COLLEGE CODE:9222

COLLAGE NAME:Theni kammavar sangam college of technology

DEPARTMENT:IT

STUDENT NM_ID:8F04FBB37AR76E8A686421632E013987

ROLL NO:922223205025

DATE:10/10/2025

Completed the project named as

Phase 4 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. Additional Features :

🔐 1. Token Blacklisting (Logout & Revoke Support)

- Maintain a blacklist (e.g., in Redis or DB) to store invalidated tokens.
- When user logs out or changes password, add token to blacklist.
- Middleware checks if a token exists in the blacklist before accepting it.

🔁 2. Auto Token Refresh (Silent Refresh)

- When the access token is about to expire (e.g., 1 minute before), automatically refresh using the refresh token without user intervention.
- Implement in frontend with interceptors (e.g., Axios interceptor in React).

📆 3. Token Expiry Tracking

- Store the access token expiry timestamp in localStorage or sessionStorage.
- Use this to show countdowns or prompt users before auto-logout.

🧑‍💻 4. Rotating Refresh Tokens

- Every time a refresh request is made, issue a new refresh token.
- Invalidate the old refresh token.
- Prevents token reuse and enhances security.

? 5. Device or Session-Based Refresh Tokens

- Bind refresh tokens to a specific device or browser.
- Store device ID or fingerprint hash in DB with the refresh token.
- Useful to detect unusual login activity.

🕵️‍♂️ 6. IP & User-Agent Binding

- Store IP address and user-agent with the refresh token.
- On refresh, compare stored values with current request to detect anomalies.

🧱 7. Refresh Token Expiry Limit

- Set a maximum lifetime for refresh tokens (e.g., 30 days).
- After that, user must log in again, even if refresh token is valid.

⚙ 8. Role-Based Access Control (RBAC) in JWT

- Include user roles or permissions inside the JWT payload.
- Example: `{ role: "admin", permissions: ["read", "write"] }`
- Backend routes validate role claims directly from the token.

⍰ 9. Middleware for Token Validation

- Centralize token validation in middleware.
- Automatically handle:
  - Missing/expired tokens
  - Token refresh logic
  - Token blacklist checks

📜 10. Detailed Token Logging

- Log token issuance, refresh, and revocation.
- Store logs with timestamp, user ID, IP, and token type.

⍰ 11. Grace Period for Expiry

- Allow a short grace period (e.g., 5–10 seconds) after token expiry to prevent race conditions during auto-refresh.

⚡ 12. Rate Limiting Refresh Requests

- Prevent multiple refresh attempts within a short time (e.g., 1 refresh every 30 seconds).

- Protects against brute force or DDoS attacks.

2. UI/UX Improvements :

To enhance the UI/UX for JWT Token Refresh on Expiry, the authentication process should feel smooth and non-intrusive. Implementing a seamless auto-refresh mechanism ensures tokens are renewed in the background without disrupting the user's workflow. When the session nears expiry, a gentle prompt such as "Your session will expire soon. Stay signed in?" can appear, allowing users to extend their session easily. Adding subtle indicators like a countdown timer or progress bar helps users stay aware of their session status. If reauthentication becomes necessary, a silent refresh or redirect back to the user's last visited page improves continuity. Clear and friendly error messages, responsive login state management across tabs, and a simple, secure login interface create a trustworthy experience. Additionally, including options like "Remember Me," dark mode compatibility, and real-time toast notifications for session updates enhance both usability and accessibility. Overall, these UI/UX improvements make token refresh systems feel intuitive, reliable, and user-centered rather than technical or disruptive.

🎨 1. Seamless Auto-Refresh (No Disruption)

- Automatically refresh the token in the background when it's near expiry.
- Show no pop-ups unless the refresh fails.
- Use a loading spinner only if absolutely needed (like network delay).

🔔 2. Session Expiry Warning

- Show a gentle prompt like:

  "Your session will expire in 2 minutes. Stay signed in?"

- Buttons: **[Stay Signed In]** | **[Logout]**
- Trigger this 1–2 minutes before token expiry.

⌛ 3. Progress Indicator or Countdown

- Add a subtle countdown bar or timer to indicate remaining session time.
- Can be in profile dropdown or footer:

  "Session expires in 4:32"

3.API Enhancements :

To strengthen and optimize the JWT Token Refresh on Expiry mechanism, several API-level enhancements can be implemented. Introducing **rotating refresh tokens** increases security by invalidating old tokens each time a new one is issued, preventing token reuse attacks. Implementing **token blacklisting** or **revocation APIs** ensures that tokens can be invalidated immediately after logout or password reset. The API should also support **device-based session management**, allowing refresh tokens to be bound to specific devices or IP addresses. Adding **rate limiting** to refresh endpoints prevents abuse and brute-force attempts. For smoother performance, **auto token renewal endpoints** can be designed to silently refresh tokens before expiry, minimizing downtime for users. Additionally, embedding **role-based claims and permissions** in tokens allows APIs to perform quick authorization checks without frequent database lookups. Incorporating **detailed audit logging** for token generation, refresh, and revocation enhances traceability and security monitoring. Finally, ensuring **standardized response formats** (with clear status codes and structured JSON messages) improves integration consistency across front-end and back-end services. Together, these API enhancements create a secure, efficient, and developer-friendly token refresh system.

Middleware / Interceptor Enhancements (Client-Side)

Enhance your frontend (or API client) with:

- **Interceptor** that catches `401 Unauthorized`
- Automatically calls `/auth/refresh`
- Replays the original request with new `accessToken`

Token Expiry Handling on Server

- Access Token: Short TTL (e.g. 15m)
- Refresh Token: Long TTL (e.g. 7d)
- Store refresh tokens in DB (optional but recommended for logout & revocation)

4. Performance & Security Checks :

To ensure a **secure and performant JWT token refresh implementation**, you should perform a series of **security hardening** and **performance optimization** checks. Below is a detailed breakdown of **best practices, checks, and safeguards**.

