**Báo cáo Dự án Web API Quản lý Người dùng**

*Link dự án:* [*https://github.com/nguyenkhanhbh01989/UserManagementApi.git*](https://github.com/nguyenkhanhbh01989/UserManagementApi.git)

Nội dung

[I. Giới thiệu dự án 2](#_Toc200472959)

[A. Tổng quan dự án 2](#_Toc200472960)

[B. Nghiệp vụ cốt lõi 2](#_Toc200472961)

[II. Kiến trúc và Công nghệ 3](#_Toc200472962)

[A. Mô hình kiến trúc 3](#_Toc200472963)

[B. Công nghệ sử dụng 4](#_Toc200472964)

[III. Các Khái niệm và Lý thuyết Nền tảng 5](#_Toc200472965)

[A. Web API (RESTful API) 5](#_Toc200472966)

[B. ASP.NET Core 6](#_Toc200472967)

[C. Middleware 6](#_Toc200472968)

[D. Dependency Injection (DI) 7](#_Toc200472969)

[E. Authentication (Xác thực) 8](#_Toc200472970)

[F. JWT (JSON Web Token) Authentication 8](#_Toc200472971)

[G. Authorization (Ủy quyền) 10](#_Toc200472972)

[H. ORM (Object-Relational Mapping) và Entity Framework Core 11](#_Toc200472973)

[I. SQL (Structured Query Language) 11](#_Toc200472974)

[J. Các hệ quản trị cơ sở dữ liệu khác 12](#_Toc200472975)

[K. Băm mật khẩu (Password Hashing - BCrypt.Net) 13](#_Toc200472976)

[L. DTO (Data Transfer Object) 13](#_Toc200472977)

[IV. Thiết kế Cơ sở dữ liệu 14](#_Toc200472978)

[A. Sơ đồ bảng (User Table) 14](#_Toc200472979)

[B. Mối quan hệ 15](#_Toc200472980)

[V. Luồng hoạt động chính của API 15](#_Toc200472981)

[A. Sơ đồ luồng hoạt động tổng thể 16](#_Toc200472982)

[B. Giải thích các luồng hoạt động chi tiết 17](#_Toc200472983)

[VI. Kết luận và Hướng phát triển 20](#_Toc200472984)

[A. Tóm tắt những gì đã đạt được 20](#_Toc200472985)

[B. Hướng phát triển trong tương lai 20](#_Toc200472986)

# Giới thiệu dự án

## Tổng quan dự án

Dự án **Web API Quản lý Người dùng** là một dịch vụ backend được xây dựng trên nền tảng **ASP.NET Core**, sử dụng ngôn ngữ lập trình **C#** và **SQL Server** làm hệ quản trị cơ sở dữ liệu. Mục tiêu chính của dự án là cung cấp một bộ các **API RESTful** mạnh mẽ và bảo mật để quản lý thông tin người dùng, bao gồm các chức năng cốt lõi như đăng ký, đăng nhập, và quản lý hồ sơ cá nhân.

API này được thiết kế để hoạt động như một dịch vụ độc lập, có khả năng tích hợp linh hoạt với nhiều loại ứng dụng frontend khác nhau như Single Page Applications (SPA) được xây dựng bằng React, Angular, Vue.js, ứng dụng di động, hoặc các ứng dụng desktop. Bằng cách tách biệt phần backend và frontend, dự án tuân thủ nguyên tắc phát triển phần mềm hiện đại, tăng cường khả năng mở rộng, dễ dàng bảo trì và quản lý.

