ChatGPT

Phân tích dự án An_Toan_Mang_May_Tinh_KMA

Giới thiệu

An_Toan_Mang_May_Tinh_KMA là một dự án quản lý điểm cho sinh viên Học viện Kỹ thuật Mật mã (KMA). Dự án được phát triển với cấu trúc **frontend-backend**: phía frontend là ứng dụng React/TypeScript (sử dụng React Router và Tailwind CSS), phía backend là API viết bằng **NestJS** (Node.js) kết hợp cơ sở dữ liệu PostgreSQL. Đặc biệt, dự án có đề cập đến tích hợp hệ thống **Identity and Access Management (IAM)**, cụ thể là sử dụng **Keycloak** – một giải pháp mã nguồn mở cho Single Sign-On (SSO) và quản lý danh tính người dùng [1]. Keycloak cho phép ứng dụng ngoài (frontend/backend) xác thực người dùng tập trung và cấp token để bảo mật các API.

Mục tiêu phân tích dưới đây là: (1) chỉ ra những **thiếu sót** hiện tại của dự án về tài liệu, mã nguồn, kiểm thử, cấu trúc và bảo mật; (2) đánh giá **tích hợp Keycloak** trong dự án (đã có hay chưa, cách tích hợp); (3) trình bày các **đoạn mã quan trọng** liên quan đến IAM/bảo mật (đặc biệt về Keycloak) kèm diễn giải dễ hiểu cho người mới học, đồng thời đề xuất cải tiến và tài liệu tham khảo giúp nhóm phát triển hoàn thiện dự án.

Các thiếu sót của dự án

- **Thiếu tài liệu:** Dự án không cung cấp tập tin README hay hướng dẫn cấu hình/chạy ứng dụng. Người dùng mới sẽ gặp khó khăn khi không có chỉ dẫn về cách cài đặt môi trường, khởi động frontend, backend, hay thiết lập Keycloak. Việc thiếu tài liệu cũng khiến cấu trúc và mục đích các thành phần (như thư mục `score-backend` hay `keycloak-26.2.4`) trở nên khó hiểu.

- **Chức năng đăng nhập chưa hoàn thiện:** Phía frontend hiện tại cho phép chọn vai trò (sinh viên, giảng viên, quản trị) rồi nhấn "Đăng nhập" nhưng **chỉ mô phỏng** đăng nhập, không thực sự kiểm tra thông tin với backend. Cụ thể, hàm `handleLogin` chỉ thực hiện `setTimeout` 1.5 giây rồi thông báo "Đăng nhập thành công" và chuyển hướng tới trang dashboard tương ứng, **không hề gọi API hay xác thực thông tin người dùng** [2]. Điều này cho thấy luồng đăng nhập chưa được triển khai đầy đủ – người dùng không cần nhập mật khẩu đúng vẫn "đăng nhập" được.

- **Chưa kết nối chặt chẽ giữa frontend và backend (SSO chưa được áp dụng ở giao diện):** Mặc dù backend có tích hợp Keycloak (phân tích bên dưới), phần giao diện web chưa tận dụng điều đó. Form đăng nhập của React **không** gọi tới Keycloak hoặc API đăng nhập của backend mà hoạt động độc lập. Điều này không chỉ làm trải nghiệm SSO không hoạt động, mà còn khiến các trang sau đăng nhập (dashboard) **không thực sự được bảo vệ** – hiện tại bất kỳ ai truy cập thẳng URL `/admin`, `/teacher`, `/student` trên frontend đều thấy nội dung vì ứng dụng không kiểm tra trạng thái đăng nhập. Nói cách khác, **frontend chưa có cơ chế xác thực/ủy quyền thực sự**.

- **Thiếu kiểm thử:** Dự án hầu như **chưa có bộ kiểm thử (test) đầy đủ** cho các chức năng chính, đặc biệt là về bảo mật và IAM. Hiện tại chỉ thấy một vài test mặc định do NestJS tạo sẵn khi scaffolding dự án (ví dụ test e2e kiểm tra endpoint gốc trả về "Hello World!") [3], và test đơn vị cho

AppController (sức khỏe hệ thống). Không có test nào cho luồng đăng nhập Keycloak hay kiểm tra phân quyền. Thậm chí, các test mặc định này chưa được cập nhật theo code (ví dụ, API gốc đã đổi thành trả về `{status: 'OK'}` nhưng test vẫn mong đợi chuỗi "Hello World!", dẫn đến thất bại nếu chạy) [3]. Việc thiếu và không đồng bộ kiểm thử cho thấy dự án chưa chú trọng đến đảm bảo chất lượng và an ninh.

- **Cấu trúc dự án chưa rõ ràng:** Mã nguồn được tổ chức thành nhiều phần nhưng **không được giải thích**. Thư mục `score-backend` chứa mã nguồn NestJS (backend), trong khi thư mục `src/` bên ngoài có mã React (frontend). Ngoài ra, còn có thư mục `keycloak-26.2.4` chứa **toàn bộ server Keycloak** (bao gồm các tệp cấu hình, script khởi động) [4]. Việc đưa cả bộ cài Keycloak vào repo là khá bất thường, khiến kho mã nặng nề và khó quản lý phiên bản. Cấu trúc monorepo (frontend + backend + IAM server) này không được giải thích, dễ gây nhầm lẫn cho người phát triển khác: làm sao để chạy toàn bộ hệ thống, thứ tự khởi động các thành phần, v.v. Nếu không đọc mã, rất khó đoán được cách thức các phần này tương tác.

