VIETNAM AUSTRALIA Vocational College

React Advance
Bài 8: Xác thực trong ứng dụng React

daynghevietuc.com

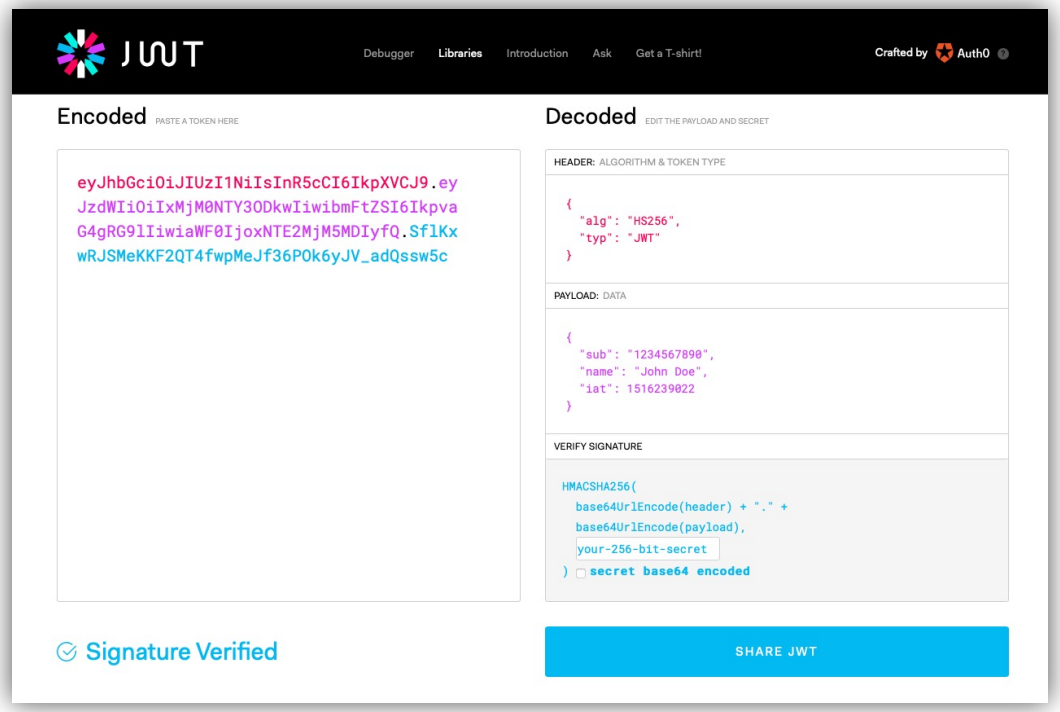
Nội dung

1. Access Token và Refresh Token
2. Local Storage
3. Cookies
4. Xác thực React với JWT và Cookies

1. Access Token và Refresh Token

Access Token và Refresh Token

- JSON Web Token là một cách lưu trữ thông tin xác thực hiệu quả
- Có 2 cách thường dùng để lưu trữ JWT là
  - LocalStorage
  - Cookies



Access Token và Refresh Token

- **Access Token** thường là các JWT Tokens tồn tại trong một khoảng thời gian ngắn, được tạo ra từ phía server, và được server yêu cầu đính kèm trong mỗi Http Request để phía server xác thực người dùng.
- **Refresh Token** thường là các chuỗi string đặc thù nào đó được lưu trong database của server, được sử dụng để tạo ra các Access Token mới mỗi khi chúng hết hạn.

