**Authentication nâng cao trong SPA (React/Vue) dùng JWT kết hợp Cookie**

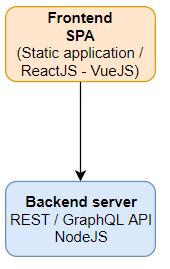
[jwt](https://viblo.asia/tags/jwt)[cookie](https://viblo.asia/tags/cookie)[Single Page Application](https://viblo.asia/tags/single-page-application)[Authentication](https://viblo.asia/tags/authentication)[Session](https://viblo.asia/tags/session)

**Có thể bạn chưa biết:** Trong tháng 5 này 300 thành viên đầu tiên hoàn thành 4 bài viết hợp lệ sẽ nhận được bộ phần quà bao gồm: 1 Áo phông, 1 Túi, Stickers. [**Đăng ký ngay tại đây.**](https://mayfest.viblo.asia/)

Cookie, session, token, JWT, lưu token ở đâu, các mối quan tâm về xác thực trong một hệ thống Single-Page Application... tất cả mọi thứ bạn cần biết đều ở đây.

**TL;DR;**

Có thể triển khai Authentication (xác thực) trong single page application (SPA) với nhiều mô hình có ưu điểm, nhược điểm riêng. Bài này sẽ nói về các concept (khái niệm) quan trọng bạn cần biết khi xử lí với user authentication, đặc biệt là trong kiến trúc xây dựng SPA khá phổ biến hiện nay:



**Điều kiện tiên quyết về bảo mật**

**Mã hoá giao thức (HTTPS)**

* Vì authentication sử dụng HTTP header để truyền các thông tin xác thực (dữ liệu nhạy cảm như: password, access token, ...), các kết nối này cần phải được mã hoá nếu không trong trường hợp các hacker có thể hack vào mạng WiFi của người dùng, những thông tin này có thể bị đánh cắp/

**Không dùng URL query params để truyền dữ liệu nhạy cảm**

* URL và URL params (url params: thí dụ như <http://example.com/?password=123456>) có thể được lưu (dưới dạng log) ở server, trình duyệt (trong history) -> có thể bị đánh cắp và lợi dụng.
* Nếu bạn để authen token ở URL query param, nhiều user ngây thơ có thể copy url trên trình duyệt và send thẳng cho "hacker". 
* Kích thước URL thường bị giới hạn ở browser hoặc server, vì vậy sẽ không thể đảm bảo tính toàn vẹn của dữ liệu được gửi đi.

**Ngăn chặn tấn công "brute-force"**

* "Brute-force" là phương thức tấn công kiểu "thử sai", thí dụ hacker sẽ thử đăng nhập bằng hàng loạt mật khẩu cho tới khi thành công (thường được thực hiện bằng tool).
* Có thể ngăn chặn bằng cách triển khai một middleware "rate limit" ở phía backend, hầu như mọi ngôn ngữ / web framework hiện nay đều có hỗ trợ implement phần này.
* Chặn IP một user nếu user này cố tình tìm kiếm lỗ hổng trên server (user này thường sẽ tạo ra các lỗi HTTP code 3xx, 4xx và 5xx), chặn luôn để tránh hậu hoạ về sau .
* Đừng có để cho người ta biết là bạn dùng code gì ở backend (nó sẽ dễ tìm ra lỗ hổng hơn đó ), thường là xoá đi phần X-Powered-By trong response header (đặc biệt là nếu xài các framework của .NET và Java thường có sẵn phần này).