## Nghiệp vụ cốt lõi

Dự án tập trung vào các nghiệp vụ chính liên quan đến quản lý người dùng:

1. **Đăng ký người dùng mới**: Cho phép người dùng tạo tài khoản mới bằng cách cung cấp tên đăng nhập, mật khẩu và email. Mật khẩu được băm (hash) an toàn trước khi lưu trữ để đảm bảo bảo mật.
2. **Đăng nhập người dùng**: Xác thực thông tin đăng nhập của người dùng. Khi đăng nhập thành công, API sẽ cấp phát **JSON Web Token (JWT)** cho các ứng dụng client yêu cầu xác thực không trạng thái (stateless) và đồng thời thiết lập **cookie session** cho các ứng dụng trình duyệt cần duy trì trạng thái phiên.
3. **Quản lý thông tin cá nhân (/api/Users/me)**:
   * **Xem thông tin cá nhân**: Người dùng có thể truy cập thông tin chi tiết về tài khoản của chính mình (ID, tên đăng nhập, email, thời gian tạo) thông qua cookie session, mà không cần sử dụng JWT.
   * **Cập nhật thông tin cá nhân**: Cho phép người dùng thay đổi tên đăng nhập hoặc email của họ.
   * **Thay đổi mật khẩu**: Cung cấp cơ chế an toàn để người dùng cập nhật mật khẩu, yêu cầu xác thực mật khẩu cũ trước khi đặt mật khẩu mới. Sau khi thay đổi mật khẩu thành công, phiên đăng nhập hiện tại sẽ bị vô hiệu hóa để tăng cường bảo mật.
   * **Xóa tài khoản**: Cho phép người dùng tự xóa tài khoản của chính mình khỏi hệ thống. Khi tài khoản bị xóa, phiên đăng nhập hiện tại cũng sẽ bị chấm dứt.
4. **Xem thông tin người dùng công khai (/api/Users và /api/Users/{id})**:
   * **Xem danh sách tất cả người dùng**: Cung cấp API cho phép xem danh sách tất cả người dùng trong hệ thống (chỉ bao gồm ID, tên đăng nhập và email). Endpoint này yêu cầu xác thực bằng JWT.
   * **Xem thông tin người dùng theo ID**: Cho phép truy vấn thông tin cơ bản của một người dùng cụ thể dựa trên ID của họ. Endpoint này cũng yêu cầu xác thực bằng JWT.
   * Cả hai endpoint này đều là **chỉ đọc (read-only)**, không cho phép thêm, sửa, hoặc xóa dữ liệu người dùng để duy trì tính toàn vẹn và bảo mật của dữ liệu.

# Kiến trúc và Công nghệ

## Mô hình kiến trúc

Dự án Web API Quản lý Người dùng được xây dựng dựa trên mô hình kiến trúc **RESTful API (Representational State Transfer Application Programming Interface)**. Đây là một phong cách kiến trúc phổ biến để xây dựng các dịch vụ web, nhấn mạnh vào việc sử dụng các tài nguyên (resources) và các phương thức HTTP tiêu chuẩn để thao tác với chúng. Các đặc điểm chính của kiến trúc này trong dự án bao gồm:

* **Có tài nguyên (Resources)**: Mỗi đối tượng (ví dụ: User) được coi là một tài nguyên và được định danh bằng một URI (Uniform Resource Identifier) duy nhất (ví dụ: /api/Users, /api/Users/me, /api/Users/{id}).
* **Sử dụng phương thức HTTP tiêu chuẩn**: Các thao tác CRUD (Create, Read, Update, Delete) được ánh xạ tới các phương thức HTTP chuẩn:
  + POST: Để tạo tài nguyên mới (ví dụ: POST /api/Auth/register để tạo người dùng).
  + GET: Để đọc/truy xuất tài nguyên (ví dụ: GET /api/Users/me, GET /api/Users/{id}, GET /api/Users).
  + PUT: Để cập nhật tài nguyên hiện có (ví dụ: PUT /api/Users/me để cập nhật thông tin cá nhân).
  + DELETE: Để xóa tài nguyên (ví dụ: DELETE /api/Users/me để xóa tài khoản).
* **Không trạng thái (Stateless)**: Mỗi yêu cầu từ client đến server phải chứa tất cả thông tin cần thiết để server hiểu và xử lý yêu cầu đó. Server không lưu trữ bất kỳ thông tin nào về trạng thái phiên của client giữa các yêu cầu. Điều này được thực hiện chủ yếu thông qua **JWT** cho phần lớn API, giúp API dễ dàng mở rộng theo chiều ngang. Tuy nhiên, dự án cũng có sử dụng **Cookie/Session** cho các endpoint cá nhân (/api/Users/me) nhằm tối ưu cho các trình duyệt web truyền thống và các thao tác liên quan đến trạng thái phiên đăng nhập trực tiếp.
* **Giao diện thống nhất**: API tuân thủ một bộ quy tắc và hạn chế chung, giúp client dễ dàng hiểu và tương tác với các tài nguyên.

