COLLEGE CODE:9222

COLLAGE NAME:Theni kammavar sangam college of technology

DEPARTMENT:IT

STUDENT NM_ID:8F04FBB37A676E8A686421632E013987

ROLL NO:922223205025

DATE:24/09/2025

Completed the project named as

Phase 1 NJ-JWT TOKEN REFRESH ON EXPIRY

SUBMITTED BY,

NAME:Mohanraj.R

MOBILE NO:8489395548

PROBLEM UNDERSTANDING & REQUIREMENTS

Title:NJ-JWT TOKEN REFRESH ON EXPIRY

Tech Stack Selection:

When selecting a tech stack for handling token refresh on expiry (usually in the context of JWT authentication or similar systems), there are several factors to consider based on your existing tech stack, architecture, security requirements, and performance goals. Here's an approach to help guide your selection, along with some possible technologies

Key Requirements:

• **Token Expiry Handling**: Automatically detect expired tokens and initiate the refresh process.

• **Security**: Secure handling of refresh tokens, prevent replay attacks, and avoid exposing sensitive information.

•**Performance**: Minimize latency during the refresh process, ensuring a seamless user experience.

UI Structure / API Schema Design:

Designing the **UI structure** and **API schema** for handling token refresh on expiry involves planning out both the front-end (UI) and back-end (API) components to handle token expiration and refresh flows. Here's a breakdown of each.

UI Structure for Token Refresh on Expiry

Key Considerations for the UI:

- **Smooth User Experience**: The UI should handle token expiry gracefully without disrupting the user's experience.

- **Silent Token Refresh**: Ideally, the refresh process should happen automatically in the background without interrupting the user.

- **Error Handling**: If the refresh process fails, the user should be notified and prompted to re-authenticate (e.g., log in again).

UI Flow:

Login Screen:

- **Fields**: Username/Email, Password, Login Button.
- **Action**: The user logs in, receives an access token and refresh token.

Main Application Screen (Authenticated Area):

- **Token Handling**: The app makes authenticated API requests using the access token. If an access token expires, the UI automatically triggers a refresh flow.
- **Access Token Expiry Handling**:
  - Use an **API interceptor** (e.g., with Axios or Fetch API) to automatically detect a `401 Unauthorized` error when the access token expires.
  - Trigger a background request to refresh the access token using the refresh token stored in the browser's **HttpOnly cookie** or **localStorage**.

API Schema Design for Token Refresh

Overview:

The API should support the following key endpoints:

- **Login Endpoint**: To authenticate users and provide an access token and refresh token.
- **Token Refresh Endpoint**: To refresh the access token using the refresh token.
- **Logout Endpoint**: To revoke the refresh token and logout the user.

API Endpoints:

1. **Login Endpoint**:
   - **Path**: `POST /api/auth/login`
   - **Description**: Authenticates the user and returns an access token and refresh token.
   - **Request Body**:

```
{
  "username": "user@example.com",

  "password": "password123"

}
```

Response:

```
{

  "access_token": "jwt_access_token",

  "refresh_token": "jwt_refresh_token"

}
```

Data Handling Approach:

Designing a **data handling approach for token refresh on expiry** involves defining how you store, validate, and secure access and refresh tokens across your system. This includes both **client-side and server-side** concerns. Your goal is to ensure a **secure, scalable, and seamless experience** for users while protecting against token misuse.

Here's a structured data handling approach:

1. **Token Lifecycle:**

   Access Token (AT): Short lifespan (e.g., 5–15 minutes). Used for API calls.

   Refresh Token (RT): Longer lifespan (e.g., days/weeks). Used **only** to request a new AT.

2. Storage Strategy:

- **Access Token**:
  - Store in **memory** (preferred for security).
  - If persistence needed, store in **HTTP-only, Secure cookies** or encrypted local storage.
- **Refresh Token**:
  - Store securely in **HTTP-only, Secure cookies** (not accessible to JavaScript).

3. Refresh Flow:

□ **API Request with Access Token**:Send AT in request headers (e.g., `Authorization: Bearer <AT>`).

□ **Token Expiry Detection**:API returns `401 Unauthorized` or a specific "token expired" error.

4. Data Handling Approaches:

- **Client-Side Handling**
  - Keep track of AT expiry (JWT `exp` claim).
  - Implement a background **silent refresh** (e.g., refresh 1 minute before expiry).
  - On `401`, trigger refresh flow transparently.

5. Security Best Practices:

- Never expose Refresh Tokens to client-side JS.
- Rotate refresh tokens (invalidate old ones when issuing a new one).
- Implement **refresh token revocation** (logout, compromised account).
- Use HTTPS only.
- Shorten AT lifetime as much as possible.