- **Vấn đề bảo mật tiềm ẩn:** Một số điểm có thể gây rủi ro bảo mật hoặc vi phạm nguyên tắc IAM:

- *Lộ thông tin nhạy cảm:* File cấu hình Keycloak trong repo chứa thông tin kết nối DB Keycloak gồm username/password ở dạng plaintext (ví dụ: `db-username=postgres`, `db-password=3147` trong `keycloak.conf`) [5]. Mặc dù đây có thể chỉ là mật khẩu dev, việc commit thông tin nhạy cảm là không an toàn. Ngoài ra, nếu file .env (chưa thấy trong repo) có lưu `clientSecret` của Keycloak, thì cũng cần được giữ bí mật, không nên chia sẻ công khai.
- *Chưa cấp token sau đăng nhập:* Sau khi Keycloak xác thực, backend chỉ trả về thông tin người dùng dạng JSON mà **không cấp JWT** hoặc thiết lập phiên đăng nhập (session cookie) [6]. Điều này nghĩa là frontend không có bằng chứng nào để gửi kèm trong các request tiếp theo để chứng minh người dùng đã đăng nhập. Nếu các API khác của backend không được bảo vệ, người dùng có thể truy cập tùy ý; còn nếu API có bảo vệ thì hiện tại frontend cũng không gọi được do thiếu token. Việc thiếu bước cấp token khiến kiến trúc đăng nhập chưa hoàn thiện, dễ dẫn đến API không an toàn hoặc chức năng gián đoạn.
- *Chưa triển khai phân quyền (authorization):* Dự án có khái niệm vai trò (admin/teacher/student) ở giao diện, nhưng backend chưa áp dụng kiểm soát quyền dựa trên vai trò này. Mỗi người dùng trong DB chỉ lưu tên và email, không có trường vai trò; code backend cũng không kiểm tra quyền dựa trên vai trò Keycloak. Ví dụ, API `GET /api/users/:sub` hiện **không yêu cầu bất kỳ xác thực nào** – bất cứ ai gọi endpoint này với một `sub` hợp lệ đều nhận được thông tin người dùng tương ứng [7]. Điều này tiềm ẩn nguy cơ lộ dữ liệu nếu triển khai thực tế. Nên có guard yêu cầu người dùng phải đăng nhập (và có thể đúng vai trò) mới được truy cập thông tin người khác, nhưng dự án chưa xử lý.

Tóm lại, dự án cần cải thiện nhiều mặt: bổ sung tài liệu, hoàn thiện luồng đăng nhập sử dụng Keycloak đúng cách cho frontend, viết thêm kiểm thử, làm rõ cấu trúc triển khai, và khắc phục các điểm chưa an toàn (như quản lý secret và phân quyền API).

Tích hợp Keycloak trong dự án

Hiện trạng tích hợp: Dự án đã tích hợp Keycloak ở phía backend (NestJS) thông qua cơ chế xác thực OAuth2/OpenID Connect của Passport. Cụ thể, trong `score-backend` có module **Auth** cấu hình chiến lược Keycloak và guard để bảo vệ các route đăng nhập/callback. Điểm qua các phần tích hợp chính:

- **Chiến lược Keycloak (Keycloak Strategy):** Dự án định nghĩa một strategy Passport tùy biến cho Keycloak. File `keycloak.strategy.ts` cho thấy lớp `KeycloakStrategy` kế thừa `PassportStrategy` với strategy name là `'keycloak'`. Trong constructor, strategy được cấu hình bằng các tham số OIDC cần thiết: `clientID`, `clientSecret` ứng với client ứng dụng đăng ký trên Keycloak; `callbackURL` là URL mà Keycloak sẽ gọi lại sau khi người dùng đăng nhập; `realm` và `authServerURL` được suy ra từ URL issuer của Keycloak; đặt `idpLogout: true` để cho phép đăng xuất thông qua nhà cung cấp (Keycloak) [8]. Các giá trị này đều được lấy từ **ConfigService**, tức là có thể được cung cấp qua biến môi trường hoặc file cấu hình. Như vậy, KeycloakStrategy đóng vai trò như *cầu nối* giữa ứng dụng NestJS và máy chủ Keycloak, giúp Passport biết cách chuyển hướng và xử lý dữ liệu từ Keycloak.

Ngoài cấu hình, strategy còn triển khai hàm `validate()`. Đây là hàm sẽ được Passport gọi sau khi Keycloak xác nhận thành công người dùng và gửi về thông tin hồ sơ (profile). Trong code, `validate` tạo một đối tượng `user` gồm các thông tin cơ bản lấy từ `profile` của Keycloak (ID người dùng – gọi là **sub**, email, firstName, lastName) rồi gọi `done(null, user)` [9]. Việc này nhằm chuyển thông tin người dùng đã đăng nhập vào ứng dụng (gắn vào `req.user`). Hiện tại, `validate` chỉ đơn giản ánh xạ dữ liệu profile -> user, chưa kiểm tra hay bổ sung gì thêm.

- **Sử dụng AuthGuard cho luồng SSO:** NestJS tích hợp Passport thông qua **AuthGuard**. Dự án định nghĩa một **AuthController** với các endpoint phục vụ SSO:
- `GET /api/auth/login`: Gắn `@UseGuards(AuthGuard('keycloak'))` [10]. Khi người dùng truy cập đường dẫn này, NestJS sẽ tự động kích hoạt chiến lược `'keycloak'` đã đăng ký. Hiểu đơn giản, request sẽ được chuyển cho Passport-Keycloak Strategy, kết quả là người dùng được **chuyển hướng đến trang đăng nhập Keycloak**. (Hàm `login()` không có thân, vì toàn bộ logic chuyển hướng do Passport xử lý).
- `GET /api/auth/callback`: Cũng gắn guard tương tự [11]. Đây là URL để Keycloak gọi lại sau khi người dùng đăng nhập thành công (Keycloak sẽ redirect browser về đây kèm theo một Authorization Code). Guard `'keycloak'` sẽ nhận code đó, tự động liên hệ Keycloak để trao đổi lấy tokens và hồ sơ người dùng. Nếu hợp lệ, guard sẽ cho phép đi tiếp vào hàm `callback()`, đồng thời gắn thông tin user vào `req.user` (chính là object user do hàm `validate` ở strategy trả về). Trong `AuthController.callback`, code gọi `this.usersService.upsertFromKeycloak(req.user)` để lưu người dùng vào cơ sở dữ liệu hoặc cập nhật nếu đã có [6]. Sau đó, code **dự kiến** sẽ "issue JWT hoặc set session" (theo chú thích) nhưng hiện tại chỉ `res.json(user)` trả về thông tin người dùng dưới dạng JSON cho phía client xem [6].

Tóm lại, **luồng SSO** trên backend như sau: người dùng truy cập `/api/auth/login` → được chuyển đến Keycloak để đăng nhập → Keycloak xác thực xong chuyển về `/api/auth/callback` → Nest guard xử lý,

lấy thông tin user → lưu user vào DB → (thiếu bước) cấp token/thiết lập phiên → trả phản hồi. Dự án đã làm được phần lớn luồng này ngoại trừ bước cuối là cấp chứng thực cho phiên làm việc.