2. Local Storage

Local Storage

- Ưu điểm
  - Đây là một tính năng hầu hết các trình duyệt đều hỗ trợ mà không phải cài gì thêm. Nếu bạn không có phần backend và phải dựa vào API của bên thứ 3 thì bạn không thể yêu cầu người ta đặt Cookies cụ thể cho website của mình.
  - Nó làm việc với các API bằng cách đặt Access Token vào Request Header như ví dụ sau đây

```
Authorization: `Bearer ${accessToken}`
```

- Nhược điểm
  - Dễ bị tấn công XSS

Local Storage

- **Cross-Site Scripting (XSS)** là một trong những kĩ thuật tấn công phổ biến nhất hiện nay.
- Kỹ thuật XSS được thực hiện dựa trên việc chèn các đoạn script nguy hiểm vào trong source code ứng dụng web, nhằm thực thi các đoạn mã độc Javascript để chiếm phiên đăng nhập của người dùng.
- Một cuộc tấn công XSS có thể xảy ra từ mã JavaScript của bên thứ ba có trong trang web của bạn, như React, Vue, jQuery, Google Analytics…
- Ở thời điểm người người xài js, nhà nhà xài js như hiện tại, hầu như không thể không đưa bất kỳ thư viện bên thứ ba nào vào trang web của bạn.
- Hacker có thể chạy 1 đoạn js kiểu: *localStorage.get('token')* trên site của mình để lấy đc token, sau đó gửi ajax tới server của hacker.

Local Storage và tấn công XSS

- *localStorage* dễ bị tấn công vì nó có thể dễ dàng truy cập bằng JavaScript và hacker có thể lấy Access Token của bạn và sử dụng nó sau này.
- Tuy nhiên, mặc dù không thể truy cập cookie gắn cờ *httpOnly* bằng JavaScript, điều này không có nghĩa là bằng cách sử dụng cookie, bạn sẽ an toàn trước các cuộc tấn công XSS liên quan đến Access Token của mình.
- Nếu kẻ tấn công có thể chạy JavaScript trong ứng dụng của bạn, thì chúng chỉ có thể gửi một HTTP Request đến server của bạn và điều đó sẽ tự động gửi kèm cookie của bạn. Nó chỉ kém thuận tiện hơn cho kẻ tấn công vì họ không thể đọc nội dung Access Token mặc dù họ hiếm khi phải làm vậy.
- Việc hacker phải tấn công bằng cách sử dụng trình duyệt của nạn nhân (bằng cách gửi HTTP Request) đương nhiên sẽ tốt hơn là hacker có thể sử dụng máy của chính hacker để tấn công bạn bằng các phương thức khác.

3. Cookies

Cookies

- Ưu điểm
  - Không thể truy cập *httpOnly* và *secure cookie* qua JavaScript, do đó nó không dễ bị tấn công XSS như localStorage.
  - Nếu bạn đang sử dụng *httpOnly* và *secure cookie*, hacker không thể truy cập cookie của bạn bằng JavaScript.
  - Điều này có nghĩa là, ngay cả khi hacker có thể chạy javascript trên trang web của bạn, chúng cũng không thể đọc các Access Token của bạn từ cookie.
  - Cookies tự động được gửi trong mọi yêu cầu HTTP đến máy chủ của bạn.

Cookies

- Nhược điểm
  - Tùy thuộc vào các use case cụ thể mà bạn có thể không lưu trữ được Access Token của mình vào trong cookie.
  - Cookie có giới hạn kích thước là 4KB. Do đó, nếu sử dụng một JWT Access Token lớn, thì việc lưu trữ trong cookie không phải là một lựa chọn đúng đắn.
  - Có những trường hợp bạn không thể chia sẻ cookie với máy chủ API của mình hoặc API yêu cầu bạn đặt Access Token vào trong Authorization Header. Trong trường hợp này, bạn sẽ không thể sử dụng cookie để lưu trữ Access Token của mình.

