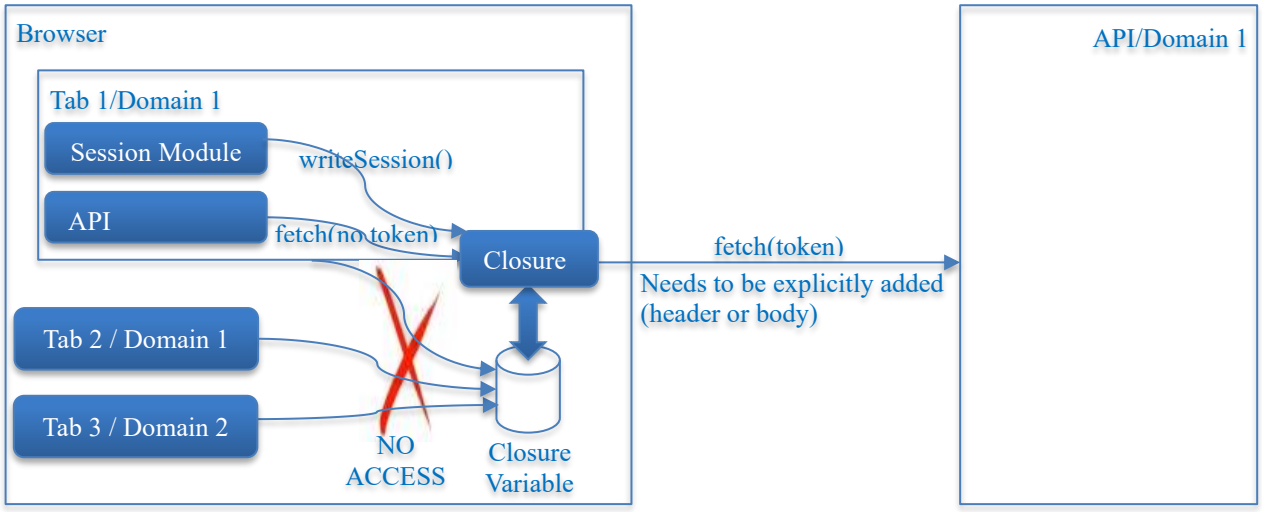

Fig. 8 Session management using Closures

Any time a UI module or a micro-frontend wants to send an API request with the token, it can use the closure's fetch instead of using its own implementation. This approach minimizes XSS impacts but does not completely remediate it.

3.5. 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 Figure 9) that execute in an impendent context that persists even if a web app refreshes or reloads. Developers of web applications can use service worker to store the session token and send the session token for any network resources as required.

The security logic discussed in the earlier section about closure can apply to service worker as well. The service worker can be configured to never return code back to the calling application. There is also the ability to only send out tokens to a pre-configured whitelist of API domains. Service workers do have additional benefits of running in its own context and hence may be more secure than a closure.

