COLLEGE CODE:9222

COLLAGE NAME:Theni kammavar sangam college of technology

DEPARTMENT:IT

STUDENT NM_ID:8F04FBB37A676E8A686421632E013987

ROLL NO:922223205025

DATE:03/10/2025

Completed the project named as

Phase 3 NJ-JWT TOKEN REFRESH ON EXPIRY

SUBMITTED BY,

NAME:Mohanraj.R

MOBILE NO:8489395548

PROBLEM UNDERSTANDING & REQUIREMENTS

Tittle: NJ-JWT TOKEN REFRESH ON EXPIRY

1. Project Setup:

To set up JWT token refresh on expiry in a Spring Boot project, implement a dual-token system comprising a short-lived **Access Token** and a long-lived **Refresh Token** using the JJWT (Java JWT) library. Configure Spring Security to allow open access to login and refresh endpoints, and secure all other routes. Upon successful login, generate and return both tokens. When the Access Token expires, clients call a `/refresh-token` endpoint with the Refresh Token, which is validated to issue a new Access Token without re-authenticating. Tokens are signed using a secret key and include expiration timestamps. This setup enhances API security while maintaining user session continuity without frequent logins.

Database/Storage: Set up a database (e.g., MongoDB, PostgreSQL, or a simple array for quick testing) to store and manage **Refresh Tokens**. This is crucial for **revocation** and single-device logouts. The user's ID and the associated Refresh Token should be stored.

2. Core Features Implementation:

The core feature implementation for **JWT Token Refresh on Expiry** relies on a dual-token strategy: a short-lived **Access Token** and a long-lived **Refresh Token**. The system's functionality is defined by three primary server-side logic components: **Token Issuance**, **Authentication Middleware**, and the **Refresh Endpoint**.

- **Token Issuance (Login):** Upon successful user authentication, the server generates two distinct tokens. The **Access Token** (e.g., 15 minutes expiry) is used for accessing protected API resources and contains user claims; it is typically sent in the `Authorization` header. The **Refresh Token** (e.g., 7 days expiry) is more secure, contains minimal claims (usually just the user ID), is stored server-side (e.g., in a database) for revocation purposes, and is set as an **HTTP-only cookie** to mitigate XSS attacks.

- **Authentication Middleware:** This is a crucial function on all protected routes. It intercepts API requests, extracts the Access Token, and verifies its signature.
    - If the token is valid, the request proceeds.

    - If the token is **expired**, the server returns a specific status code (e.g., `401 Unauthorized`) that signals the client to initiate the refresh process.

- **Refresh Endpoint:** When the client receives the "token expired" signal, it automatically calls a dedicated `/refresh` endpoint.
    - The server attempts to read the Refresh Token from the secure **HTTP-only cookie**.
    - It verifies the Refresh Token's signature and checks its expiry.
    - It also checks the token against the stored list in the database to ensure it hasn't been revoked.
    - If all checks pass, the server generates a **new Access Token** and potentially a **new Refresh Token** (using **Refresh Token Rotation** for enhanced security), sends the new Access Token back, and replaces the old Refresh Token in the database and the client's cookie.
    - If any check fails, the user is forced to log in again.

3. Data Storage (Local State / Database):

Data storage for JWT token refresh involves a critical split between the client and the server to balance security and statelessness. The client primarily manages the short-lived **Access Token** in a relatively secure, volatile location, such as **in-memory** storage or **Session Storage**, as it's passed with every API request. Conversely, the long-lived **Refresh Token** is managed more securely. On the client side, it's typically stored in an **HTTP-only cookie** set by the server, which makes it inaccessible to client-side JavaScript, mitigating XSS risks. On the server side, the Refresh Token (or a hash/ID referencing it) **must be stored in a persistent database** (like SQL or NoSQL) along with the associated User ID and an expiration timestamp. This database storage is essential for implementing **token revocation** (e.g., when a user logs out or changes their password) and for validating that the token is still active during the refresh process, ensuring the overall security and control of user sessions.

Client-Side Storage (Local State)

The client-side storage focuses on securing the tokens necessary for API communication and the refresh process:

- **Access Token:** This **short-lived token** is used for every request to protected API routes. It should be stored in the most volatile and secure local state possible to minimize its exposure in case of a breach:
    - **In-Memory Variable:** The most secure method, storing the token in a JavaScript variable that is cleared when the page is closed or refreshed. This requires fetching a new token upon every page load (via the Refresh Token).
    - **Session Storage:** A slightly less secure but more persistent option, allowing the token to persist across page navigations within the same browser session.

Local Storage is generally discouraged due to its vulnerability to Cross-Site Scripting (XSS) attacks.

- **Refresh Token:** This **long-lived token** is used only once the Access Token expires to request a new one. For security, it **should not be accessible via JavaScript** (mitigating XSS). The standard and recommended storage method is:
  - **HTTP-Only Cookie:** The server sets the Refresh Token as a cookie with the `HttpOnly` flag. This flag prevents any client-side script from reading or manipulating the token, ensuring it is only sent automatically with requests to the server's refresh endpoint.