Cookies và tấn công CSRF

- **CSRF (Cross-site Request Forgery)** là một kiểu tấn công lợi dụng sự tin tưởng của người dùng. Chúng lừa nạn nhân gửi đi các HTTP Request không mong muốn đến website nào đó.
- Khi gửi một HTTP Request (hợp pháp hoặc không hợp pháp), trình duyệt của nạn nhân sẽ gửi kèm theo các Cookies Header.
- Các cookies thường được dùng để lưu trữ một **session** nhằm định danh người dùng giúp họ không phải xác thực lại mỗi khi thực hiện Request.
- Nếu phiên làm việc đã xác thực của nạn nhân được lưu trữ trong một Cookie vẫn còn hiệu lực, và website nạn nhân bảo mật kém và dễ bị tấn công CSRF, kẻ tấn công có thể sử dụng CSRF để chạy bất cứ Request nào với website nạn nhân mà website nạn nhân không biết được các Request này có hợp pháp hay không.
- Một lỗ hổng CSRF sẽ ảnh hưởng đến các quyền của người dùng ví dụ như quản trị viên, kết quả là chúng truy cập được đầy đủ quyền hạn trên website nạn nhân.

4. Xác thực React với JWT và Cookies

Lưu trữ JWT

- Khi di chuyển JWT của bạn ra khỏi bộ nhớ cục bộ, chúng ta nên có hai tùy chọn:
  - Bộ nhớ trình duyệt (trạng thái React)
  - HttpOnly cookie
- Tuy nhiên, lựa chọn đầu tiên không phải lúc nào cũng thực tế. Đó là bởi vì việc lưu trữ JWT ở trạng thái React của bạn sẽ khiến nó bị mất bất cứ khi nào trang được làm mới hoặc đóng rồi mở lại.
- Điều này dẫn đến trải nghiệm người dùng kém – bạn không muốn người dùng của mình cần đăng nhập lại mỗi khi họ làm mới trang.

Lưu trữ JWT

- Nếu sử dụng dịch vụ xác thực của bên thứ ba như Auth0 hoặc Okta , đây không phải là vấn đề lớn vì bạn chỉ có thể gọi một mã thông báo khác ở phía sau (sử dụng lời nhắc = không có cuộc gọi) để nhận mã thông báo mới khi làm mới.
- Tuy nhiên, điều này phụ thuộc vào máy chủ xác thực trung tâm đang lưu trữ phiên cho người dùng của bạn.

Lưu trữ JWT

- Nếu bạn không thể giữ JWT của mình ở trạng thái ứng dụng, thì tùy chọn thứ hai vẫn mang lại một số lợi ích.
- Đáng chú ý nhất, nếu ứng dụng của bạn có bất kỳ lỗ hổng XSS nào, những kẻ tấn công sẽ không thể đánh cắp mã thông báo của người dùng của bạn một cách dễ dàng.
- Mặc dù vậy, việc đưa token của bạn vào cookie HttpOnly không phải là một viên đạn bạc. Giống như bất kỳ ứng dụng an toàn nào, bạn cần bảo vệ hiệu quả chống lại cả lỗ hổng XSS và CSRF.

Ứng dụng sử dụng bộ nhớ cục bộ

- Hãy bắt đầu bằng cách xây dựng một API nút nhỏ với **Express** và một ứng dụng React nhỏ. Chúng tôi sẽ bắt đầu bằng cách có mã thông báo cửa hàng ứng dụng trong bộ nhớ cục bộ và sau đó chúng tôi sẽ chuyển chúng sang cookie HttpOnly.
- Trong khi chúng tôi đang sử dụng **Express** cho phần phụ trợ trong hướng dẫn này, các khái niệm tương tự ánh xạ đến khá nhiều phần mềm phụ trợ mà bạn có thể đang sử dụng.