Component/Module Diagram:

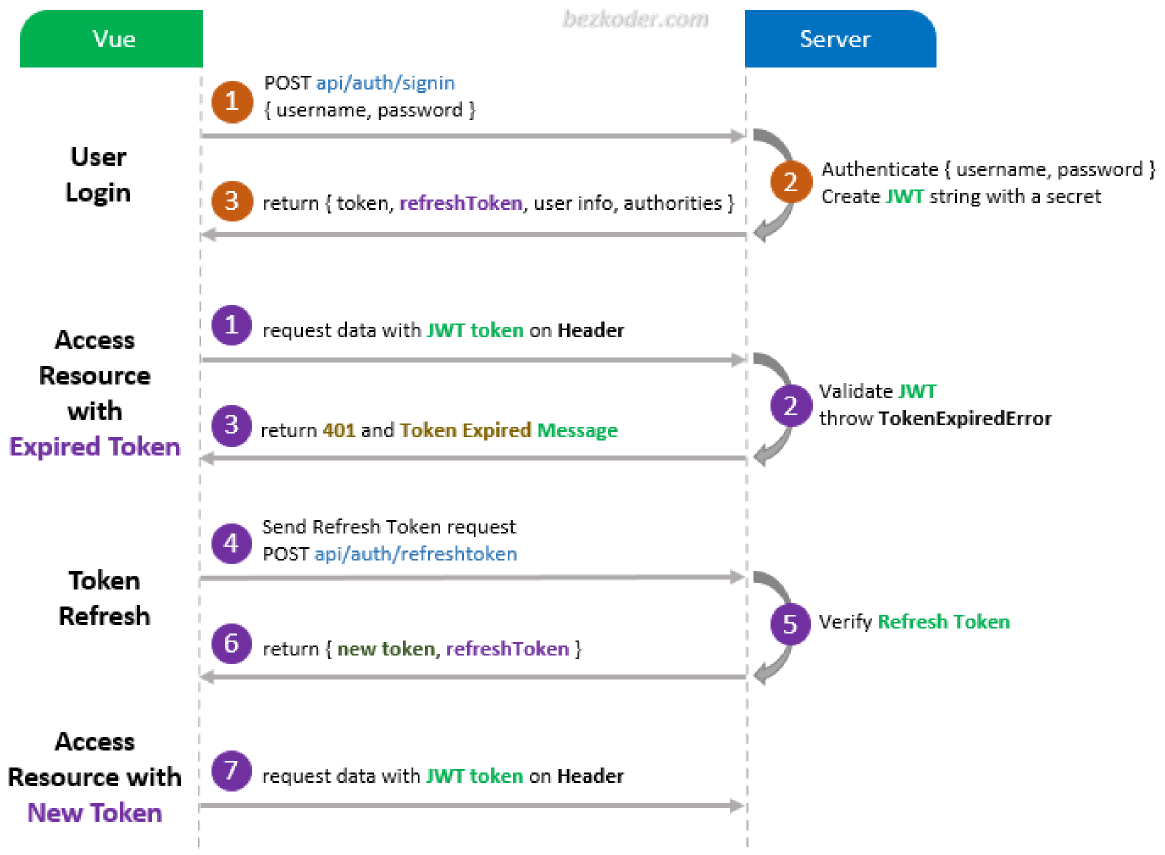
For a **Token Refresh on Expiry** system, the **Component/Module Diagram** should show how the **Client**, **Authentication Service**, and **Resource Server** interact to handle access and refresh tokens.

Here's a clear breakdown:

Main Components:

1. **Client Application (Frontend / Mobile App)**
   - **Token Manager Module**
     - Stores Access Token (memory / secure storage).
     - Tracks expiry (`exp` claim).
     - Triggers refresh process when needed.
   - **API Caller Module**
     - Sends requests with AT in headers.
     - On `401`, invokes Token Manager.
2. **Authentication Server (Auth Service / Identity Provider)**
   - **Login/Token Issuer** → Issues Access + Refresh tokens after login.
   - **Refresh Endpoint** → Accepts RT, validates, issues new AT (and RT if rotation).
   - **Revocation/Blacklist Module** → Invalidates RT on logout/compromise.
3. **Resource Server (Backend APIs / Protected Services)**
   - **Request Validator** → Validates Access Token.
   - **Access Control Module** → Authorizes based on user claims/roles.
   - Responds with data OR `401 Unauthorized` if AT is invalid/expired.

