# Securely Handling Access Tokens and Refresh Tokens in a BFF Pattern with SPA and Blazor (WASM) apps

## Introduction

This document presents a comprehensive overview of the potential security risks associated with storing access and refresh tokens in a browser, and discusses a possible solution using the Backend-for-Frontend (BFF) pattern. We will also touch upon the different types of BFF patterns as per IETF, handling of tokens (on server vs. in cookies), and how to deal with potential race conditions in a distributed environment.

## Risks of Storing Tokens in a Browser

Access and refresh tokens are often stored in a web browser for convenient access during authenticated sessions. However, this practice can expose an application to Cross-Site Scripting (XSS) attacks. XSS exploits occur when an attacker injects malicious scripts into web pages viewed by users. If a token is stored in a JavaScript-readable session or local storage, an XSS attack could allow an attacker to steal these tokens.

Refer the most recent draft of [IETF for browser based apps](https://datatracker.ietf.org/doc/html/draft-ietf-oauth-browser-based-apps) *that explains different patterns of*OAuth 2.0 for Browser-Based Apps*. It explains the security concerns of apps that fetches the tokens directly either from browsing context and service workers along with the security concerns of Implicit flow.* The document is easily consumable and provides in-depth analysis of each pattern.

Service worker solution seem to be safe enough to use tokens from browser but the responsibilities of service worker in that pattern makes it much more difficult to implement. Auth0 JS SDK does this by default. [Auth0 Single Page App SDK](https://auth0.com/docs/libraries/auth0-single-page-app-sdk#change-storage-options)

It also surprised me that the very first draft version of this BCP (Best Current Practice) is not a long ago (Jan 2019) and most recent version draft expires on Sep 2023. (Look on the right-hand section to go through different versions).

## BFF Pattern as a Solution

A potential solution to mitigate these risks involves using the BFF (Backend-for-Frontend) pattern. This architectural pattern involves creating a backend service specifically designed to serve the needs of a particular frontend application. This allows for tailored APIs that can be optimized for each frontend, increasing security and efficiency.

By using the BFF pattern, the frontend (e.g., a JavaScript application running in the browser) never directly handles the access and refresh tokens. Instead, these tokens are stored securely on the server side. The frontend communicates with the BFF, which in turn interacts with downstream services using the stored tokens.

There are libraries in dotnet that offer this BFF features from [Duende BFF](https://docs.duendesoftware.com/identityserver/v5/bff/), but it is not free for everyone [license](https://duendesoftware.com/products/identityserver#pricing)

### Types of BFF Patterns

According to the IETF, there are two primary variants of the BFF pattern: the full BFF and Token Mediation BFF (TMB). Refer the link above.

1. **Full BFF:** The BFF handles both the API calls and token management. It mediates all interactions between the client and the backend services.
2. **Token Mediation BFF (TMB):**The BFF is responsible only for token management, while the client makes API calls directly to backend services. The BFF provides the client with a token it can use for these direct calls, but this token is of a limited scope and duration.

### Storing Tokens: On Server vs. In Cookies

There are two primary methods of storing tokens in a BFF architecture: storing them on the server and storing them in HTTP cookies. [How to Effectively Use the Token Handler Pattern for SPAs | Curity](https://curity.io/resources/learn/the-token-handler-pattern/)

1. **Storing Tokens on the Server (Stateful):** This method is more secure because tokens are never exposed to the client side. The BFF stores and manages the tokens, using them to make requests to the backend services on behalf of the client.

However, this approach introduces challenges for horizontal scaling, as multiple instances of the BFF need to share token information. Implementing a shared, distributed cache (like Redis) can solve this problem but adds complexity to the architecture.

1. **Storing Tokens in HTTP Cookies (Stateless):** Tokens are stored in HTTP-only, Secure, SameSite cookies, which are not accessible by JavaScript and are automatically sent with each request to the BFF.

This approach simplifies token management and scales horizontally, as each request carries its tokens in a stateless manner. But it introduces exposure to Cross-Site Request Forgery (CSRF) attacks, where an attacker tricks a victim into making an unwanted request. Employing CSRF protection mechanisms, such as anti-forgery tokens, is necessary to mitigate this risk.

### Refreshing Tokens with Cookies

In the case of storing tokens in HTTP cookies, the BFF can refresh tokens transparently. When a token is near expiration, the BFF can request a new token from the authentication server, then set the new token in a new HTTP-only cookie. However, managing the tokens for smooth user experience is much more complex than retrieving the cookie and using it.

To ensure smooth user experience, token refresh can be performed using "sliding sessions" - renewing the session (and therefore the token) with each interaction, or after a certain period of inactivity, rather than waiting for token expiration.

## Handling Tokens in a Backend-for-Frontend (BFF) setup

When handling tokens securely in a browser environment, using a Backend-for-Frontend (BFF) pattern can provide a robust solution. Careful consideration of the token storage strategy (server vs. cookie) and proper handling of potential race conditions are key to maintaining security and efficiency in a distributed, scaled environment.