## Công nghệ sử dụng

Dự án này được xây dựng dựa trên các công nghệ và thư viện chính sau:

* **ASP.NET Core**: Là framework mã nguồn mở, đa nền tảng, hiệu suất cao của Microsoft để xây dựng các ứng dụng web và API. Nó cung cấp cấu trúc mạnh mẽ, tích hợp sẵn các tính năng như Dependency Injection, Middleware, và hỗ trợ xác thực/ủy quyền.
* **C#**: Ngôn ngữ lập trình chính được sử dụng. C# là một ngôn ngữ hiện đại, hướng đối tượng, an toàn kiểu, được hỗ trợ mạnh mẽ bởi Microsoft.
* **Entity Framework Core (EF Core)**: Là một ORM (Object-Relational Mapper) cho .NET. EF Core cho phép các nhà phát triển làm việc với cơ sở dữ liệu bằng cách sử dụng các đối tượng .NET (classes) thay vì phải viết các câu lệnh SQL trực tiếp. Nó giúp đơn giản hóa việc tương tác với database, từ việc tạo bảng (migrations) đến truy vấn và cập nhật dữ liệu.
* **SQL Server**: Hệ quản trị cơ sở dữ liệu quan hệ (RDBMS) được sử dụng để lưu trữ dữ liệu người dùng. SQL Server là một lựa chọn phổ biến và mạnh mẽ cho các ứng dụng doanh nghiệp.
* **Swagger/OpenAPI**: Một bộ công cụ mã nguồn mở giúp thiết kế, xây dựng, tài liệu hóa và sử dụng các dịch vụ RESTful API. Trong dự án này, Swagger UI được tích hợp để cung cấp giao diện người dùng thân thiện, cho phép kiểm tra và tương tác với các endpoint API một cách dễ dàng ngay trong trình duyệt.
* **BCrypt.Net**: Thư viện mã nguồn mở được sử dụng để băm (hash) mật khẩu. BCrypt là một thuật toán băm mật khẩu mạnh mẽ, được thiết kế để chống lại các cuộc tấn công vét cạn (brute-force) và tấn công từ điển, bằng cách thêm muối (salt) và thực hiện nhiều vòng lặp băm.
* **CORS (Cross-Origin Resource Sharing)**: Một cơ chế bảo mật trình duyệt cho phép tài nguyên web trên một domain được yêu cầu từ một domain khác. Trong dự án, CORS được cấu hình để cho phép các ứng dụng frontend (chạy trên localhost:80 và localhost:443) có thể gọi các API, đảm bảo rằng chính sách bảo mật Same-Origin Policy không ngăn chặn các yêu cầu hợp lệ.

# Các Khái niệm và Lý thuyết Nền tảng

Để xây dựng một Web API hiện đại và mạnh mẽ, dự án này đã áp dụng nhiều khái niệm và lý thuyết quan trọng trong phát triển phần mềm. Dưới đây là giải thích chi tiết về từng khái niệm:

## Web API (RESTful API)

**Web API** là một giao diện lập trình ứng dụng (API) trên web, cho phép các ứng dụng khác (client) tương tác với một hệ thống hoặc dịch vụ thông qua giao thức HTTP. Dự án này xây dựng một **RESTful API**, nghĩa là nó tuân thủ các nguyên tắc thiết kế của kiến trúc **REST (Representational State Transfer)**.