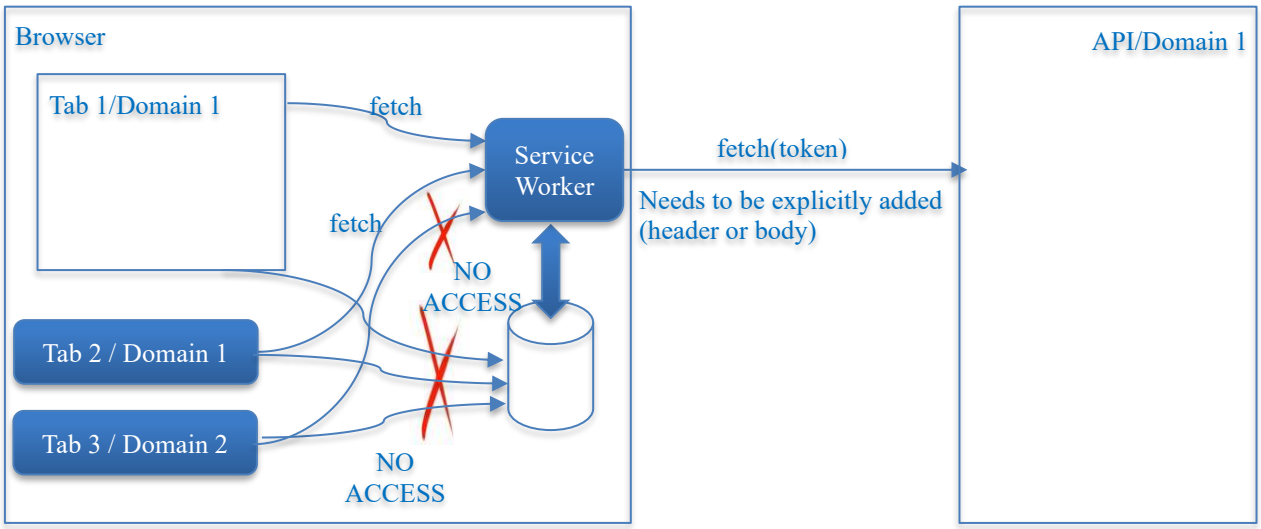

Fig. 9 Session management using service worker

3.6. IndexedDB

IndexedDB (Refer 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 tab and even a browser crash. 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.

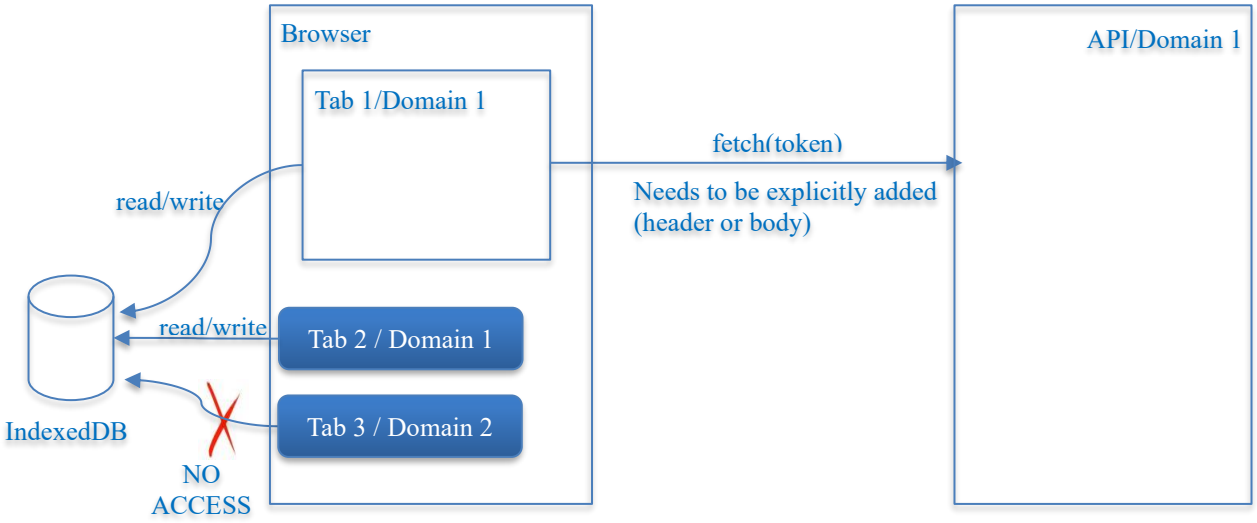

Fig. 10 Session Tokens in IndexedDB

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 that were 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 the web application being developed, and the preferences and capabilities of the web developer.

References

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

[2] The Web Storge API, Mozilla. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/Web_Storage_API/Using_the_Web_Storage_API

[3] The Indexed DB API, Mozilla. [Online]. Available: https://developer.mozilla.org/en-US/docs/Web/API/IndexedDB_API

[4] The Session Token Blog, Ropnop. [Online]. Available: https://blog.ropnop.com/storing-tokens-in-browser/#global-variable

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

[6] State Management in Micro-frontends, Medium. [Online]. Available: https://medium.com/sysco-labs/state-management-in-micro-frontends-ee273830f95f

[7] Navdeep Singh Gill, Micro-frontend Architecture and best Practices, Xenonstack, 2023. [Online]. Available: https://www.xenonstack.com/insights/micro-frontend-architecture

[8] Nathan Sharma, Session Management 101: A Beginner's Guide for Web Developers, MojoAuth, 2024. [Online]. Available: https://mojoauth.com/blog/session-management-a-beginners-guide-for-web-developers/

[9] Ashan Fernando, React MicroFrontend Authentication: Step by Step Guide, Medium, 2024. [Online]. Available: https://blog.bitsrc.io/react-microfrontend-authentication-step-by-step-guide-ca4f3947996f