Client-Side Implementation of JWT Authentication for FastAPI

Overview

This document outlines the implementation of a client-side application that interacts with a FastAPI backend using JWT (JSON Web Token) authentication. It covers user authentication, token management, making authenticated requests, and handling token expiration.

1. Setting Up Your Frontend

To create the client-side application, you can use any frontend framework. This guide assumes you are using React.

Installation

If you haven't already set up a React application, you can create one using Create React App:

```bash
npx create-react-app my-app
cd my-app
```

2. Storing Tokens Securely

After successfully logging in, you need to store the received access and refresh tokens.

Token Storage Strategy

- Access Token: Store it in memory or session storage since it is short-lived.
- Refresh Token: Store it in local storage or cookies (preferably with HttpOnly flag).

Example Code

```javascript
// After a successful login
const handleLogin = async (username, password) => {
    const response = await fetch('http://localhost:8000/token', {
        method: 'POST',
```

```
        headers: {
            'Content-Type': 'application/json',
        },
        body: JSON.stringify({ username, password }),
    });

    if (response.ok) {
        const data = await response.json();
        localStorage.setItem('access_token', data.access_token);
        localStorage.setItem('refresh_token', data.refresh_token);
    } else {
        console.error('Login failed:', response.statusText);
    }
};
```

3. Making Authenticated Requests

To access protected routes, include the access token in the Authorization header.

Example of an Authenticated Fetch Request

```javascript
const fetchProtectedData = async () => {
    const accessToken = localStorage.getItem('access_token');

    const response = await fetch('http://localhost:8000/protected-route', {
        method: 'GET',
        headers: {
            'Authorization': `Bearer ${accessToken}`,
```

```
        },
    });

    if (response.ok) {
        const data = await response.json();
        console.log('Protected data:', data);
    } else if (response.status === 401) {
        console.error('Token expired or invalid');
        // Handle token expiration (e.g., refresh the token)
    } else {
        console.error('Error fetching protected data:', response.statusText);
    }
};
```

4. Handling Token Expiration

Since access tokens are short-lived, implement a mechanism to refresh the access token using the refresh token.

Example of Refreshing the Access Token

```javascript
const refreshAccessToken = async () => {
    const refreshToken = localStorage.getItem('refresh_token');

    const response = await fetch('http://localhost:8000/refresh', {
        method: 'POST',
        headers: {
            'Content-Type': 'application/json',
```

```
        },
        body: JSON.stringify({ refresh_token: refreshToken }),
    });

    if (response.ok) {
        const data = await response.json();
        localStorage.setItem('access_token', data.access_token);
    } else {
        console.error('Refresh token failed:', response.statusText);
        // Handle refresh token expiration or invalidation (e.g., prompt user to log in again)
    }
};
```

5. Logging Out

When a user logs out, clear the stored tokens to prevent unauthorized access.

Example of a Logout Function

```javascript
const handleLogout = () => {
    localStorage.removeItem('access_token');
    localStorage.removeItem('refresh_token');
    console.log('User logged out');
};
```

6. Putting It All Together

Here's how the complete flow might look in a React component:

Complete React Component Example

```javascript
import React, { useState } from 'react';

const AuthComponent = () => {
    const [username, setUsername] = useState('');
    const [password, setPassword] = useState('');

    const handleLogin = async () => {
        // Handle login logic as shown above
    };

    const handleLogout = () => {
        // Handle logout logic as shown above
    };

    const fetchData = async () => {
        // Fetch protected data as shown above
    };

    return (
        <div>
            <input
                type="text"
                value={username}
                onChange={(e) => setUsername(e.target.value)}
                placeholder="Username"
            />
```

```
        <input
          type="password"
          value={password}
          onChange={(e) => setPassword(e.target.value)}
          placeholder="Password"
        />
        <button onClick={handleLogin}>Login</button>
        <button onClick={handleLogout}>Logout</button>
        <button onClick={fetchData}>Fetch Protected Data</button>
      </div>
    );
};

export default AuthComponent;
```

Best Practices

1. Security:
   - Use HTTPS to protect tokens during transmission.
   - Store refresh tokens in HttpOnly cookies for additional security.

2. Token Management:
   - Regularly rotate refresh tokens and implement token expiration policies.
   - Log users out globally when they revoke their tokens.

3. Error Handling:
   - Implement comprehensive error handling for all API requests.

4. User Experience:

- Provide feedback to users during authentication and data retrieval processes, such as loading indicators and error messages.

Summary

This document provides a comprehensive overview of the client-side implementation of JWT authentication for a FastAPI application. By following the outlined steps, you can create a secure and efficient authentication system that enhances user experience while protecting sensitive resources.