**Các nguyên tắc cơ bản của RESTful API bao gồm:**

* **Tài nguyên (Resources)**: Mọi thứ trong API đều được coi là một tài nguyên (ví dụ: người dùng, sản phẩm, bài viết). Mỗi tài nguyên có một định danh duy nhất (URI - Uniform Resource Identifier), ví dụ: /api/Users đại diện cho tập hợp người dùng, /api/Users/me đại diện cho người dùng hiện tại, và /api/Users/{id} đại diện cho một người dùng cụ thể.
* **Phương thức HTTP chuẩn**: Các thao tác trên tài nguyên được thực hiện thông qua các phương thức HTTP tiêu chuẩn (verbs):
  + **GET**: Truy xuất dữ liệu.
  + **POST**: Tạo mới tài nguyên.
  + **PUT**: Cập nhật toàn bộ tài nguyên.
  + **DELETE**: Xóa tài nguyên.
  + **PATCH**: Cập nhật một phần tài nguyên (chưa được sử dụng trong dự án này nhưng là một phương thức RESTful phổ biến).
* **Không trạng thái (Stateless)**: Mỗi yêu cầu từ client đến server phải chứa đầy đủ thông tin để server xử lý nó. Server không lưu trữ bất kỳ trạng thái phiên nào giữa các yêu cầu. Điều này giúp API dễ dàng mở rộng và chịu tải cao.
* **Hệ thống phân lớp (Layered System)**: Các thành phần của API có thể được tổ chức thành các lớp khác nhau (ví dụ: lớp trình bày, lớp logic nghiệp vụ, lớp truy cập dữ liệu), giúp modular hóa và quản lý phức tạp.
* **Giao diện thống nhất (Uniform Interface)**: API tuân thủ một bộ quy tắc và hạn chế chung, giúp client dễ dàng hiểu và tương tác với các tài nguyên một cách nhất quán.

## ASP.NET Core

**ASP.NET Core** là một framework mã nguồn mở, đa nền tảng, hiệu suất cao để xây dựng các ứng dụng web và API trên nền tảng .NET. Trong dự án này, ASP.NET Core đóng vai trò là xương sống, cung cấp các công cụ và cấu trúc cần thiết để:

* **Xây dựng các Controller và Action**: Định nghĩa các endpoint API và logic xử lý yêu cầu HTTP.
* **Quản lý yêu cầu HTTP (HTTP Request Pipeline)**: Sử dụng hệ thống Middleware để xử lý các yêu cầu theo từng bước.
* **Tích hợp Dependency Injection (DI)**: Dễ dàng quản lý và cung cấp các dịch vụ cho các thành phần khác.
* **Hỗ trợ Xác thực và Ủy quyền**: Cung cấp cơ chế mạnh mẽ để bảo vệ các endpoint API.
* **Hỗ trợ CORS**: Cho phép API giao tiếp an toàn với các ứng dụng frontend từ các nguồn gốc khác.

## Middleware

**Middleware** trong ASP.NET Core là các thành phần phần mềm được lắp ráp thành một chuỗi (pipeline) xử lý yêu cầu HTTP. Mỗi Middleware có khả năng:

* Xử lý yêu cầu HTTP đến.
* Thực hiện các thao tác (như ghi log, xác thực, ủy quyền).
* Truyền yêu cầu cho Middleware tiếp theo trong chuỗi.
* Hoặc "short-circuit" (ngắt) chuỗi và trả về phản hồi trực tiếp.