**Update dependency trong code thường xuyên**

* Nên cập nhật thường xuyên các dependency, thư viện hoặc framework mà bạn xài trong code, thường các bản cập nhật sẽ fix các lỗi về bảo mật được phát hiện.
* Các kiểm tra và update dependency nếu bạn xài NodeJS (cả server-side lẫn client-side) như sau:

# Hiển thị list các lib bị outdated

npm audit

# Update minor và patch version trong package.json

yarn outdated

yarn update

# Update dependency theo minor và patch trong packjage.js

yarn upgrade-interactive

# Update lên bản mới nhất

yarn upgrade-interactive --latest

# Nếu xài NPM thì cũng tương tự

npm outdated

npm update

# Có thể xài tools này để check kĩ hơn: npm-check-updates

npm install -g npm-check-updates

ncu

# Hiển thị list các lib bị outdated

npm audit

# Update minor và patch version trong package.json

yarn outdated

yarn update

# Update dependency theo minor và patch trong packjage.js

yarn upgrade-interactive

# Update lên bản mới nhất

yarn upgrade-interactive --latest

# Nếu xài NPM thì cũng tương tự

npm outdated

npm update

# Có thể xài tools này để check kĩ hơn: npm-check-updates

npm install -g npm-check-updates

ncu

* Ngoài ra, update phiển bản OS ở server thường xuyên (lên bản LTS mới nhất), nếu bạn không xài PaaS (như Google App Engine hoặc Heroku).

**Monitor server thường xuyên**

* Triển khai monitor, logging trên server để biết trước các thay đổi bất thường trước khi xảy ra sự cố.

**Cơ chế authentication**

Có 2 cơ chế authentication chính (chúng ta sẽ đưa ra ưu nhược và so sánh sau) để xác thực user trong một hệ thống REST API.

* Bearer Token
* Authentication cookie

**Bearer Token**

**Bearer Token là gì?**

* Bearer token là một giá trị nằm trong phần Authorization header của mỗi HTTP request. Nó mặc định không tự được lưu ở bất cứ đâu (không như cookie), bạn phảu quyết định nơi lưu nó. Ngoài ra nó không có thời gian hết hạn và không có associated domain (như cookie), nó chỉ là một chuỗi giá trị:

GET https://www.example.com/api/users

Authorization: Bearer my\_bearer\_token\_value

Để xây dựng một ứng dụng stateless, chúng ta có thể dùng [JWT](https://jwt.io/introduction/) để triển khai Bearer Token. Về cơ bản, JWT (JSON Web Token) có 3 phần:

* Header
* Payload (chứa các mô tả về user, thường là chứa user id và quyền của user đó: member hoặc admin + thời gian hết hạn của token)
* Signature (chữ kí)

JWT là một chuẩn mở cryptographically secure định nghĩa cách truyền thông tin xác thực một cách stateless giữa 2 nơi dưới dạng JSON. Stateless nghĩa là ở phía server không cần lưu lại state của token này, phần thông tin của user được đóng thẳng vào token. Chuỗi JWT được encode bằng Base64. Phần signature của JWT là một chuỗi được mã hoá bởi header, payload cùng một secrect key (mã bí mật). Do chính bản thân signature đã bảo gồm cả header và payload nên signature có thể được dùng để kiểm tra tính toàn vẹn của dữ liệu khi truyền tải (giống MD5 checksum).

Về cơ bản thì, client sẽ nhận được JWT token một khi đã được công nhận xác thực (authentication) bằng một user/password (hoặc một số phương pháp khác).

Sau khi đã authentication thành công và client giữ token, mỗi request tiếp theo của client sẽ đính kèm token này vào request header. Server khi nhận được request với token sẽ kiểm tra signature có hợp lệ không, nếu hợp lệ server sẽ dùng phần payload của token để truy xuất expire time và thông tin user (tuỳ nhu cầu).

**Use case cơ bản**

* Gửi và nhận các kết nối cần xác thực giữa trình duyệt (browser) và server backend.
* Gửi và nhận các kết nối cần xác thực giữa ứng dụng di động (mobile app), ứng dụng desktop và server backend.
* Gửi và nhận các kết nối cần xác thực giữa server với server (M2M) của các tổ chức khác nhau (OpenId Connect là một ví dụ).

**Lưu JWT ở đâu?**

Nhắc lại lần nữa, JWT (và các bearer token) không tự động được lưu lại trên client (trình duyệt, app), mà chúng ta phải tự implement việc lưu nó ở đâu (RAM, local/session storage, cookie, etc...).