Tạo API

```
❯ npm i express express-jwt jsonwebtoken cors
```

Server.js > ...

```
1  const express = require('express');
2  const jwt = require('express-jwt');
3  const jsonwebtoken = require('jsonwebtoken');
4  const cors = require('cors');
5  const app = express();
6
7  app.use(cors());
8  const jwtSecret = 'secret123';
9
10 app.get('/jwt', (req, res) => {
11   res.json({
12     token: jsonwebtoken.sign({ user: 'johndoe' }, jwtSecret)
13   });
14 });
15
16 app.use(jwt({ secret: jwtSecret, algorithms: ['HS256'] }));
17
18 const foods = [
19   { id: 1, description: 'burritos' },
20   { id: 2, description: 'quesadillas' },
21   { id: 3, description: 'churos' }
22 ];
23
24 app.get('/foods', (req, res) => {
25   res.json(foods);
26 });
27
28 app.listen(3001, () => {
29   console.log('App running on http://localhost:3001');
30 });
```

Tạo API

- Trong jwtSecret ví dụ này là siêu yếu.
- Đừng sử dụng loại khóa bí mật này trong project thực tế.
- Nên sử dụng khóa bí mật dài, phức tạp, không thể truy cập.
- Lộ trình lấy JWT không kiểm tra bất kỳ thông tin đăng nhập nào, nó chỉ phục vụ JWT khi được yêu cầu.

Ứng dụng React

- Tạo ứng dụng React – food-app

```
❯ create-react-app food-app
```

- Cài đặt package axios

```
❯ npm install --save axios
```

src > App.js > App > getJwt

You, seconds ago | 1 author (You)

```
import React, { useState } from "react";
import axios from "axios";
import "./App.css";

const apiUrl = "http://localhost:3001";

axios.interceptors.request.use(
  (config) => {
    const { origin } = new URL(config.url);
    const allowedOrigins = [apiUrl];
    const token = localStorage.getItem("token");
    if (allowedOrigins.includes(origin)) {
      config.headers.authorization = `Bearer ${token}`;
    }
    return config;
  },
  (error) => {
    return Promise.reject(error);
  }
);

```

Ứng dụng React

src > App.js > ...

```
22  function App() {
23    const storedJwt = localStorage.getItem("token");
24    const [jwt, setJwt] = useState(storedJwt || null);
25    const [foods, setFoods] = useState([]);
26    const [fetchError, setFetchError] = useState(null);
27    const getJwt = async () => {
28      const { data } = await axios.get(`${apiUrl}/jwt`);
29      localStorage.setItem("token", data.token);
30      setJwt(data.token);
31    };
32    const getFoods = async () => {
33      try {
34        const { data } = await axios.get(`${apiUrl}/foods`);
35        setFoods(data);
36        setFetchError(null);
37      } catch (err) {
38        setFetchError(err.message);
39      }
40    };
41    return (
42      <>
43        <section style={{ marginBottom: "10px" }}>
44          <button onClick={() => getJwt()}>Get JWT</button>
45          {jwt && (
46            <pre>
47              <code>{jwt}</code>
48            </pre>
49          )}
50        </section>
51        <section>
52          <button onClick={() => getFoods()}>Get Foods</button>
53          <ul>
54            {foods.map((food, i) => (
55              <li>{food.description}</li>
56            ))}
57          </ul>
58          {fetchError && <p style={{ color: "red" }}>{fetchError}</p>}
59        </section>
60      </>
61    );
62  }
63  export default App;
64
```

Ứng dụng React

- Nếu gọi đồ ăn trước khi nhận JWT thì sẽ gặp lỗi.
- Nhưng nếu chúng ta gọi JWT trước, nó sẽ được lưu trữ trong bộ nhớ cục bộ và ở trạng thái thành phần cục bộ của chúng ta.
- Sau đó chúng tôi có thể thực hiện yêu cầu.
- Mã thông báo đang được đính kèm với yêu cầu bằng cách thiết lập bộ chặn HTTP với **axios** .
- Nó tìm kiếm xem liệu yêu cầu gửi đi có phải là một nguồn gốc mà chúng tôi đã xác định trước là được phép hay không và sau đó đính kèm JWT của người dùng vào Authorizationtiêu đề nếu có.