- **Thư viện và cấu hình liên quan:** Trong `package.json` của backend, dự án đã thêm dependency **passport-keycloak-oauth2-oidc** phiên bản 1.0.5 [12] – đây là strategy Passport hỗ trợ Keycloak (OIDC). Việc sử dụng thư viện này cho thấy đúng hướng tích hợp Keycloak. Ngoài ra, dự án sử dụng TypeORM cho DB và có entity User để lưu người dùng. **Thư mục** `keycloak-26.2.4` trong repo chứa bản cài đặt Keycloak server (phiên bản 26.2.4) bao gồm các file cấu hình, script khởi động, v.v. Điều này gợi ý rằng nhóm phát triển có chạy một instance Keycloak cục bộ khi lập trình, thay vì sử dụng Keycloak qua Docker hoặc cài đặt ngoài. Họ có thể khởi chạy Keycloak bằng cách chạy script `bin/kc.sh start-dev` trong thư mục này. Tuy nhiên, dự án không nêu rõ cách thức cấu hình realm/clients trong Keycloak – có lẽ những thông tin đó nằm ngoài code (vd: trong quá trình trình bày, nhóm sẽ nhập tay trên giao diện Keycloak).

Mức độ tích hợp: Từ phân tích trên, có thể thấy **dự án đã tích hợp Keycloak ở mức cơ bản phía backend**: sử dụng Keycloak làm nhà cung cấp danh tính, ứng dụng của chúng ta đóng vai trò client OAuth2, thực hiện redirect và callback để lấy thông tin người dùng. Tuy nhiên, tích hợp này **chưa được tận dụng đầy đủ**: - Backend chưa cấp token hoặc session sau khi nhận diện người dùng từ Keycloak, khiến việc bảo vệ các API khác chưa được thực hiện. - Frontend chưa được kết nối với hệ thống Keycloak – người dùng không thực sự tương tác với Keycloak khi đăng nhập. Lý tưởng, frontend nên chuyển hướng người dùng tới Keycloak hoặc mở cửa sổ đăng nhập Keycloak, nhưng hiện tại điều đó không xảy ra (frontend tách rời). - Chưa có phân quyền theo vai trò sử dụng dữ liệu từ Keycloak (ví dụ Keycloak có thể cung cấp thông tin roles của user, nhưng code chưa dùng đến).

Khả năng và vị trí tích hợp Keycloak (nếu chưa hoặc để hoàn thiện): Nếu giả sử dự án **chưa tích hợp** (ở một kịch bản khác), thì Keycloak nên được tích hợp tại **module Auth** của backend (để xử lý đăng nhập, bảo vệ các route nhạy cảm) và tại **phần đăng nhập của frontend** (để thực hiện SSO). Trong trường hợp dự án này, backend đã có sẵn tích hợp, do đó việc cần làm là **mở rộng tích hợp ra frontend và các phần còn thiếu**: - *Tại frontend:* Thay vì form đăng nhập tự xử lý, ứng dụng React nên sử dụng Keycloak cho SSO. Có thể thực hiện theo hai cách: (1) **Chuyển hướng thủ công**: Khi bấm "Đăng nhập", gọi tới endpoint `/api/auth/login` của backend (ví dụ bằng cách `window.location.href = "/api/auth/login"`), khi đó người dùng sẽ được đưa đến Keycloak, đăng nhập xong quay lại trang do backend chỉ định; hoặc (2) **Tích hợp SDK Keycloak JS**: Sử dụng thư viện `keycloak-js` trên frontend để cấu hình realm, client và tự điều khiển quá trình login (thư viện này sẽ mở trang Keycloak trong context, sau đó nhận token và cho phép truy cập tài nguyên). Cách thứ hai linh hoạt hơn và có thể quản lý phiên trên frontend dễ dàng [13]. Dù cách nào, kết quả là frontend sẽ nhận được **token** (JWT) do Keycloak phát hành (hoặc do backend phát hành dựa trên thông tin Keycloak) để sử dụng gọi API. - *Tại backend:* Bổ sung bước **phát hành token JWT** hoặc thiết lập **session** sau khi `upsertFromKeycloak`. NestJS có thể dùng thư viện `@nestjs/passport` và `@nestjs/jwt` để tạo JWT chứa các claim như userId, roles, thời hạn... ký bằng khóa bí mật của server. Token này trả về cho client (qua JSON hoặc set-cookie). Mọi request sau đó, client gửi token, và backend dùng **JwtAuthGuard** để kiểm tra và lấy thông tin user từ token. Như vậy, các API (ví dụ `/api/users/:sub` hay các API xem điểm, xem lịch) sẽ được bảo vệ – chỉ cho phép truy cập nếu request kèm JWT hợp lệ chứng minh người dùng đã đăng nhập. Ngoài ra, có thể tận dụng **roles** của Keycloak: trong token ID hoặc access token do Keycloak cấp thường có thông tin vai trò (realm roles hoặc client roles). Backend có thể đọc các claim này để quyết định quyền truy cập (ví dụ chỉ cho user có role “admin” gọi được API quản trị...). Nếu dùng JWT do ta tự phát hành, ta có thể lưu kèm role trong payload JWT khi tạo. - *Tích hợp logout:* Hiện tại frontend logout chỉ xóa trạng thái giả lập. Khi đã có SSO, cần tích hợp logout để đăng xuất cả phía Keycloak

(KeycloakStrategy đã bật `idpLogout: true` để hỗ trợ). Tức là khi người dùng bấm đăng xuất, frontend có thể gọi `/api/auth/logout` (chưa được dự án hiện tại cài, ta có thể tạo) để xóa session phía ứng dụng và **đăng xuất người dùng trên Keycloak** (thông qua endpoint end-session của Keycloak). Như vậy phiên đăng nhập bị hủy hoàn toàn.

Tóm lại, dự án đã **có nền tảng tích hợp Keycloak** ở backend, nhưng cần triển khai thêm ở frontend và hoàn thiện các bước xử lý sau xác thực để hệ thống SSO hoạt động trọn vẹn.