Việc lưu JWT ở local storage trên browser không được khuyến khích:

* Khi user tắt trình duyệt thì JWT còn đó và có thể được dùng tiếp vào lần tiếp theo cho tới khi hết hạn.
* Mọi đoạn JavaScript trên trang của bạn đều có thể truy cập vào local storage: không có gì bảo vệ cả.
* Nó có thể được dùng bởi [web worker](https://developer.mozilla.org/en-US/docs/Web/API/Web_Workers_API/Using_web_workers).

Lưu JWT token ở session cookie có thể là giải pháp tốt, chúng ta sẽ nói tiếp về vấn đề này sau.

Xem thêm chi tiết về store token trong tài liệu của [auth0.com](http://auth0.com/): <https://auth0.com/docs/security/store-tokens>

**Các kiểu attack cơ bản**

* [Cross-Site Scripting (XSS)](https://www.owasp.org/index.php/Cross-site_Scripting_%28XSS%29) là phương thức tấn công cơ bản nhất mà bạn phải quan tâm khi code JavaScript: Hacker sẽ bằng một cách nào đó (thao túng các JS dependency hoặc dùng user input để add các đoạn malicious javascript code) để trộm JWT của user, sau đó mạo danh họ.
* Thí dụ, ở phần comment của blog, một user có thể thêm một comment với mã JavaScript để làm gì đó trên trang này (các user khác sẽ phải load phần JS của user này):

<img src=x onerror="&#0000106&#0000097&#0000118&#0000097&#0000115&#0000099&#0000114&#0000105&#0000112&#0000116&#0000058&#0000097&#0000108&#0000101&#0000114&#0000116&#0000040&#0000039&#0000088&#0000083&#0000083&#0000039&#0000041">

XSS có thể phòng tránh khi bạn escape kĩ các đoạn input (như user comment, post), không cho chèn scripting code vào. Hoặc tránh dùng các public CDN vì khó kiểm soát, nếu hacker xâm nhậm và thay đổi các đoạn script trên CDN mà bạn sử dụng thì web của bạn có thể bị tấn công dễ dàng.

**Authentication cookie**

Một cookie được cấu tạo theo name-value, và được lưu ở trình duyệt, có ngày hết hạn và associated domain (domain được tích hợp). Có thể tạo cookie bằng JavaScript:

document.cookie = ‘my\_cookie\_name=my\_cookie\_value’ // JavaScript

Hoặc từ phía server bằng cách trả về một HTTP Response Header như sau:

Set-Cookie: my\_cookie\_name=my\_cookie\_value // HTTP Response Header

Web Browser tự động thêm cookie vào mọi request tới cookie domain (associated domain).

GET https://www.example.com/api/users

Cookie: my\_cookie\_name=my\_cookie\_value

Trong hầu hết use case (statful), cookie được dùng để lưu một session ID. Trong đó session ID được quản lí và lưu trên server (tạo ra và xoá đi), trái với JWT là stateless thì cookie session cần phía server phải lưu lại.

Có 2 loại cookie ([nguồn](https://developer.mozilla.org/en-US/docs/Web/HTTP/Cookies)):

* **Session cookie**: nghĩa là cookie chỉ tồn tại trong một phiên (session) của browser, cookie dạng này không có Expires hoặc Max-Age để chỉ thời gian hết hạn. Tuy nhiên, từ phía browser có thể dùng session restoring để khôi phục cookie này kể cả khi đã tắt browser, chính vì thế cần phải xử lí session timeout trên phía server nữa cho chắc .
* **Permanent cookies**: thay vì bị xoá đi khi tắt trình duyệt, permanent cookie hết hạn vào một thời gian được chỉ định (Expires) hoặc sau một khoảng thời gian nhất định (Max-Age).