Các Middleware quan trọng đã được sử dụng trong dự án:

* **app.UseHttpsRedirection()**: Chuyển hướng các yêu cầu HTTP không an toàn sang HTTPS để đảm bảo bảo mật đường truyền.
* **app.UseCors()**: Áp dụng chính sách CORS đã cấu hình, cho phép các yêu cầu từ các nguồn gốc được phép vượt qua chính sách Same-Origin của trình duyệt. Nó phải được đặt trước UseAuthentication() và UseAuthorization() để header CORS được xử lý trước.
* **app.UseAuthentication()**: Kích hoạt dịch vụ xác thực. Middleware này sẽ kiểm tra các thông tin xác thực (như JWT trong header Authorization hoặc cookie session) trong yêu cầu và tạo ra một ClaimsPrincipal (đối tượng đại diện cho người dùng đã xác thực) để các Middleware tiếp theo (đặc biệt là UseAuthorization()) có thể sử dụng.
* **app.UseAuthorization()**: Kích hoạt dịch vụ ủy quyền. Middleware này sẽ kiểm tra các quyền (permissions) của người dùng dựa trên ClaimsPrincipal được tạo bởi UseAuthentication() và các [Authorize] attribute trên các Controller/Action, quyết định xem người dùng có được phép truy cập tài nguyên hay không.

## Dependency Injection (DI)

**Dependency Injection (DI)**, hay Tiêm phụ thuộc, là một mẫu thiết kế phần mềm giúp giảm sự phụ thuộc cứng giữa các thành phần trong ứng dụng, từ đó tăng tính mô-đun, khả năng kiểm thử và bảo trì. Thay vì một đối tượng tự tạo ra các đối tượng mà nó phụ thuộc, các đối tượng phụ thuộc sẽ được "tiêm" vào nó từ bên ngoài.

**Trong ASP.NET Core, DI được tích hợp sẵn và đóng vai trò trung tâm:**

* **Lợi ích**:
  + **Giảm khớp nối (Loose Coupling)**: Các lớp không cần biết cách tạo ra các phụ thuộc của chúng.
  + **Tăng khả năng kiểm thử (Testability)**: Dễ dàng thay thế các phụ thuộc bằng các đối tượng giả lập (mocks/stubs) trong các bài kiểm thử đơn vị.
  + **Dễ dàng bảo trì và mở rộng**: Dễ dàng thay đổi hoặc thêm các triển khai khác nhau cho cùng một interface mà không ảnh hưởng đến các lớp sử dụng.
* **Cách áp dụng trong dự án**:
  + Đăng ký ApplicationDbContext và IConfiguration vào container DI trong Program.cs bằng builder.Services.AddDbContext<ApplicationDbContext>(...) và builder.Configuration được truy cập trực tiếp.
  + Các Controller như AuthController và UsersController yêu cầu ApplicationDbContext và IConfiguration trong constructor của chúng. Hệ thống DI của ASP.NET Core sẽ tự động "tiêm" các thể hiện (instance) của các dịch vụ này khi một Controller được tạo.

## Authentication (Xác thực)

**Xác thực** là quá trình xác minh danh tính của người dùng (Ai là bạn?). Dự án này sử dụng hai cơ chế xác thực chính để phù hợp với các kịch bản client khác nhau:

**Cookie-based Authentication (Session)**

* **Cách hoạt động**: Khi người dùng đăng nhập thành công, server tạo ra một **cookie session** chứa thông tin định danh (thường là một ID phiên hoặc thông tin người dùng được mã hóa) và gửi nó về trình duyệt của client. Trình duyệt sẽ tự động gửi cookie này trong các yêu cầu tiếp theo đến cùng một domain. Server sau đó sử dụng cookie để nhận diện và xác minh người dùng cho mỗi yêu cầu.
* **Ưu điểm**:
  + Dễ dàng tích hợp với các ứng dụng web truyền thống và các form HTML.
  + Quản lý phiên tập trung trên server (dễ dàng vô hiệu hóa phiên).
* **Nhược điểm**:
  + Yêu cầu server phải duy trì trạng thái phiên, có thể gây khó khăn cho việc mở rộng theo chiều ngang (scale horizontally) nếu không có giải pháp quản lý trạng thái phiên phân tán.
  + Dễ bị tấn công CSRF (Cross-Site Request Forgery) nếu không có biện pháp bảo vệ thích hợp (đã cấu hình SameSiteMode.Strict để giảm thiểu).
  + Không phù hợp trực tiếp với các ứng dụng di động hoặc SPA vì chúng thường không quản lý cookie session một cách tự nhiên.
* **Khi nào sử dụng trong dự án**: Được sử dụng cho các endpoint quản lý thông tin cá nhân của người dùng (/api/Users/me, PUT /api/Users/me, POST /api/Users/me/change-password, DELETE /api/Users/me) vì chúng thường được truy cập từ trình duyệt web, nơi cookie session là một cơ chế xác thực tự nhiên và thuận tiện.