Các đoạn mã quan trọng về bảo mật/IAM

Dưới đây là một số đoạn mã tiêu biểu trong dự án liên quan đến tích hợp IAM (Keycloak) và bảo mật, kèm theo diễn giải:

- **KeycloakStrategy (backend)**: Đoạn mã sau được trích từ `score-backend/src/auth/keycloak.strategy.ts`, định nghĩa chiến lược Passport cho Keycloak:

```
import { Injectable } from '@nestjs/common';
import { PassportStrategy } from '@nestjs/passport';
import { Strategy, Profile, VerifyCallback } from 'passport-keycloak-oauth2-oidc';
import { ConfigService } from '@nestjs/config';

@Injectable()
export class KeycloakStrategy extends PassportStrategy(Strategy, 'keycloak') {
  constructor(private cs: ConfigService) {
    super({
      clientID: cs.get('keycloak.clientId'),
      clientSecret: cs.get('keycloak.clientSecret'),
      callbackURL: 'http://localhost:3000/api/auth/callback',
      realm: new URL(cs.get('keycloak.issuer') as string).pathname.split('/').pop(),
      authServerURL: (cs.get('keycloak.issuer') as string).replace(/\/realms.*/, ''),
      idpLogout: true,
    });
  }

  async validate(accessToken: string, refreshToken: string, profile: Profile, done: VerifyCallback) {
    const user = {
      sub: profile.id,
      email: profile.emails?.[0],
      firstName: profile.name?.givenName,
      lastName: profile.name?.familyName,
    };
    done(null, user);
```

```
  }
}
```

Đoạn mã trên định nghĩa một chiến lược Passport mang tên `'keycloak'`, sử dụng strategy từ thư viện Keycloak OAuth2 OIDC [14] [15]. - Trong phần cấu hình (`super({...})` trong constructor), `clientID` và `clientSecret` được lấy từ cấu hình ứng dụng (thông qua `ConfigService` – thường các giá trị này nằm trong biến môi trường). Đây là ID và secret của **client ứng dụng** đã đăng ký trên Keycloak. `callbackURL` đặt là URL mà Keycloak sẽ gọi đến sau khi người dùng đăng nhập xong (ở đây là `http://localhost:3000/api/auth/callback`). Thuộc tính `realm` được suy ra từ đường dẫn của `issuer` (issuer có dạng `http://localhost:8080/realms/<realm_name>` nên tách lấy phần `<realm_name>`), còn `authServerURL` là URL gốc máy chủ Keycloak (bằng cách bỏ đi đoạn `/realms/...` khỏi issuer) [8]. Việc đặt `idpLogout: true` cho phép khi logout, Passport sẽ gọi tới Keycloak để đăng xuất người dùng khỏi phiên SSO luôn. - Phương thức `validate` sẽ chạy sau khi Keycloak xác thực thành công và gửi thông tin về. Nó nhận các tham số: `accessToken`, `refreshToken` và `profile` (hồ sơ người dùng). Ở đây, ta tạo ra một object `user` đơn giản chứa các thông tin cần thiết từ profile (như **sub** – ID duy nhất của user trong Keycloak, email, tên riêng, tên họ) [9]. Sau đó gọi `done(null, user)` để hoàn tất quá trình – Passport sẽ gắn object `user` này vào `req.user` cho các bước xử lý sau. Lưu ý: ta không sử dụng đến `accessToken` hay `refreshToken` ở đây, nhưng có thể lưu chúng nếu muốn thực hiện gọi API Keycloak bổ sung. Hiện tại, chiến lược này chỉ lo việc **lấy thông tin cơ bản của user** sau khi đăng nhập.

Giải thích dễ hiểu: Bạn có thể hình dung KeycloakStrategy như một "cổng" kết nối NestJS với Keycloak. Khi người dùng cần đăng nhập, NestJS sẽ dùng chiến lược này để biết cần chuyển người dùng đến đâu (URL Keycloak, realm nào, client nào). Khi người dùng đăng nhập xong, Keycloak trả dữ liệu về, NestJS nhờ chiến lược này để **đọc dữ liệu người dùng** và gói gọn lại thành object JavaScript. Nhờ đó, phần còn lại của ứng dụng (các controller) có thể biết "ai đang đăng nhập" mà không cần quan tâm tới giao thức OIDC phức tạp bên dưới.

- **AuthController (backend)**: Đây là controller chịu trách nhiệm quản lý các endpoint liên quan đến xác thực. Mã nguồn (`score-backend/src/auth/auth.controller.ts`) trông như sau:

```
@Controller('api/auth')
export class AuthController {
  constructor(private usersService: UsersService) {}

  @Get('login')
  @UseGuards(AuthGuard('keycloak'))
  login() {
    // redirect to Keycloak
  }

  @Get('callback')
  @UseGuards(AuthGuard('keycloak'))
  async callback(@Req() req, @Res() res) {
    // req.user chứa thông tin từ strategy.validate
    const user = await this.usersService.upsertFromKeycloak(req.user);
    // issue JWT hoặc set session nếu muốn
```

```
    return res.json(user);
  }
}
```

Đoạn code trên cho thấy hai endpoint chính: `/api/auth/login` và `/api/auth/callback` [10] [6]: - `GET /api/auth/login`: Gắn guard `AuthGuard('keycloak')`. Guard này được cung cấp bởi NestJS Passport, khi kích hoạt sẽ tự động chuyển hướng người dùng đến trang đăng nhập Keycloak. Do đó, hàm `login()` không cần xử lý gì (phần comment "// redirect to Keycloak" chỉ để giải thích). Nói cách khác, chỉ cần người dùng truy cập đường dẫn này, cơ chế Passport-Keycloak sẽ lo phần còn lại (chuyển hướng người dùng ra khỏi ứng dụng để sang Keycloak xác thực). - `GET /api/auth/callback`: Cũng được bảo vệ bởi `AuthGuard('keycloak')`. Khi Keycloak xác thực xong, nó sẽ điều hướng về URL này kèm theo mã (authorization code). Guard sẽ chặn request này lại, dùng code đó để **liên hệ máy chủ Keycloak** (qua thư viện passport-keycloak) nhằm lấy thông tin người dùng. Nếu thành công, guard cho phép request đi tiếp và đồng thời gắn đối tượng user (kết quả của `validate` ở strategy) vào `req.user`. Lúc này, trong phần thân hàm `callback`, ta có thể truy cập `req.user`. Code đã triển khai: gọi `this.usersService.upsertFromKeycloak(req.user)` để lưu người dùng vào database (nếu user đã tồn tại thì cập nhật thông tin, chưa có thì tạo mới) [6]. Sau đó, định hướng tiếp theo (theo chú thích) là "phát hành JWT hoặc thiết lập session nếu cần". Tuy nhiên ở phiên bản hiện tại, code chỉ đơn giản `return res.json(user)`, gửi lại thông tin user dưới dạng JSON cho phía frontend.