### Handling Race Conditions in Multiple Instances of Full BFF or parallel request upon/nearing token expiration

In a multi-instance BFF setup, a race condition might occur when the client makes multiple simultaneous requests with a token that is about to expire. In this scenario, each BFF instance might attempt to refresh the token simultaneously. To handle this, a few strategies can be applied:

1. **Centralized Token Management:** If possible, having a centralized token management service or utilizing an API Gateway could help. The centralized service or gateway could handle the token refresh operation, making sure that only one refresh operation is conducted at any given time (since we don’t have any single instance of gateway and adding additional layer would complicate things, I’m not considering it).
2. **Sticky Sessions:** Another way to handle this could be using sticky sessions. This is a method used in load balancing where each client is consistently directed to the same instance of the BFF for each request. This way, even if multiple requests are made simultaneously, they will all be handled by the same BFF instance, eliminating the risk of a race condition. However, this approach could lead to an uneven distribution of load and is not recommended for most scenarios. (since it is not a scalable solution, not considering it)
3. **Distributed Locking Mechanism:** . Libraries such as IdentityModel.AspNetCore provide mechanisms for locks, but it assumes a single server instance. A distributed lock can be employed before token refresh operations, ensuring that only one instance can refresh the token at any given time. This mechanism introduces some latency into the system as incoming requests must wait if a token refresh is in progress. However, the impact on performance largely depends on the frequency of token refresh operations and the time taken to refresh a token. To mitigate this, one strategy is to start the refresh operation well before the token expiration, reducing the likelihood that a user request will coincide with a token refresh operation. And when multiple token refresh calls(ex:/bff/refresh) made from UI (when UI is not implemented correctly or UI has multiple tabs open), only the duplicate token refresh operations will wait for token refresh with distributed lock and not the functional API calls (they continue with old access\_token based cookie that hasn’t yet expired) which doesn’t cause slowness to end user experience.

In practice, the process of refreshing the token should be decoupled from functional API calls. The token refresh operation should ideally be initiated only once when the token is nearing its expiration and should not be triggered by individual requests from the UI. While the token refresh operation is in progress, the existing, non-expired tokens can continue to be used to authorize the regular API calls However, appropriate retry logic and error handling should be implemented to ensure graceful recovery in situations where the token refresh operation fails.

**\*\****Even where there is no multi-instance BFF setup, there is still chance of race condition if the UI app makes parallel calls to BFF when nearing token expiration, hence it is important to have locks within in single instance.* IdentityModel.AspNetCore (Duende.AccessTokenManagement now) *provide mechanisms for this using Semaphore. In case of stateful BFF where the tokens are stored on server (ex: Redis) and client simply uses a session cookie, a worker process can be used to update the tokens behind the scenes. However, it is important to have distributed lock for worker process logic to perform duplicate refresh in multiple instances.*

*The cookie generated in asp.net core is encrypted using Data protection keys which are in-memory by default and you have to have a centralized mechanism to persist these keys using* [*Key storage providers in ASP.NET Core*](https://learn.microsoft.com/en-us/aspnet/core/security/data-protection/implementation/key-storage-providers?view=aspnetcore-7.0&tabs=visual-studio) *or use a common cert in all instances.*

### How about using TMB instead of Full BFF

A Token Mediating Backend (TMB) is indeed an alternative to a full Backend-for-Frontend (BFF) solution. A TMB focuses only on managing tokens and does not encapsulate or proxy other application functionality. The client is responsible for deciding when to request a new token from the TMB, and the TMB handles the process of obtaining and issuing the tokens.

TMB consists of endpoints to return tokens, refresh and provide user information. Refer the draft proposed for this pattern here [Token Mediating and session Information Backend For Frontend](https://datatracker.ietf.org/doc/html/draft-bertocci-oauth2-tmi-bff).

This approach does have some potential advantages:

* **Simplicity:** A TMB is generally simpler than a full BFF since it has a more limited responsibility. This can make it easier to develop and maintain.
* **Separation of Concerns:** It separates the concerns of token management from the rest of your application functionality. This can lead to cleaner, more modular code.
* **Scalability:** A TMB can be scaled independently of the rest of the application, allowing you to allocate resources where they are most needed.

However, there are also some potential disadvantages to be aware of:

* **Client Complexity:** The client has to handle more logic, as it needs to know when to request new tokens and handle potential failures of token requests and logic to await all functional calls till the token endpoint returns the token.
* **Security Considerations:** Tokens must still be stored and handled securely on the client side, which can be more challenging than server-side storage. *It is recommended not to store tokens in local/session storage, just keep them in-memory to avoid storage targeted XSS attacks.*
* **Race Conditions:** While having the client decide when to request a new token can help avoid some race condition scenarios, there can still be issues if the client makes multiple simultaneous token refresh requests (from multiple tabs) when the token is near its expiration.

TMB can be a good choice if it better aligns with your application's requirements and constraints. Regardless of whether you choose a BFF, a TMB, or some other solution, the most important factor is to ensure that your approach adequately addresses your application's security, scalability, and maintainability needs.