## JWT (JSON Web Token) Authentication

* **JWT** là một tiêu chuẩn mở (RFC 7519) để tạo các token truy cập. Token được thiết kế để nhỏ gọn, an toàn URL và dùng để đại diện cho các claim (khẳng định) giữa hai bên.
* **Cấu trúc JWT**: Một JWT gồm ba phần, được phân tách bằng dấu chấm (.):
  + **Header**: Chứa kiểu token (JWT) và thuật toán mã hóa được sử dụng (ví dụ: HS256).
  + **Payload**: Chứa các "claim" (các thông tin về người dùng hoặc các quyền), ví dụ: sub (chủ thể - ID người dùng), name (tên người dùng), exp (thời gian hết hạn), iss (nhà phát hành), aud (đối tượng).
  + **Signature**: Chữ ký điện tử để xác minh rằng token không bị giả mạo hoặc thay đổi. Nó được tạo bằng cách mã hóa Header, Payload và một khóa bí mật (secret key) của server bằng thuật toán đã chọn.
* **Cách hoạt động**:
  + Người dùng đăng nhập với tên người dùng và mật khẩu.
  + Server xác thực và nếu thành công, sẽ tạo một JWT chứa các thông tin (claims) về người dùng và ký bằng khóa bí mật.
  + Server gửi JWT này về client.
  + Client lưu trữ JWT (ví dụ: trong localStorage hoặc sessionStorage của trình duyệt) và gửi nó trong header Authorization (dưới dạng Bearer Token) của mỗi yêu cầu tiếp theo đến server.
  + Server nhận yêu cầu, trích xuất JWT, giải mã bằng khóa bí mật. Nếu chữ ký hợp lệ và token chưa hết hạn, server sẽ tin tưởng các claims trong token và xử lý yêu cầu.
* **Ưu điểm**:
  + **Không trạng thái (Stateless)**: Server không cần lưu trữ bất kỳ trạng thái phiên nào. Điều này giúp API dễ dàng mở rộng theo chiều ngang (thêm nhiều server instance mà không cần đồng bộ hóa trạng thái phiên).
  + **Linh hoạt**: Có thể được sử dụng bởi các ứng dụng di động, SPA, hoặc các dịch vụ khác mà không cần phụ thuộc vào cookie.
  + **Độc lập**: Token có thể được gửi qua bất kỳ kênh nào (HTTP header, URL parameter).
* **Nhược điểm**:
  + **Không thể dễ dàng vô hiệu hóa token sớm**: Một khi token đã được cấp phát, nó vẫn hợp lệ cho đến khi hết hạn (trừ khi có cơ chế danh sách đen token).
  + **Kích thước token**: Token có thể lớn hơn cookie session nếu chứa nhiều claims.
  + **Bảo mật lưu trữ**: Client phải lưu trữ token một cách an toàn (tránh XSS).
* **Tại sao lại dùng JWT thay vì phương thức khác (như Session)**:
  + Đối với các **API hiện đại và microservices**, JWT là lựa chọn ưu việt hơn Cookie Session. Việc không trạng thái giúp API có thể mở rộng quy mô một cách dễ dàng, vì bất kỳ server nào cũng có thể xác thực token mà không cần truy cập chung vào một kho lưu trữ trạng thái phiên tập trung.
  + Nó phù hợp với các **ứng dụng di động và Single Page Applications (SPA)** nơi việc quản lý cookie session phức tạp hơn và việc gửi token qua header Authorization là tiêu chuẩn.
* **Khi nào sử dụng trong dự án**: Được sử dụng cho các endpoint cần xác thực không trạng thái hoặc được truy cập từ các ứng dụng client đa dạng (GET /api/Users/{id}, GET /api/Users).