Diễn giải: Như vậy, AuthController thiết lập hai đường dẫn cho quá trình SSO. Đầu tiên, khi người dùng muốn đăng nhập, frontend nên đưa họ tới `/api/auth/login`. Tại đây, người dùng sẽ được đưa sang trang đăng nhập của Keycloak (ví dụ trang yêu cầu nhập username/password do Keycloak cung cấp). Sau khi điền đúng thông tin, Keycloak sẽ chuyển trình duyệt về `/api/auth/callback` của ứng dụng. Khi đó, NestJS (nhờ guard) sẽ lấy thông tin người dùng từ Keycloak và gọi `upsertFromKeycloak` để lưu lại trong cơ sở dữ liệu ứng dụng. Cuối cùng, ứng dụng cần **tạo thông tin đăng nhập phiên** cho user (như JWT hoặc session). Dự án này chưa làm bước đó: trả về JSON nghĩa là phía client nhận được thông tin user nhưng không có token nào để dùng cho lần sau. Việc này giải thích tại sao ở phần thiếu sót chúng ta nói luồng login chưa hoàn thiện. Để hoàn thiện, ta cần sửa hàm callback này: thay vì `res.json(user)` trực tiếp, nên tạo JWT ký bằng khóa bí mật server, chứa ID user (hoặc chứa luôn toàn bộ profile nếu tiện) rồi trả về JWT đó cho client (ví dụ `{ token: <jwt> }`). Hoặc thiết lập cookie phiên và redirect người dùng về trang chủ. Những bước này giúp duy trì trạng thái đăng nhập cho các request kế tiếp.

- **UsersService.upsertFromKeycloak (backend)**: Sau khi xác thực, hệ thống gọi `usersService.upsertFromKeycloak()` để đồng bộ dữ liệu user. Định nghĩa hàm này trong `score-backend/src/users/users.service.ts` như sau:

```
async upsertFromKeycloak(dto: Partial<User>): Promise<User> {
  const existing = await this.repo.findOne({ where: { sub: dto.sub } });
  if (existing) {
    return this.repo.save({ ...existing, ...dto });
  }
  const user = this.repo.create(dto);
  return this.repo.save(user);
}
```

Hàm trên đảm nhiệm việc thêm mới hoặc cập nhật người dùng vào cơ sở dữ liệu ứng dụng [16]. Tham số đầu vào `dto` là một object Partial<User> – trong trường hợp này chính là object user được truyền từ `req.user` (gồm các thuộc tính `sub, email, firstName, lastName`). Logic của hàm: - Tìm trong bảng `users` xem có bản ghi nào với `sub` (subject) trùng khớp không. Trường `sub` ở đây chính là ID duy nhất của người dùng từ Keycloak (ví dụ Keycloak tạo một UUID cho mỗi user và cung cấp nó qua `profile.id`). Lưu ý, trong entity User, `sub` được dùng làm khóa chính (PrimaryColumn) để đảm bảo mỗi user Keycloak tương ứng một bản ghi user duy nhất [17]. - Nếu tìm thấy `existing` user, tiến hành cập nhật (`repo.save`) các thông tin mới từ `dto` đè lên (giữ nguyên các trường cũ không có trong dto). - Nếu không có user nào trùng `sub`, tạo mới (`repo.create`) và lưu vào DB. - Trả về user đã lưu (hoặc đã cập nhật).

Giải thích: Mục đích của upsert này là để mỗi khi người dùng đăng nhập thông qua Keycloak, ứng dụng sẽ lưu lại (hoặc cập nhật) thông tin của họ trong cơ sở dữ liệu riêng. Điều này hữu ích cho việc quản lý bổ sung trong ứng dụng (ví dụ gắn thêm dữ liệu khác cho user, hoặc đơn giản để có danh sách user nội bộ). Dùng `sub` làm khóa giúp chúng ta không tạo trùng lặp người dùng mỗi lần họ đăng nhập. Ngoài ra, nếu thông tin email hoặc tên của user trên Keycloak thay đổi, lần đăng nhập kế tiếp sẽ cập nhật vào DB ứng dụng. Lưu ý, hiện tại User entity không lưu mật khẩu hay vai trò – vì những thứ đó được quản bởi Keycloak, ứng dụng này chỉ lưu thông tin nhận dạng cơ bản.

- **Đăng nhập phía client (frontend)**: Để hiểu bối cảnh tích hợp IAM, ta xem qua cách frontend xử lý đăng nhập. File `src/pages/Index.tsx` chứa form đăng nhập và logic giả lập:

```
const handleLogin = async (role: string) => {
  setIsLoading(true);
  // Simulate login process
  setTimeout(() => {
    setIsLoading(false);
    toast.success(`Đăng nhập thành công với vai trò ${role}`);
    // Navigate to appropriate dashboard
    switch(role) {
      case 'admin':
        navigate('/admin');
        break;
      case 'teacher':
        navigate('/teacher');
        break;
      case 'student':
        navigate('/student');
        break;
    }
  }, 1500);
};
```

Đoạn mã trên được gọi khi người dùng nhấn nút "Đăng nhập". Ta thấy ứng dụng **không hề gọi API tới backend hay Keycloak**, mà chỉ chờ 1.5 giây rồi coi như đăng nhập thành công (hiện thông báo và điều hướng người dùng đến trang dashboard tương ứng) [2]. Việc này được dùng để **mô phỏng** quá trình đăng

nhập (có lẽ phục vụ thử nghiệm giao diện), nhưng không thực hiện xác thực thật. Toàn bộ thông tin email/mật khẩu người dùng nhập vào form không được gửi đi đâu cả.