Ngoài ra, cookie được tạo bởi server (HTTP Response Header) có thể có một số tuỳ chọn:

* **HttpOnly** cookie: Javascript ở browser sẽ không bao giờ đọc được những cookie này.
* **Secure**\* cookie: browser sẽ chỉ đính kèm cookie này vào request khi request đó được thực hiện thông qua giao thức mã hoá (thường là HTTPS).
* **SameSite** cookie: cho phép server yêu cầu một cookie sẽ không được gửi đi với cross-site requests, phần nào đó bảo vệ khỏi các cuộc tấn công cross-site request forgery (CSRF). SameSite chỉ mới là bản thử nghiệm và chưa được hỗ trợ bởi tất cả trình duyệt.

**Use case cơ bản**

* Gửi và nhận các kết nối cần xác thực giữa **trình duyệt** (browser) và server backend.
* Nếu phát triển front-end là mobile app hoặc desktop app thì việc authentication với cookie sẽ khó hơn so với dùng JWT.

**Lưu cookie ở đâu?**

**Cookie được lưu tự động** bởi trình duyệt và có sẵn thời gian hết hạn (tuỳ trường hợp) vả cả associated domain.

**Các kiểu attack cơ bản**

* Cross-Site Scripting (XSS): tương tự như với JWT Bearer Token nếu cookie không được tạo ra với HttpOnly option, hạcker có thể đánh cắp cookie này và giả mạo user để đánh cắp thông tin hoặc thực hiện giao dịch bất hợp pháp.
* Cross-Site Request Forgery (**CSRF**) là một phương thức attack khá phổ biến với những trang authentication bằng cookie. Cấu hình **CORS** (Cross-Origin Resource Sharing) có thể được thực hiện trên server để **giới hạn các hostname** được gửi request tới. Tuy nhiên, CORS được kiểm tra ở phía client bằng trình duyệt. Tệ hơn, CORS chỉ có thể giới hạn request được thực hiện bằng các ngôn ngữ phía browser (JavaScript hoặc WSM), có nghĩa là nếu bạn gửi request qua form (HTML Form), CORS sẽ không thể kiểm tra, kiểu như thế này:

<form action="http://someotherserver.com">

Bởi vì không có đoạn JavaScript nào liên quan tới request được tạo ra bởi form này, CORS bị vô hiệu hoá và cookie sẽ được gửi qua request theo form này .

Một ví dụ khác về attack bằng CRSRF: giả sử user đang đăng nhập ở facebook, truy cập một trang tên bad.com. Trang [bad.com](http://bad.com/) này đã bị kiểm soát bởi hackers và có một đoạn code như sau trong trang:

<img src="https://facebook.com/postComment?userId=dupont\_123&comment=I\_VE\_BEEN\_HACKED>

Để ngăn chặn CSRF, tuỳ chọn SameSite phải được thiết lập ở cookie. Tuy nhiên tuỳ chọn SameSite chưa được hỗ trợ ở mọi trình duyệt, nên nó không thể chặn hết mọi cuộc tấn công CSRF. Vì vậy cần phải sử dụng một số chiến lược khác (có thể sử dụng kết hợp với nhau):

* **Thời gian session timeout ngắn** trên server: ở các trang liên quan tới giao dịch ngân hàng, thời gian hết hạn session thường vào khoảng 10 phút hoặc ít hơn.
* **Luôn đòi user đăng nhập khi thực hiện các hành đọng quan trọng**: ví dụ khi user muốn đổi email thì bắt user đăng nhập lại bằng password.