**Sự kết hợp giữa Cookie và JWT trong dự án**

Dự án này đã áp dụng một cách tiếp cận **lai (hybrid)**, kết hợp cả Cookie-based Authentication và JWT Authentication. Điều này được thực hiện để tối ưu hóa cho cả hai kịch bản sử dụng phổ biến:

* **Cookie Authentication** cho các tác vụ quản lý thông tin cá nhân (/api/Users/me, PUT/DELETE/POST trên /me) vì chúng thường được thao tác thông qua trình duyệt web, nơi cookie tự động được quản lý bởi trình duyệt và tiện lợi cho người dùng.
* **JWT Authentication** cho các API mang tính "công khai" hơn (nhưng vẫn cần xác thực) như lấy danh sách người dùng hoặc thông tin người dùng cụ thể. Điều này giúp API có thể được sử dụng bởi các ứng dụng không phải trình duyệt (ví dụ: mobile app) mà không cần quan tâm đến cơ chế cookie.

Bằng cách này, có thể tận dụng ưu điểm của cả hai phương pháp xác thực, cung cấp sự linh hoạt và bảo mật phù hợp cho từng loại endpoint.

## Authorization (Ủy quyền)

**Ủy quyền** là quá trình xác định xem người dùng đã xác thực có quyền truy cập một tài nguyên cụ thể hoặc thực hiện một hành động cụ thể hay không (Bạn có quyền làm điều đó không?).

* **Cách hoạt động với [Authorize] attribute**: Trong ASP.NET Core, ủy quyền được thực hiện thông qua [Authorize] attribute. Khi một Controller hoặc action được đánh dấu bằng [Authorize], hệ thống sẽ kiểm tra xem yêu cầu có kèm theo thông tin xác thực hợp lệ hay không.
  + Đã chỉ định rõ lược đồ xác thực nào sẽ được sử dụng cho từng endpoint bằng cách truyền tham số AuthenticationSchemes vào [Authorize]. Ví dụ:
    - [Authorize(AuthenticationSchemes = CookieAuthenticationDefaults.AuthenticationScheme)] cho các endpoint yêu cầu cookie.
    - [Authorize(AuthenticationSchemes = JwtBearerDefaults.AuthenticationScheme)] cho các endpoint yêu cầu JWT.

## ORM (Object-Relational Mapping) và Entity Framework Core

* **ORM (Object-Relational Mapping)** là một kỹ thuật lập trình cho phép các nhà phát triển làm việc với cơ sở dữ liệu quan hệ (relational database) bằng cách sử dụng các đối tượng của ngôn ngữ lập trình hướng đối tượng (ví dụ: C# classes) thay vì phải viết các câu lệnh SQL trực tiếp.
* **Entity Framework Core (EF Core)** là một ORM mã nguồn mở của Microsoft dành cho .NET.
* **Vai trò của EF Core trong dự án**:
  + **Định nghĩa mô hình dữ liệu (Data Model)**: định nghĩa lớp User làm mô hình cho bảng Users trong cơ sở dữ liệu.
  + **Tạo và quản lý cơ sở dữ liệu (Migrations)**: EF Core cho phép tạo và cập nhật cấu trúc cơ sở dữ liệu dựa trên mô hình dữ liệu thông qua các lệnh Migration (ví dụ: Add-Migration, Update-Database).
  + **Truy vấn dữ liệu**: sử dụng LINQ (Language Integrated Query) với \_context.Users.Where(...), \_context.Users.FirstOrDefaultAsync(...), \_context.Users.FindAsync(...), \_context.Users.Select(...) để truy vấn dữ liệu từ database một cách dễ dàng và an toàn kiểu. EF Core sẽ tự động dịch các câu lệnh LINQ này thành SQL phù hợp.
  + **Cập nhật và xóa dữ liệu**: Sử dụng các phương thức như \_context.Users.Add(), \_context.Users.Remove(), và \_context.SaveChangesAsync() để thực hiện các thao tác ghi dữ liệu vào database.