Rõ ràng, **frontend hiện chưa tích hợp với Keycloak**. Trong một kịch bản hoàn thiện, đoạn code trên cần được thay thế bằng logic SSO thực: - Thay vì gọi `setTimeout` giả lập, ứng dụng có thể chuyển hướng trình duyệt tới đường dẫn đăng nhập Keycloak trên backend. Ví dụ: khi bấm nút, nếu vai trò là sinh viên, ta có thể vẫn gửi role đó để biết dashboard nào mở sau, nhưng quan trọng hơn là mở trang `/api/auth/login`. Khi đó, trang đăng nhập Keycloak thực sự xuất hiện. Người dùng đăng nhập xong, Keycloak trả về `/api/auth/callback` và backend có thể redirect người dùng về trang frontend (có thể là trang dashboard). - Một cách khác là dùng thư viện Keycloak JS: import Keycloak từ `keycloak-js`, khởi tạo với cấu hình URL server Keycloak, realm, clientId của ứng dụng [13]. Sau đó gọi `keycloak.login()` để mở cửa sổ đăng nhập. Thư viện này cũng cung cấp các hàm để kiểm tra trạng thái đăng nhập, lấy token, đăng xuất, v.v. (như bài viết Viblo đã trình bày [1] [13]). Khi login thành công, ta có thể sử dụng `keycloak.token` (Access Token JWT) trực tiếp để gọi API backend (vì backend có thể chấp nhận và kiểm tra JWT do Keycloak cấp).

Tóm lại, đoạn code hiện tại cho thấy **sự thiếu vắng tích hợp SSO ở phía client**. Người mới học có thể hiểu rằng: dự án đã chuẩn bị một phần ở server cho SSO, nhưng client vẫn chưa nối dây vào. Việc cần làm là thay thế cơ chế giả lập này bằng cơ chế đăng nhập qua Keycloak thực sự để đảm bảo an toàn và đúng mục đích của dự án về IAM.

Đề xuất cải tiến

Dựa trên những phân tích và thiếu sót đã nêu, dưới đây là một số đề xuất nhằm cải thiện dự án cả về chức năng lẫn bảo mật, kèm tài nguyên tham khảo cho việc tích hợp Keycloak:

- **Bổ sung tài liệu (README và hướng dẫn cấu hình):** Dự án nên có một file README.md chi tiết mô tả mục đích, công nghệ sử dụng, cách cài đặt và chạy. Ví dụ: hướng dẫn cài Node.js, cách chạy `npm install` cho frontend và backend, cách khởi động Keycloak (có thể hướng dẫn dùng Docker hoặc sử dụng folder `keycloak-26.2.4`), thiết lập các biến môi trường (`KEYCLOAK_ISSUER`, `KEYCLOAK_CLIENT_ID`, `KEYCLOAK_CLIENT_SECRET`, v.v.). Ngoài ra, cần mô tả nhanh kiến trúc thư mục (frontend, backend, keycloak) để người đọc hiểu mối liên hệ. Tài liệu rõ ràng sẽ giúp cả người phát triển mới lẫn người dùng cuối có thể tiếp cận dự án một cách dễ dàng và đúng cách hơn.

- **Hoàn thiện quy trình đăng nhập SSO:** Như phân tích, frontend cần được kết nối với backend/Keycloak để thực hiện SSO thực sự. Cụ thể:

- Thay thế việc **giả lập đăng nhập** bằng cách sử dụng Keycloak: Có thể thêm một nút "Đăng nhập bằng Keycloak" hoặc trực tiếp chỉnh sửa hành vi nút đăng nhập hiện tại để nó gọi tới backend. Khi người dùng bấm đăng nhập, ta có thể redirect sang `http://localhost:3000/api/auth/login` (giả sử frontend đang chạy cũng trên cổng 3000 khác route, cần kiểm tra cấu hình CORS cho backend cho phép điều này). Sau khi người dùng đăng nhập trên Keycloak xong, backend có thể chuyển hướng người dùng trở lại một route trên frontend, kèm theo token dưới dạng tham số hoặc lưu trong cookie.

- **Nhận và lưu token trên frontend:** Nếu backend trả về JWT trong JSON, frontend cần lấy JWT đó và lưu (trong memory hoặc localStorage, nhưng tốt nhất là lưu vào **HTTP-only cookie** để tránh XSS). Nếu dùng Keycloak JS adapter, thư viện sẽ tự quản lý token trong ứng dụng.
- Mỗi khi gọi các API cần bảo vệ, frontend gửi kèm token (qua header Authorization Bearer nếu dùng JWT).
- Cài đặt trang chặn (Protected Route): trong React Router, tạo component để bao bọc các route `/admin`, `/teacher`, `/student` – bên trong kiểm tra nếu chưa đăng nhập (không có token/đã hết hạn) thì redirect về trang login. Điều này ngăn người dùng truy cập giao diện khi chưa đăng nhập.
- Triển khai chức năng **logout**: Xoá token và điều hướng người dùng về trang chủ hoặc trang login. Đồng thời gọi tới backend (ví dụ endpoint `/api/auth/logout`) để Keycloak đăng xuất phiên của user, đảm bảo phiên SSO bị huỷ.

Tham khảo: Bài viết trên Viblo "Cách đăng nhập SSO Keycloak với ReactJs và Typescript" có hướng dẫn tuần tự cách cài thư viện keycloak-js, khởi tạo đối tượng Keycloak và sử dụng để bảo vệ các route React [1] [13]. Đây là tài liệu tiếng Việt hữu ích cho lập trình viên frontend khi làm việc với Keycloak.

- **Bảo vệ API và phân quyền trên backend:** Hiện một số API backend chưa được bảo vệ bởi bất kỳ lớp an ninh nào (như `users.controller.ts` cho phép lấy thông tin người dùng tự do). Sau khi đã có cơ chế JWT, cần bổ sung **Guard** kiểm tra JWT cho các request:
- Có thể sử dụng **Passport JWT strategy**: tạo một `JwtStrategy` trong NestJS để validate JWT (giải mã token bằng secret, lấy thông tin user từ payload). Sau đó, dùng `@UseGuards(AuthGuard('jwt'))` ở các controller hoặc thiết lập global guard để tất cả các route (trừ những route public như login, health check) đều yêu cầu JWT. NestJS có tài liệu về kỹ thuật này rất chi tiết [18] [19].
- Hoặc sử dụng thư viện **nest-keycloak-connect**: đây là một package tích hợp sẵn Keycloak cho Nest, cho phép khai báo một lần và áp dụng guard tự động cho toàn bộ ứng dụng. Ví dụ, có thể đăng ký `KeycloakConnectModule` trong `AppModule` với thông tin server, realm, clientId, secret Keycloak [20], đồng thời cung cấp các `APP_GUARD` để bật **AuthGuard** (xác thực token), **ResourceGuard** và **RoleGuard** (phân quyền theo vai trò) ở phạm vi toàn cục [21]. Sau đó, có thể sử dụng các decorator như `@Roles('admin')` trên controller để hạn chế quyền. Cách này tiện lợi và giảm công sức tự kiểm tra token.
- Xác định **quy tắc phân quyền**: Dựa trên vai trò (role) trong Keycloak. Ví dụ: trang `/admin` chỉ cho phép user có role “admin”, tương tự cho teacher và student. Có thể thiết lập trong Keycloak: gán realm role hoặc client role cho người dùng. Khi user đăng nhập, token sẽ chứa roles của họ. Backend khi kiểm tra JWT có thể đọc claim roles để quyết định. Triển khai đơn giản: viết một guard custom kiểm tra `req.user.role` (nếu ta lưu role vào JWT khi tạo) hoặc tra DB để lấy role rồi so sánh với vai trò yêu cầu của endpoint.