Refactor để lưu trữ JWT trong Cookies

- Bước đầu tiên để chuyển sang sử dụng cookie là yêu cầu API của chúng tôi đặt cookie trong trình duyệt của người dùng sau khi họ đăng nhập thành công.
- Cookie được đặt trong trình duyệt nếu phản hồi cho một lệnh gọi HTTP có chứa Set-Cookie tiêu đề. Tiêu đề này sẽ có một chuỗi các tên và giá trị cookie, cộng với bất kỳ cài đặt bổ sung nào cho cookie (như liệu chúng có nên là HttpOnly hay không).
- Trong *API Express* của bạn, hãy bắt đầu bằng cách cài đặt cookie-parser. Đó là một phần mềm trung gian nhanh cho phép chúng tôi phân tích cú pháp cookie theo các yêu cầu đến.
- Điều này sẽ giúp chúng tôi sau này khi chúng tôi cần đọc giá trị cookie để cấp quyền truy cập vào foods.

Refactor để lưu trữ JWT trong Cookies

- Cài đặt Cookie-parser cho API Express

```
❯ npm install --save cookie-parser
```

- Refactor lại mã nguồn trong file Server.js

Server.js > app.get("/foods") callback

```
10  app.get("/jwt", (req, res) => {
11    const token = jsonwebtoken.sign({ user: "johndoe" }, jwtSecret);
12    res.cookie("token", token, { httpOnly: true });
13    res.json({ token });
14  });
15
16  app.use(jwt({ secret: jwtSecret, algorithms: ["HS256"] }));
```

Sử dụng proxy HTTP

- Cho đến nay, ứng dụng React đã chạy trên cổng 3000 và API trên 3001.
- Điều này là tốt nếu bạn đang gửi JWT trong Authorization tiêu đề của lệnh gọi API của mình, nhưng vì bây giờ chúng tôi muốn gửi nó trong cookie, chúng tôi cần chạy hai ứng dụng trên cùng một cổng.
- Điều này là do cookie chỉ có thể đi đến nguồn gốc mà chúng đến.
- Vì sử dụng **create-react-app**, nên chúng tôi có thể thực hiện việc này khá dễ dàng trong chế độ phát triển. Chúng ta chỉ cần đặt URL API làm proxy giá trị trong package.json.

```
{
  "proxy": "http://localhost:3001"
}
```

Sử dụng proxy HTTP

- Cấu trúc lại lệnh gọi đến /jwt điểm cuối để không còn đặt JWT trả về trong bộ nhớ cục bộ.
- Thay vào đó, bây giờ nó sẽ được đặt làm cookie. Chúng ta có thể giữ setJwt để có thể nhìn thấy JWT trên màn hình.

```
const getJwt = async () => {
  const { data } = await axios.get(`/jwt`);
  setJwt(data.token);
}
```

Xác thực JWT từ Cookies

- Bây giờ JWT đã ở trong cookie, nó sẽ tự động được gửi đến API trong bất kỳ lệnh gọi nào mà chúng tôi thực hiện với nó.
- Đây là cách trình duyệt hoạt động theo mặc định. Nhưng một lần nữa, chúng ta cần phải cung cấp giao diện người dùng và phụ trợ trên cùng một nguồn gốc để thực hiện điều này.
- Phần mềm trung gian xác thực JWT do **express-jwt** cung cấp sẽ tìm kiếm JWT trên Authorizationtiêu đề của yêu cầu theo mặc định.
- Hãy cập nhật nó để sử dụng một getTokenchức năng tùy chỉnh sẽ tìm kiếm mã thông báo trên cookie đến.