Server-Side Storage (Database)

The server must maintain a persistent record of the Refresh Tokens for security and session control, negating the "stateless" nature of JWTs only for the refresh mechanism itself:

- **Refresh Token Record:** The server stores the Refresh Token (or a unique ID associated with it) in a **persistent database** (SQL or NoSQL). This record must be linked to the user and include:
  - **User ID:** To identify the owner of the token.
  - **Token Value:** The token itself (or a secure hash of it).
  - **Expiration Timestamp:** The actual expiry time, which is validated during the refresh request.
  - **Revocation Status (Optional):** A flag to instantly invalidate the token.
- **Purpose:** This database storage is crucial for:
  - **Revocation:** Allowing the token to be immediately invalidated upon user action (e.g., logout, password change).
  - **Validation:** Ensuring the Refresh Token presented by the client is the valid, non-expired, and most recently issued one (especially when using **Refresh Token Rotation**).

4. Testing Core Features:

- Query successful

Try again without apps

Testing the core features of a JWT Token Refresh on Expiry mechanism is crucial to ensure both **security** and a **seamless user experience**. The tests must validate the intended behavior across the three main components: Token Issuance, Authentication Middleware, and the Refresh Endpoint.

The most effective way to test expiry-related features is to temporarily **set very short expiry times** (e.g., 5 seconds) for the Access and Refresh Tokens in a testing configuration, allowing tests to quickly simulate token expiration.

▯ **Initial Token Issuance Verification (Positive) ✅:** Verify that upon successful login, the server issues a short-lived **Access Token** (in the response body) and a long-lived **Refresh Token** (in an **HTTP-only cookie**).

▯ **Valid Access Token Usage (Positive) :** Confirm that the newly issued Access Token can successfully access a protected API resource immediately after issuance.

▯ **Access Token Expiry Detection (Simulated) ▯:** Configure a very short Access Token lifespan (e.g., 5 seconds) and verify that a protected API call using the expired token returns the expected "Token Expired" status (e.g., 401 or a custom error code).

▯ **Successful Token Refresh (Core Flow) :** Following expiry detection, verify that the client's automated call to the dedicated `/refresh` endpoint, providing the valid Refresh Token, successfully receives a **new Access Token**.

▯ **Refresh Token Re-use Prevention (Security) :** If **Refresh Token Rotation** is implemented, verify that the old Refresh Token is immediately invalidated after one successful refresh use, and any subsequent attempt to use it is denied (401).

▯ **Refresh Token Signature Validation (Security) ✍:** Verify that a tampered Refresh Token (even if unexpired) is rejected by the server, demonstrating signature verification logic.

▯ **Refresh Token Expiry (Negative) :** Verify that a Refresh Token that has exceeded its long-term lifespan is rejected by the server, forcing a full user re-login.

▯ **Manual Revocation/Logout (Security) :** Verify that after a user logs out, the associated Refresh Token is successfully deleted from the server-side database, and any subsequent refresh attempt is denied (401).

▯ **Concurrent Session Handling (Security) :** Test scenarios where a user is logged in on multiple devices/browsers and verify that revoking the Refresh Token for one session does **not** unintentionally invalidate the others (unless a global logout is intended).

▯ **New Token Functionality (Positive) ✨:** Verify that the newly acquired Access Token (after a successful refresh) is immediately functional and can access protected API resources.

5. Version Control (GitHub):

1.Repository Structure

The structure should clearly separate the client and server components, as they handle different tokens and security logic:

- **Monorepo:** A single repository containing two main directories (e.g., `/server` and `/client`). This is often preferred for token refresh, as the client's logic (detecting 401 and calling /refresh) is tightly coupled with the server's implementation.
- **Polyrepo (Separate Repositories):** If the client and server are developed by different teams or deployed independently, two separate repositories should be used. The server repo is crucial, as it contains the sensitive logic for token generation, revocation, and the database interaction.

2. Branching Strategy

A robust strategy, like **Gitflow** or a modified **Trunk-Based Development**, is necessary to manage secure feature development:

- **Feature Branches:** All work on the refresh logic (e.g., `feature/implement-refresh-endpoint`, `fix/token-expiry-bug`) should happen in isolated branches.
- **Pull Requests (PRs) and Code Review: Mandatory PRs** are vital. Reviews must strictly scrutinize:
    - **Refresh Token Handling:** Ensuring the Refresh Token is set as an **HTTP-only cookie** and is *never* exposed to client-side code.
    - **Secret Management:** Checking for hardcoded secrets, especially the JWT signing keys.
    - **Database Interaction:** Verifying that the Refresh Token is correctly stored, validated, and revoked.
- **Deployment Flow:** Only merging into `main` (or `develop`/`release`) after thorough testing.

3. Security and Secrets Management

GitHub and Git are primarily designed for code, not secrets. Dedicated practices must be enforced to protect the JWT secrets:

- **Exclusion from Git:** Sensitive secrets (e.g., `JWT_ACCESS_SECRET`, `JWT_REFRESH_SECRET`, database credentials) **must be excluded** using `.gitignore`.
- **Environment Variables:** Secrets should be loaded from **environment variables** (`.env` files for local development, which must be git-ignored) and secured in the CI/CD pipeline or deployment environment (e.g., GitHub Secrets, Kubernetes Secrets).
- **Git Hooks (Optional):** Pre-commit hooks can be configured to automatically scan code for known patterns of secrets before a commit is created.

4. Commit Hygiene

Clear, descriptive commit messages are essential for traceability, especially when security features are involved:

- **Server Commits:** Should explicitly mention the part of the flow being modified (e.g., "Server: Implement token rotation on refresh endpoint," or "Fix: Ensure access token validation checks 'exp' claim").
- **Client Commits:** Should detail the client's response to token state (e.g., "Client: Add interceptor to handle 401 response and trigger refresh," or "Client: Clear session on refresh token failure").