- Đừng quên bảo vệ cả những API nhạy cảm khác (vd nếu có API quản lý điểm số, thông tin cá nhân...): đảm bảo chỉ người có thẩm quyền (đúng người, hoặc đúng vai trò) mới truy cập được.

- **Quản lý thông tin nhạy cảm và cấu hình an toàn:** Xử lý các thông tin như mật khẩu, secret cẩn trọng hơn:

- **Không commit thẳng mật khẩu**: Mật khẩu database của Keycloak và bất kỳ mật khẩu nào khác không nên để trong file cấu hình commit lên Git (như trường hợp `db-password=3147` [5]). Thay

vào đó, sử dụng biến môi trường. Có thể tạo file `.env.example` (không chứa giá trị nhạy cảm, chỉ liệt kê key) để người dùng biết cần thiết lập biến nào.

- **Bảo vệ client secret:** Tương tự, clientSecret của Keycloak (dùng cho client type Confidential) phải được lưu trong biến môi trường (ví dụ KEYCLOAK_CLIENT_SECRET) và nạp vào ConfigService. Đảm bảo file .env thật được liệt kê trong .gitignore để không đẩy lên repo.
- **Sử dụng HTTPS (trong môi trường production):** Keycloak làm việc với OpenID Connect – khuyến cáo triển khai trên kênh HTTPS để bảo mật token. Dù trong dev có thể dùng http, khi triển khai thực tế nên cấu hình `ssl` cho backend và chạy Keycloak qua https (hoặc ít nhất proxy qua https). Thuộc tính `https-required=true` trong Keycloak config có thể bật lên khi deploy.

- **CORS và cookie an toàn:** Nếu frontend và backend deploy trên domain khác nhau, cần thiết lập CORS cho phép domain frontend gọi API backend. Nếu dùng cookie để lưu token (session), phải đặt cookie `SameSite=None; Secure` khi chạy trên HTTPS.

- **Tách bạch và tự động hóa môi trường chạy:** Hiện dự án bao gồm cả mã frontend, backend và Keycloak server trong một repo. Cách này tiện cho phát triển tập trung, nhưng khi triển khai có thể tách riêng. Để thuận tiện, có thể viết một file **docker-compose.yml** để định nghĩa 3 service: frontend (ví dụ một image Nginx phục vụ build static React), backend (Node NestJS), và Keycloak (sử dụng image Keycloak chính thức) kèm một dịch vụ PostgreSQL cho Keycloak. Việc này giúp bất kỳ ai cũng có thể chạy toàn bộ hệ thống bằng một lệnh mà không cần cài đặt thủ công từng thành phần. Docker-compose cũng cho phép cấu hình sẵn realm, tài khoản admin Keycloak qua biến môi trường (KEYCLOAK_ADMIN, KEYCLOAK_ADMIN_PASSWORD). Tài liệu "Setting up Keycloak with NestJS" của CreoWis có ví dụ về việc dùng docker-compose cho Keycloak + Postgres và tích hợp NestJS từng bước [22] [23]. Nếu chưa quen Docker, ít nhất dự án nên cung cấp script hoặc hướng dẫn rõ ràng để chạy từng phần:

- Cách chạy Keycloak: ví dụ "vào thư mục keycloak-26.2.4, chạy `bin/kc.sh start-dev`, tài khoản admin mặc định và mật khẩu...".
- Cách chạy backend: "cd vào score-backend, `npm install`, `npm run start:dev`... (cần Node version bao nhiêu, v.v.)".
- Cách chạy frontend: "cd vào thư mục giao diện, `npm install`, `npm run dev` ...".

- Bước cấu hình: "Truy cập Keycloak tại http://localhost:8080, tạo realm tên X, tạo client, v.v." – phần này nếu có thể script hóa (qua import file JSON config realm hoặc sử dụng Keycloak Admin CLI) sẽ tốt hơn là làm tay.

- **Bổ sung kiểm thử tự động:** Nên viết thêm các **unit test** và **integration test** cho các chức năng chính:

- Test cho UsersService.upsertFromKeycloak: đảm bảo khi gửi vào thông tin user mới thì tạo mới, gửi thông tin user cũ thì cập nhật.
- Test cho AuthController (có thể sử dụng **supertest** hoặc ** Pactum ** để mô phỏng luồng OAuth – việc này hơi phức tạp vì cần một Keycloak chạy để test end-to-end; có thể mock strategy để test logic sau khi có req.user).
- Test bảo vệ endpoint: ví dụ gọi `/api/users/:sub` mà không có JWT phải nhận 401 Unauthorized, có JWT hợp lệ nhưng không đúng role thì 403 Forbidden (nếu làm phân quyền), v.v.

- Kiểm thử giao diện: có thể viết vài test với Jest + React Testing Library để đảm bảo nút "Đăng nhập" sẽ gọi đúng hành động (ví dụ chuyển hướng tới `/api/auth/login`). Nếu tích hợp thực tế phức tạp, có thể tách logic đó ra để test.

Việc thêm kiểm thử sẽ giúp dự án tránh được các lỗi khi tích hợp bảo mật (ví dụ lỡ mở quyền unintended, hoặc quên cấp token cho một luồng nào đó). Hơn nữa, trong bối cảnh đây có thể là một đồ án môn học về An toàn mạng, việc có kiểm thử cho các tính năng an toàn (như đăng nhập, phân quyền) sẽ là điểm cộng chứng tỏ nhóm hiểu và kiểm soát được hệ thống của mình.