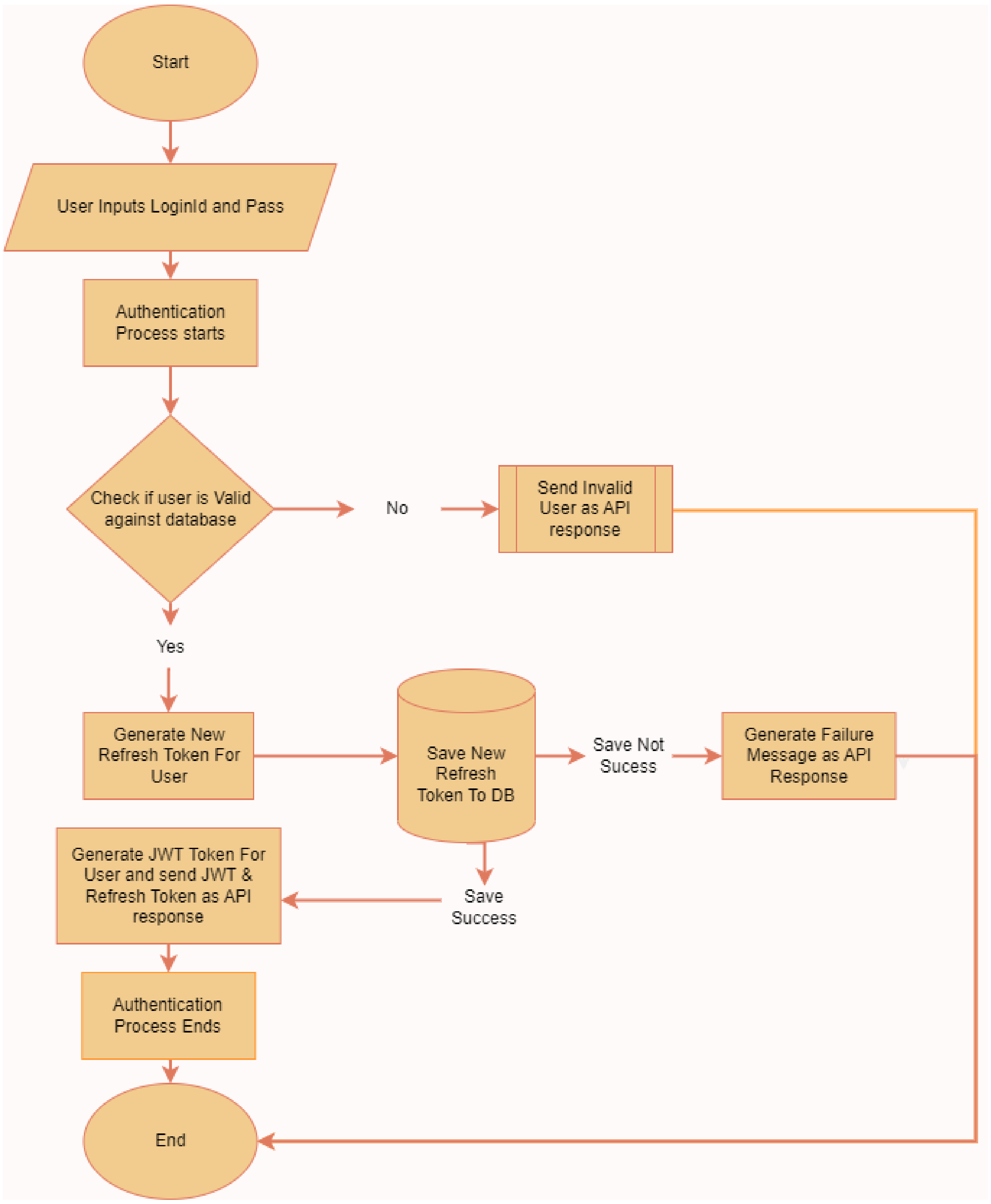
Component / Module Diagram (Textual Representation):


Vue
bezkoder.com
Server
User Login
1 POST api/auth/signin { username, password }
2 Authenticate { username, password } Create JWT string with a secret
3 return { token, refreshToken, user info, authorities }
Access Resource with Expired Token
1 request data with JWT token on Header
2 Validate JWT throw TokenExpiredError
3 return 401 and Token Expired Message
Token Refresh
4 Send Refresh Token request POST api/auth/refreshtoken
5 Verify Refresh Token
6 return { new token, refreshToken }
Access Resource with New Token
7 request data with JWT token on Header

Flow in Diagram

1. **Login** → Client gets AT + RT from Auth Server.
2. **API Call** → Client sends AT → Resource Server validates.
3. **Expiry Detected** → Resource Server returns `401`.
4. **Refresh** → Client sends RT → Auth Server validates RT, issues new AT.
5. **Retry** → Client retries API call with fresh AT.

Basic Flow Diagram:


Start
User Inputs LoginId and Pass
Authentication Process starts
Check if user is Valid against database
No
Send Invalid User as API response
Yes
Generate New Refresh Token For User
Save New Refresh Token To DB
Save Not Sucess
Generate Failure Message as API Response
Generate JWT Token For User and send JWT & Refresh Token as API response
Save Success
Authentication Process Ends
End

Flow Steps

1. **User Login** → Auth Server issues **Access Token (AT)** + **Refresh Token (RT)**.
2. **API Request** → Client sends AT → Resource Server validates.
3. **If AT Valid** → Resource Server returns data.
4. **If AT Expired** → Resource Server returns `401 Unauthorized`.
5. **Client Uses RT** → Sends RT to Auth Server `/refresh`.
6. **Auth Server Validates RT** → Issues new AT (and new RT if rotation enabled).