W9 *PRACTICE*

Full Auth (Backend + Frontend)

At the end of his practice, you should be able to…

- Implement full authentication flow with Backend and Frontend
- Understand how JWT secure your routes
- Understand various way to store JWT token in Frontend
- Manipulate UI based on auth state

How to start?

- ✓ Clone **start code** from git repositoryx
- ✓ Run `npm install` on both front and back projects
- ✓ Create .env and fill the info for the backend code based on .env.example
- ✓ Run `npm run dev` on both front and back projects to run the client and the server

How to submit?

- ✓ Submit your **code** on MS Team assignment

Are you lost?

JWT Playground
https://jwt.io/

VIDEOS
https://www.youtube.com/watch?v=7Q17ubqLfaM

*EXERCISE 1 – **Add Authentication Logics***

Objective:

- Add user registration and login routes using JWT.
- Secure routes so only authenticated users can access them.
- Update Swagger API docs to reflect authentication requirements.
- Reflect on the significance of token-based authentication in REST APIs.

Based on your W8 source code

https://github.com/KimangKhenng/school-api

Q1: Create Authentication Routes

1. **Create POST /register route**
    - Accept name, email, and password from the request body.
    - Hash the password using bcryptjs.
    - Store the user data (you can use an in-memory array or a database).
2. **Create POST /login route**
    - Accept email and password from the request body.
    - Check if the user exists and the password matches using bcryptjs.
    - If valid, generate a JWT token using jsonwebtoken.
    - Return the token to the client.

Q2: Add JWT Middleware

3. **Create a middleware function to validate JWT**
    - Extract the token from the Authorization header.
    - Verify the token using the secret key.
    - Reject unauthorized or invalid requests.
4. **Apply the middleware to protect existing routes**
    - Example: Only allow access to /users if the JWT is valid.

Q3: Update Swagger Documentation

5. **Add JWT security scheme in Swagger setup**
    - Define bearerAuth under Swagger components.securitySchemes.
    - Set it as a global or route-specific requirement.
6. **Document the following routes in Swagger**
    - /register (no authentication required)
    - /login (no authentication required)
    - /users (JWT authentication required)

Q4: Reflective Questions (write your answers)

7. What are the main benefits of using JWT for authentication?
8. Where should you store your JWT secret and why?
9. Why is it important to hash passwords even if the system is protected with JWT?
10. What might happen if a protected route does not check the JWT?
11. How does Swagger help frontend developers or API consumers?
12. What tradeoffs come with using token expiration (e.g., 1 hour)?

EXERCISE 2 – Implement React Frontend

For this exercise, you start with a start frontend and your backend code.
using the provided website as the preferred final result:
 https://school-management-cadt.netlify.app/

Objective:

- Track and manage authentication state using JWT stored in `localStorage`
- Show/hide UI components based on whether a user is authenticated
- Implement login and logout logic
- Redirect users appropriately upon authentication status change
- Create a shared `AuthContext` for global access to auth state

Prerequisites

- React project bootstrapped (start source code is given)
- Basic familiarity with `react-router-dom`, `localStorage`, and React hooks (`useState`, `useEffect`, `useContext`)

Q 1: Set Up Auth Context

Learn the use of context: https://dev.to/dayvster/use-react-context-for-auth-288g
https://legacy.reactjs.org/docs/context.html

1. Create `context/AuthContext.jsx`.
2. Create a context with:
   - `auth`: current user (null if not authenticated)
   - `setAuth`: function to update auth state
   - `loading`: boolean to show loading screen if needed
3. On mount, check `localStorage` for a JWT token and decode it if valid.
4. Provide the context to the app in `main.jsx`.

 Hint: use `useEffect(() => {}, [])` to check token existence on page load.

Q 2: Utility Functions in `utils/auth.js`

Create these helper functions:

- `isAuthenticated()`: returns decoded token if it exists and is not expired
- `logout()`: removes token from `localStorage`
- `getToken()`: get the raw JWT
- and more if needed

Q3: Show Navbar UI Based on Auth

1. Open `components/Navbar.jsx`.
2. Read the token from `AuthContext`.
3. If user is authenticated:
   - Show their name or role
   - Show "Logout" button
4. If not authenticated:
   - Show "Login" and/or "Register" links
5. Implement logout button click:
   - Clear the token
   - Reset auth state
   - Redirect to `/login`

Q4: Implement Login Page

1. Open `pages/Login.jsx`.
2. Add a form with username/password (Done)
3. On submit:
   - Send credentials to backend (mock API or working endpoint)
   - On success, store JWT in `localStorage`
   - Update `auth` state in context
   - Redirect to `/dashboard` or home page

 Only store the token if login is successful.

Q5: Protect Routes Based on Token

1. Create a wrapper component `ProtectedRoute.jsx` that:
   - Reads `auth` from context
   - If `auth` is null, redirect to `/login`
   - Else, render the children

```
// Example usage:
<ProtectedRoute>
  <Dashboard />
</ProtectedRoute>
```

Q6: Auto Redirect if Already Logged In

- On the `/login` page:
  - If the user is already authenticated, redirect them to `/dashboard` automatically.

Reflective Questions

1. Why do we use localStorage to store the JWT token instead of saving it in a React state? What are the advantages and risks?

2. How does the AuthContext improve the way we manage user authentication across different pages?

3. What would happen if the token in localStorage is expired or tampered with? How should our app handle such a case?

4. How does using a ProtectedRoute improve the user experience and security of the application?

5. What are the security implications of showing different UI elements (like "Logout" or "Dashboard") based on the token state? Could this ever leak information?