### Conclusion

While handling tokens securely in a browser environment presents challenges, using a Backend-for-Frontend (BFF) pattern can provide a robust solution. Careful consideration of the token storage strategy (server vs. cookie), proactive management of token lifetimes, and proper handling of potential race conditions are key to maintaining security and efficiency in a distributed, scaled environment.

As with any solution, it is important to continuously monitor for any new potential vulnerabilities and to stay updated with the latest best practices in authentication and authorization methodologies.

It is also important to understand the other factors in deciding the Full BFF vs TMB.

For ex: Full BFF offers full security against XSS but mediating all API calls through BFF can be non-performant if API load balancer is configured to use the nearest node for client for low latency.

In TMB, a successful XSS can also call the token endpoint to get the cached token or start a new OAuth flow. In-Detail explanation is available in IETF link provided at the beginning of this

Document.

### References:

[Securing SPAs using the BFF Pattern (once and for all) | Duende Software Blog](https://blog.duendesoftware.com/posts/20210326_bff)[Architecture ::   
  
Duende IdentityServer Documentation (duendesoftware.com)](https://docs.duendesoftware.com/identityserver/v5/bff/architecture/)

**Examples:**  
***BFF asp.net example (Auth0):*** [Backend For Frontend Authentication Pattern with Auth0 and ASP.NET Core](https://auth0.com/blog/backend-for-frontend-pattern-with-auth0-and-dotnet/) ***Todo app with BFF using*** [***YARP***](https://microsoft.github.io/reverse-proxy/)***, in asp.net core:*** (doesn’t handle the race and refresh mechanisms)  
[davidfowl/TodoApi: Todo application with ASP.NET Core Blazor WASM, Minimal APIs and Authentication (github.com)](https://github.com/davidfowl/TodoApi)  
***if you like to use Duende BFF:*** [No tokens in the browser implementation with Duende BFF - Blog by Kalle Marjokorpi](https://www.kallemarjokorpi.fi/blog/no-tokens-in-browser-with-duende-bff.html)

***Basic example of refreshing cookie with new access\_token in asp.net core*:** (doesn’t handle the use case of paralall request and multi BFF instance):  
[aspnet-core-token-renewal/Startup.cs at master · mderriey/aspnet-core-token-renewal (github.com)](https://github.com/mderriey/aspnet-core-token-renewal/blob/master/src/MvcClient/Startup.cs#enroll-beta)  
And [oauth 2.0 - AddOpenIdConnect and Refresh Tokens in ASP.NET Core - Stack Overflow](https://stackoverflow.com/questions/60858985/addopenidconnect-and-refresh-tokens-in-asp-net-core)

Storing tokens in browser and XSS: - [(27) alert‘OAuth 2 0’; // The impact of XSS on OAuth 2 0 in SPAs - YouTube](https://www.youtube.com/watch?v=lEnbi4KClVw) sliding refresh token explained in this video is now mandatory in OAuth2.1

Curity examples: (their documents and examples also consists of advanced enhanced security patterns for Financial grade apps using MTLS, PAR and JARM): [SPA using the Token Handler Pattern | Curity Identity Server](https://curity.io/resources/learn/token-handler-spa-example/)

Automatic token management in Single instance: [Automatic OAuth 2.0 Token Management in ASP.NET Core | leastprivilege.com](https://leastprivilege.com/2019/01/14/automatic-oauth-2-0-token-management-in-asp-net-core/) and   
[Web Applications — IdentityModel documentation](https://identitymodel.readthedocs.io/en/latest/aspnetcore/web.html)

### Additional topics References:

POP tokens:[OAuth 2.0: The long Road to Proof-of-Possession Access Tokens | leastprivilege.com](https://leastprivilege.com/2020/01/15/oauth-2-0-the-long-road-to-proof-of-possession-access-tokens/)  
And the return of Dpop :  
[OAuth and Proof of Possession Access Tokens | Duende Software Blog](https://blog.duendesoftware.com/posts/20230328_pop/)

If using asp.net and any SPA (react/angular) using new SPAproxy instead of old SpaServices: [ASP.NET Core and SPAs in .NET 6 · Issue #27887 · dotnet/aspnetcore (github.com)](https://github.com/dotnet/aspnetcore/issues/27887)

if you like to add more complexity expiry and sliding expirations:  
[Authentication cookie lifetime and sliding expiration in ASP.NET Core (brokul.dev)](https://brokul.dev/authentication-cookie-lifetime-and-sliding-expiration)  
[Update for .NET 6's OnCheckSlidingExpiration · Issue #18 · DuendeSoftware/BFF (github.com)](https://github.com/DuendeSoftware/BFF/issues/18)  
[Add a new OnCheckSlidingExpiration event to control renewal by Tratcher · Pull Request #33016 · dotnet/aspnetcore (github.com)](https://github.com/dotnet/aspnetcore/pull/33016)

To understand the SameSite, Lax, None Cookies and SPA cookie auth vs CSRF:  
[ASP.NET Core SPA Cookie Authentication vs XSRF (.NET 7 Minimal Apis C#) - YouTube](https://www.youtube.com/watch?v=9OU_SsOb2SE)