Xác thực JWT từ Cookie

- Hiệu chỉnh mã nguồn của file Server.js trong API Express

```
app.use(cookieParser);
app.use(
  jwt({
    secret: "secret123",
    getToken: (req) ⇒ req.cookies.token,
  })
);
```

Xác thực JWT từ Cookie

- Điều chỉnh getFoods của mình để đi thẳng đến /foods vì chúng ta đang sử dụng proxy.
- Bây giờ chúng ta sẽ thấy rằng chúng ta không còn gửi JWT dưới dạng Authorizationtiêu đề nữa.
- Thay vào đó, nó sẽ nằm trong một cookie.
- Hiệu chỉnh lại mã nguồn trong file App.js của React

```
const getFoods = async () => {
  try {
    const { data } = await axios.get(`/foods`);
    setFoods(data);
    setFetchError(null);
  } catch (err) {
    setFetchError(err.message);
  }
};
```

Chống tấn công với CSURF

- Để chống lại việc tấn công CSURF thường sử dụng thư viện CSURF cho API Express

```
❯ npm install --save csurf
```

- Hiệu chỉnh mã nguồn file Server.js của API Express

```
const csrfProtection = csrf({
  cookie: true,
});
app.use(csrfProtection);
app.get("/csrf-token", (req, res) => {
  res.json({ csrfToken: req.csrfToken() });
});
```

Chống tấn công với CSURF

- Bổ sung API POST: /foods trong file Server.js của API Express

```
app.post("/foods", (req, res) => {
  foods.push({
    id: foods.length + 1,
    description: "new food",
  });
  res.json({
    message: "Food created!",
  });
});
```

Chống tấn công với CSURF

- Hiệu chỉnh App.js của React

```
const [newFoodMessage, setNewFoodMessage] = useState(null);
const createFood = async () => {
  try {
    const { data } = await axios.post("/foods");
    setNewFoodMessage(data.message);
    setFetchError(null);
  } catch (err) {
    setFetchError(err.message);
  }
};
```

```
<section>
  <button onClick={() => createFood()}>
    Create New Food
  </button>
  {newFoodMessage && <p>{newFoodMessage}</p>}
</section>
```

Nhận và thiết lập thông báo CSRF

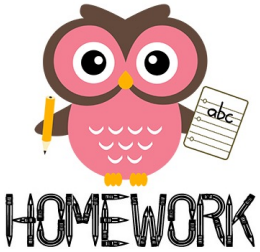
- Có một số cách khác nhau để chúng ta có thể lấy mã thông báo CSRF và đặt nó để sử dụng sau này.
- Một phương pháp phổ biến là đặt nó vào meta khi ứng dụng tải.
- Sau đó, nó có thể được đặt làm tiêu đề trong các yêu cầu sau này nếu cần.
- Vì chúng tôi đang làm việc từ gốc của ứng dụng React nên chúng tôi có thể gửi yêu cầu khi ứng dụng tải và giữ mã thông báo CSRF kết quả ở trạng thái ứng dụng.
- Sau đó, nó có thể được đặt làm tiêu đề mặc định theo POST yêu cầu của **axios**. Điều này có thể phù hợp hoặc không phù hợp với ứng dụng cụ thể của bạn.

Nhận và thiết lập thông báo CSRF

- Hiệu chỉnh mã nguồn của App.js trong React

```
useEffect(() => {
  const getCsrfToken = async () => {
    const { data } = await axios.get("/csrf-token");
    axios.defaults.headers.post["X-CSRF-Token"] = data.csrfToken;
  };
  getCsrfToken();
}, []);
```

Bài tập về nhà



- Xây dựng một ứng dụng React cho hiển thị thông tin sản phẩm khi người dùng đăng nhập vào hệ thống:
- Thang điểm:
  - Xây dựng API đăng nhập với user = admin, password = 123 (2 điểm)
  - Xây dựng API trả ra danh sách sản phẩm (Tên và giá) nếu người dùng đăng nhập đúng với tài khoản ở trên, còn không thì thông báo lỗi. (2 điểm)
  - Xâ dựng ứng dụng React hiển thị màn hình đăng nhập (2 điểm)
  - Xây dựng chức năng đăng nhập và lưu thông tin đăng nhập (2 điểm)
  - Nếu đăng nhập thành công thì hiện danh sách sản phẩm (2 điểm)