Documentation: Secure API Authentication Flow

1. Overview

This document outlines the architecture and dataflow of the authentication system used in the project. The system is designed for modern Single Page Application (SPA) architecture, with a **React frontend** and a **Laravel backend** communicating via a RESTful API.

The core strategy is a **hybrid token-based approach** that uses both stateless and stateful concepts to provide a high level of security and a seamless user experience.

Key Characteristics:

- **Stateless API Calls:** Standard API requests are authenticated using a short-lived **Bearer Token**, requiring no server-side session.
- **Secure Token Refresh:** Session persistence is managed by a long-lived **Refresh Token** stored securely in an HttpOnly browser cookie, making it inaccessible to client-side JavaScript and protecting it from XSS attacks.
- **Database-Backed Tokens:** All tokens are stored and validated against the personal_access_tokens table, allowing for server-side revocation and auditing.
- **Silent Authentication:** The frontend automatically refreshes expired access tokens in the background without interrupting the user.

2. The Authentication Dataflow

This is the step-by-step lifecycle of a user session, from login to logout.

1. **Login Request:**
    - **React:** The user submits their email and password. The React app sends a POST request to the /api/login endpoint.
    - **Laravel:** The AuthController@login method validates the credentials.

2. **Token Generation:**
    - **Laravel:** Upon successful validation, Laravel performs two key actions:
        1. It generates a short-lived **Access Token** (e.g., expires in 15 minutes).
        2. It generates a long-lived **Refresh Token** (e.g., expires in 7 days).
    - Both tokens are saved as records in the personal_access_tokens database table.

3. **Token Delivery:**
    - **Laravel:** The server sends a 200 OK response with two distinct parts:
        1. The **Access Token** is placed in the JSON response body.
        2. The **Refresh Token** is sent in a Set-Cookie header, configured as HttpOnly, Secure, and SameSite.
    - **React:** The React app receives the response. It extracts the access token from the JSON body and stores it in its state management (e.g., Zustand, Redux). The browser automatically and securely stores the refresh token cookie.

4. **Authenticated API Calls:**
    - **React:** For any subsequent request to a protected endpoint (e.g., GET /api/user), a specially configured Axios instance automatically attaches the access token to the request header: Authorization: Bearer <access_token>.
    - **Laravel:** The auth:sanctum middleware intercepts the request, validates the Bearer Token, and authenticates the user.

5. **Access Token Expiration & Refresh:**
    - **Laravel:** Eventually, the access token expires. The auth:sanctum middleware rejects the request with a 401 Unauthorized status.
    - **React:** An Axios response interceptor catches the 401 error. Instead of failing, it pauses the original request and triggers a POST request to the /api/refresh endpoint.
    - **Browser:** The browser automatically attaches the HttpOnly refresh token cookie to this /refresh request.
    - **Laravel:** The AuthController@refreshToken method manually reads the cookie, validates the token against the database, and generates a **new access token**. It returns this new token in the JSON response body.
    - **React:** The interceptor receives the new access token, updates its state, and automatically retries the original API request that failed. The user experiences no disruption.

6. **Logout:**
    - **React:** The user clicks "Logout." The app sends a POST request to /api/logout, including the current access token.
    - **Laravel:** The AuthController@logout method deletes all of the user's tokens from the personal_access_tokens table and sends a response with a header that instructs the browser to clear the refresh token cookie.
    - **React:** The app clears the access token from its state and redirects the user to the login page.

3. Laravel: Backend Perspective

The backend is configured to be a **sessionless API** that manually manages the refresh token lifecycle.

Configuration (bootstrap/app.php & routes/api.php)

- The bootstrap/app.php file is configured **without** the StartSession middleware for API routes. This ensures the API does not create or rely on traditional web sessions.
- The /api/refresh route in routes/api.php is uniquely protected by the EncryptCookies and AddQueuedCookiesToResponse middleware. This allows it to decrypt and process the incoming refresh token cookie while the rest of the API remains unaware of cookies.

Controller Logic (AuthController.php)

- **login():**
  - Validates credentials against the users table.
  - Deletes any pre-existing tokens for the user to enforce a clean login.
  - Uses $user->createToken() to generate both an access_token and a refresh_token. The tokens are stored in the database.
  - Returns the access_token in the JSON response.
  - Uses Laravel's cookie() helper (or in your case, a manual header) to construct and attach the HttpOnly refresh_token cookie to the response.

- **refreshToken():**
  - This method does **not** use auth:sanctum. It is a public route (protected only by the cookie middleware).
  - It manually retrieves the cookie using $request->cookie('refresh_token').
  - It parses the token string (<id>|<token>), finds the token in the personal_access_tokens table by its ID, and performs a secure hash_equals comparison.
  - It verifies the token's name is 'refresh_token' and that it hasn't expired.
  - If valid, it issues a new access token and returns it.

- **logout():**
  - This is a protected route. It uses the incoming access token to identify the user.
  - It deletes all tokens associated with that user from the database.
  - It returns a response with a header to expire and clear the refresh_token cookie on the client's browser.

4. React: Frontend Perspective

The frontend is responsible for state management of the access token and gracefully handling token expiration.

State Management

- The **access token** received from the /login or /refresh endpoint is stored in a global state (e.g., using Zustand or React Context). It must **never** be stored in localStorage due to XSS vulnerabilities.
- The frontend is completely unaware of the refresh token. The browser manages the cookie automatically and securely.

Axios Configuration

- **Public Instance:** A basic Axios instance is used for public routes like login and register.
- **Private Instance:** A second, separate Axios instance is created for protected API calls. This instance uses an **request interceptor** to automatically inject the Authorization: Bearer <token> header into every outgoing request, pulling the token from the global state.

Handling Expiration (Axios Response Interceptor)

This is the most critical piece of the frontend logic. The "private" Axios instance is configured with a **response interceptor**.

1. The interceptor checks every response for a 401 Unauthorized status code.
2. If it detects a 401, it triggers a custom useRefreshToken hook.
3. The useRefreshToken hook sends a request to the /refresh endpoint. It is configured with withCredentials: true to ensure the browser sends the HttpOnly cookie.
4. After receiving a new access token, the hook updates the global state.
5. The interceptor then takes the original, failed request and re-sends it, now with the new, valid access token.

This entire process is asynchronous and invisible to the user.