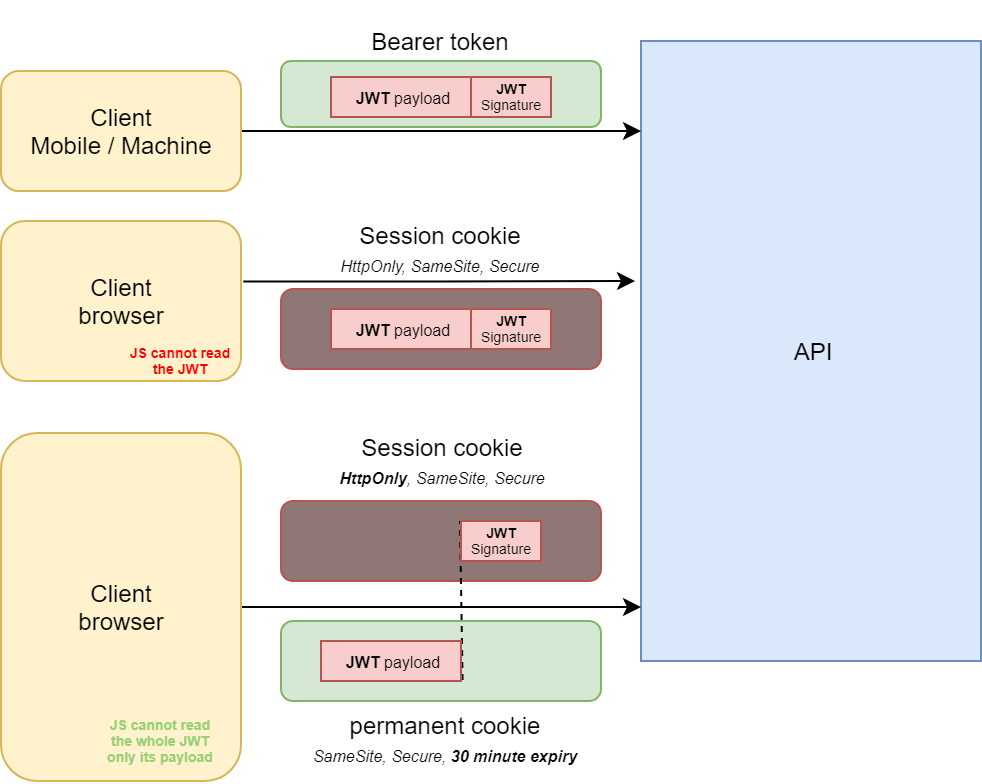
**Có thể kết hợp cả 2 cơ chế không?**

Hãy tổng kết những thứ mà chúng ta mong muốn khi triển khai authentication với server API:

* Hỗ trợ xác thực client-server và server-server (M2M)
* Ngăn chặn tối đa các cuộc tấn công bằng XSS và CSRF
* Stateless nếu có thể (server lưu ít thôi hoặc không cần lưu state luôn )

**Nếu đặt JWT trong cookie value thì sẽ kết hợp được ưu điểm của 2 thằng nhỉ? **

Server API nên hỗ trợ đọc JWT bearer token từ request header cũng như đọc JWT token được lưu bên trong một session cookie. Nếu chúng ta muốn cho phép JavaScript đọc JWT payload thì có thể tiếp cận phương pháp two cookie authentication bằng cách kết hợp 2 loại cookie, nếu vậy sẽ hạn chế được XSS attack khá tốt.