Token Rotation (Recommended)

- Issue a **new refresh token** every time the refresh endpoint is hit.
- **Invalidate** the old refresh token (maintain a whitelist or blacklist).
- Prevent **replay attacks**.

Example:

```
POST /auth/refresh
Request:
{
  "refreshToken": "old_token"
}

Response:
{
  "accessToken": "new_access",
  "refreshToken": "new_refresh"
}
```

Secure JWT Claims

- Do not include sensitive info in JWT payload (even if encrypted).
- Include:
  - `sub` (subject/user ID)
  - `iat` (issued at)
  - `exp` (expiry).
  - `type` = `"access"` or `"refresh"`

Example :

```
{
  "sub": "user_id",
  "iat": 1633834800,
  "exp": 1633835700,
  "type": "refresh"
}
```

5.Testing of enhancements

The robustness and security of JWT token refresh mechanisms, comprehensive testing has been conducted around token expiry and refresh workflows. The enhancements focus on automatically refreshing the access token when it expires, using a valid refresh token. Key scenarios tested include successful token refresh upon access token expiry, handling of expired or tampered refresh tokens, prevention of replay attacks with previously used refresh tokens (particularly in rotating refresh token setups), and proper invalidation of tokens upon user logout. Additionally, behavior was verified when a refresh token is used before access token expiry, ensuring that redundant token issuance is avoided unless explicitly allowed. Logging and auditing were validated to ensure all refresh attempts are properly recorded. Security considerations such as HTTP-only cookies, short-lived access tokens, and rate limiting on refresh endpoints were also included in the evaluation. These tests confirm that the system now handles token expiry more securely and gracefully, ensuring a seamless user experience while maintaining strict security controls.

✅ Pre-conditions

- You have a **valid access token** and **refresh token**.
- JWT expiry times are known (e.g., access token = 15 mins, refresh token = 7 days).
- The backend exposes:
  - `/login` (to get tokens)
  - `/refresh` (to refresh token using refresh token)
  - Protected endpoints (require valid access token)

🔍 Test Scenarios

🔁 1. Valid Refresh Flow

Steps:

1. Log in and obtain access & refresh tokens.
2. Wait for the access token to expire (or simulate expiry).
3. Send the expired access token with the refresh token to `/refresh`.
4. Assert a new access token is returned.
5. Use the new token to access a protected endpoint.

✅ **Expected:**

- New access token issued.
- Access to protected resources continues.

⌛ 2. Access Token Not Expired

Steps:

- Send refresh request before token expiry.

✅ **Expected:**

- Refresh endpoint may reject the request or return the same token depending on policy.
- Should not issue a new token if current is still valid (unless explicitly allowed).

⛔ 3. Expired Refresh Token

Steps:

1. Let both access and refresh tokens expire.
2. Try hitting `/refresh` with expired refresh token.

✅ **Expected:**

- Server returns 401 Unauthorized or custom error.
- Requires re-authentication (login again).

Deployment(Nrtlify, Vercel, or Cloud platform)

Deployment (Netlify, Vercel, or Cloud Platform) for JWT Token Refresh on Expiry

The JWT token refresh-on-expiry feature can be effectively deployed using modern platforms such as **Netlify**, **Vercel**, or any major **cloud provider** (e.g., AWS, Google Cloud, or Azure). For frontend applications hosted on Netlify or Vercel, the token refresh logic is typically implemented in client-side code or middleware, ensuring access tokens are refreshed silently when they expire. If backend functions (e.g., refresh token endpoints) are needed, **Serverless Functions** on these platforms can be used to handle secure token issuance and validation. Alternatively, deploying the backend to a cloud platform using containers or serverless services (like AWS Lambda, Google Cloud Functions, or Azure Functions) allows better scalability and control. Environment variables should be securely stored (e.g., refresh token secrets, JWT signing keys), and HTTPS is enforced by default. Proper **CORS configuration**, **secure cookie handling**, and **token expiration policies** should be maintained to ensure secure cross-platform integration. This deployment setup enables a seamless, scalable, and secure implementation of token refresh workflows across both frontend and backend systems.

1. Frontend Hosting with Netlify or Vercel

Deploy your frontend application (React, Vue, etc.) on Netlify or Vercel. Implement logic to detect JWT access token expiry and automatically call the refresh endpoint to obtain a new token before making further API requests.

2. Backend API via Serverless Functions

Use **Netlify Functions**, **Vercel Serverless Functions**, or cloud serverless platforms (e.g., **AWS Lambda**, **Google Cloud Functions**) to host secure API endpoints such as `/refresh` and `/login`. These endpoints handle JWT validation and refresh securely.

3. Secure Environment Variables

Store sensitive data such as JWT secret keys and refresh token secrets securely using each platform’s environment variable management system (e.g., Netlify Environment Variables, Vercel Project Settings, or AWS Secrets Manager).

4. HTTPS and Secure Cookies

All platforms enforce HTTPS by default, allowing you to securely transmit JWTs and refresh tokens. Use **HTTP-only**, **Secure** cookies for storing refresh tokens to prevent XSS attacks, and configure CORS correctly for cross-origin requests.

5. Scalability and Monitoring

Deploying on cloud platforms ensures scalability. Use logging and monitoring tools (like Vercel/Netlify Analytics or CloudWatch for AWS) to track token refresh activity, monitor failures, and prevent abuse (e.g., via rate limiting refresh requests).