- **Học tập và tham khảo thêm về Keycloak:** Để hỗ trợ nhóm phát triển tự tin hơn với việc tích hợp IAM, dưới đây là một số tài nguyên hữu ích:
- **Tài liệu chính thức Keycloak:** Trang chủ Keycloak (keycloak.org) có phần Guides và Documentation rất đầy đủ, hướng dẫn từ cơ bản đến nâng cao việc thiết lập realms, clients, vai trò, cũng như tích hợp với các ngôn ngữ khác nhau. Nên đọc các phần về **Securing Applications** (bảo mật ứng dụng) cho trường hợp ứng dụng React (Public client) và NestJS API (Bearer-only or Confidential client).
- **Bài viết hướng dẫn và ví dụ:**
  - Bài Viblo mình đã nhắc ở trên về tích hợp Keycloak cho React (tiếng Việt) [1] [13].
  - Bài **"Protecting your NestJS API with Keycloak"** trên ITNEXT (tiếng Anh) cung cấp một góc nhìn khác: thay vì dùng Passport, bài viết hướng dẫn gọi thẳng tới endpoint **UserInfo** của Keycloak để xác thực token [24] [25]. Cách này đơn giản và hiệu quả, có thể tham khảo để hiểu sâu hơn về cách Keycloak cung cấp thông tin người dùng.
  - Ví dụ dự án GitHub **"Secure NestJS REST API with Keycloak"** [26]: Đây là một repo mẫu thiết lập bảo vệ API NestJS bằng Keycloak (có thể sử dụng làm đối chiếu với dự án của mình xem còn thiếu gì).
  - Thư viện **nest-keycloak-connect** trên GitHub/NPM cũng có README rất chi tiết về cách dùng các guard và decorator cho Keycloak trong NestJS, kèm theo ví dụ code [20] [21].
- **Cộng đồng và thảo luận:** Nếu gặp vướng mắc, có thể tìm kiếm trên Stack Overflow (từ khóa "Keycloak NestJS", "Keycloak React") – nhiều câu trả lời hướng dẫn từng bước cấu hình. Ngoài ra, các blog Medium, trang như Dzone, v.v. cũng có loạt bài về Keycloak.
- **Thực hành nhiều lần:** IAM nói chung và Keycloak nói riêng có khá nhiều khái niệm (realm, client, role, token, logout, v.v.). Cách học tốt là tự dựng một môi trường Keycloak (dùng Docker cho nhanh), tạo vài ứng dụng mẫu (một ứng dụng frontend, một backend) và thử đăng nhập/đăng xuất, thử các chế độ client (public vs confidential vs bearer-only) để hiểu sự khác biệt. Ví dụ: tạo một realm "Demo", tạo một client cho frontend (Public, dùng Javascript Adapter) và một client cho backend (Bearer-only), tạo vài user và gán role rồi kiểm thử. Việc này sẽ cho trải nghiệm thực tế để áp dụng vào dự án chính.

Những đề xuất và tài nguyên trên hy vọng sẽ giúp nhóm phát triển **củng cố bảo mật IAM cho dự án**. Bằng cách hoàn thiện tích hợp Keycloak, dự án sẽ đạt được mục tiêu về An toàn mạng: người dùng đăng nhập một lần an toàn, các thành phần của hệ thống (frontend, backend) đều xác thực và ủy quyền đúng đắn, giảm thiểu các lỗ hổng do xác thực thủ công. Chúc các bạn triển khai thành công!

---

[1] [13] Cách đăng nhập SSO Keycloak với ReactJs và Typescript
https://viblo.asia/p/cach-dang-nhap-sso-keycloak-voi-reactjs-va-typescript-38X4E52j4N2

2 Index.tsx
https://github.com/nam091/An_Toan_Mang_May_Tinh_KMA/blob/acda4e10067197c46538f6fbc10f8c5e1a7224c4/src/pages/Index.tsx

3 app.e2e-spec.ts
https://github.com/nam091/An_Toan_Mang_May_Tinh_KMA/blob/acda4e10067197c46538f6fbc10f8c5e1a7224c4/score-backend/test/app.e2e-spec.ts

4 README.md
https://github.com/nam091/An_Toan_Mang_May_Tinh_KMA/blob/acda4e10067197c46538f6fbc10f8c5e1a7224c4/keycloak-26.2.4/README.md

5 keycloak.conf.new
https://github.com/nam091/An_Toan_Mang_May_Tinh_KMA/blob/acda4e10067197c46538f6fbc10f8c5e1a7224c4/keycloak-26.2.4/conf/keycloak.conf.new

6 10 11 auth.controller.ts
https://github.com/nam091/An_Toan_Mang_May_Tinh_KMA/blob/acda4e10067197c46538f6fbc10f8c5e1a7224c4/score-backend/src/auth/auth.controller.ts

7 users.controller.ts
https://github.com/nam091/An_Toan_Mang_May_Tinh_KMA/blob/acda4e10067197c46538f6fbc10f8c5e1a7224c4/score-backend/src/users/users.controller.ts

8 9 14 15 keycloak.strategy.ts
https://github.com/nam091/An_Toan_Mang_May_Tinh_KMA/blob/acda4e10067197c46538f6fbc10f8c5e1a7224c4/score-backend/src/auth/keycloak.strategy.ts

12 package.json
https://github.com/nam091/An_Toan_Mang_May_Tinh_KMA/blob/acda4e10067197c46538f6fbc10f8c5e1a7224c4/score-backend/package.json

16 users.service.ts
https://github.com/nam091/An_Toan_Mang_May_Tinh_KMA/blob/acda4e10067197c46538f6fbc10f8c5e1a7224c4/score-backend/src/users/users.service.ts

17 user.entity.ts
https://github.com/nam091/An_Toan_Mang_May_Tinh_KMA/blob/acda4e10067197c46538f6fbc10f8c5e1a7224c4/score-backend/src/users/user.entity.ts

18 19 24 25 node.js - How to use keycloak with NestJS properly - Stack Overflow
https://stackoverflow.com/questions/56790476/how-to-use-keycloak-with-nestjs-properly

20 21 How to set up Keycloak with NestJS?
https://www.creowis.com/blog/how-to-set-up-keycloak-with-nestjs

22 23 Secure Your Nest.js App Using KeyCloak: Developer's Guide
https://www.meetri.in/blogs/key-cloak-with-nestjs-iam.html

26 Example Secure NestJS REST API with Keycloak. - GitHub
https://github.com/Vipcube/nestjs-keycloak-example