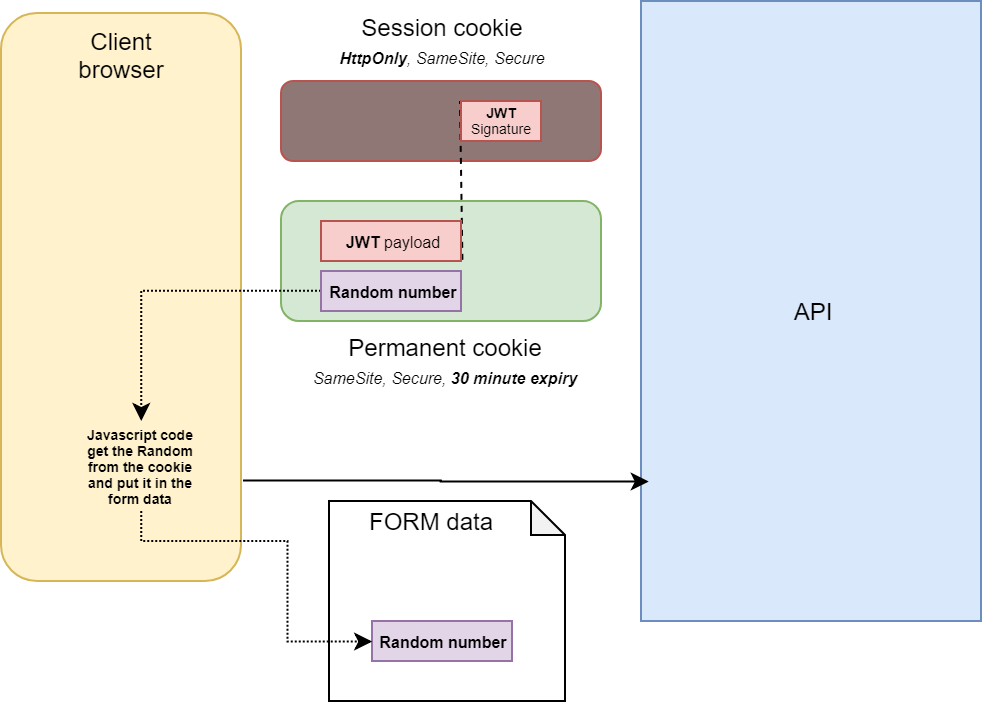


Bạn có thể tìm hiểu về cách tiếp cận two cookie authentication qua bài viết này của tác giả [Peter Locke](https://medium.com/@pdlocke) tại <https://medium.com/lightrail/getting-token-authentication-right-in-a-stateless-single-page-application-57d0c6474e3>.

kết hợp 2 cơ chế, JWT token có thể được cập nhật ở mỗi request liền mạch bởi server, token mới sẽ được trả về thông qua cookie resonse (server set cookie qua HTTP response), và JWT sẽ tự động được lưu bởi browser. Bằng cách này, thời gian hết hạn của JWT có thể được đặt lại ở mỗi request, kiểm soát tốt hơn, nhưng cũng một phần nào đó phức tạp logic hơn .

Để hạn chế CSRF attack, những hành động thay đổi (viết comment, đổi email, password, tên), không nên được thực hiện bằng HTTP GET query, nên dùng PUT hoặc POST. Những sự thay đổi quan trọng (đổi email, địa chỉ) nên bắt user đăng nhập lại lần nữa cho chắc.

Ngoài ra có thể tạo thêm temporary cookie bằng cách get ngẫu nhiên từ cookie và đặt vào form data và submit cùng với form đó dưới dạng hidden form field. Server sẽ phải check nếu random number trong cookie trùng khớp với value được gửi theo form data thì mới hợp lệ.



**Tổng kết**

Quá trình authentication trên Single Page Application của chúng ta bây giờ như sau:

* **Bước 1**: SPA sẽ check trong cookie nếu có JWT payload thì nhảy vào trang member nếu không thì văng ra ngoài trang đăng nhập (/login/). Nếu bạn dùng httpOnly cookie thì không check trực tiếp bằng JavaScript được, phải gửi request tới server để check, ví dụ gửi request tới /backend/api/me để server trả về thông tin của user hoặc lỗi 401 unauthorized error nếu cookie (chứa JWT) không hợp lệ.
* **Bước 2 - Trường hợp 1**: ở trang /login, khi user hoàn tất nhập username và password vào form, bạn có thể gửi tới server để check bằng AJAX request (XHR). Response của AJAX request này sẽ set authentication cookie kèm mã JWT bên trong.
* **Bước 2 - Trường hợp 2**: nếu trang /login dùng chuẩn xác thực bằng OpenID thông qua một cơ chế OAuth. Theo authorization code grant flow, trang /login sẽ redirect browser về /backend/auth/<provider>. Sau đó nếu flow OAuth xong và hợp lệ (user grant đăng nhập với Facebook), server response sẽ set authentication cookie với JWT bên trong. Sau đó browser sẽ redirect về trang của SPA. SPA sẽ quay lại check như bước 1.