## SQL (Structured Query Language)

* **SQL** là ngôn ngữ tiêu chuẩn để quản lý và thao tác dữ liệu trong các hệ quản trị cơ sở dữ liệu quan hệ (RDBMS) như SQL Server, MySQL, PostgreSQL, Oracle.
* **Vai trò trong dự án**: Mặc dù sử dụng EF Core (một ORM) để tương tác với database, SQL vẫn là ngôn ngữ cơ bản mà EF Core tạo ra và thực thi ở phía dưới. Hiểu biết về SQL giúp :
  + Gỡ lỗi các vấn đề liên quan đến database.
  + Tối ưu hóa hiệu suất truy vấn.
  + Hiểu rõ cách dữ liệu được lưu trữ và truy xuất.
  + Quản lý cơ sở dữ liệu trực tiếp khi cần thiết.

## Các hệ quản trị cơ sở dữ liệu khác

* **Có thể dùng DB khác** ASP.NET Core và Entity Framework Core được thiết kế để linh hoạt và hỗ trợ nhiều loại cơ sở dữ liệu khác nhau thông qua các nhà cung cấp (provider) tương ứng.
* **Ví dụ**:
  + **MySQL/PostgreSQL**: Nếu muốn sử dụng MySQL hoặc PostgreSQL, ta chỉ cần cài đặt gói NuGet của nhà cung cấp EF Core tương ứng (ví dụ: Microsoft.EntityFrameworkCore.MySql hoặc Npgsql.EntityFrameworkCore.PostgreSQL) và thay đổi cấu hình AddDbContext trong Program.cs để sử dụng chuỗi kết nối và provider đó.
  + **MongoDB (NoSQL)**: MongoDB là một cơ sở dữ liệu NoSQL dạng tài liệu (document database). Để sử dụng MongoDB, ta sẽ không dùng Entity Framework Core (vì EF Core được thiết kế cho cơ sở dữ liệu quan hệ). Thay vào đó, mà sẽ sử dụng driver chính thức của MongoDB cho .NET (ví dụ: MongoDB.Driver) và tương tác trực tiếp với các collection (bộ sưu tập tài liệu) của MongoDB.
* **Ưu nhược điểm so với SQL Server trong ngữ cảnh này**:
  + **SQL Server (Quan hệ)**:
    - **Ưu điểm**: Đảm bảo tính toàn vẹn dữ liệu (ACID properties), phù hợp với dữ liệu có cấu trúc chặt chẽ và mối quan hệ phức tạp. Tốt cho các giao dịch phức tạp.
    - **Nhược điểm**: Khó mở rộng theo chiều ngang hơn so với NoSQL, cấu trúc dữ liệu ít linh hoạt hơn.
  + **MongoDB (NoSQL - Tài liệu)**:
    - **Ưu điểm**: Linh hoạt về schema (không cần định nghĩa schema trước), dễ dàng mở rộng theo chiều ngang, tốt cho dữ liệu bán cấu trúc hoặc phi cấu trúc, hiệu suất cao cho các thao tác đọc/ghi lớn.
    - **Nhược điểm**: Khó khăn hơn trong việc đảm bảo tính toàn vẹn dữ liệu và các giao dịch phức tạp (so với SQL), không phù hợp bằng cho dữ liệu có mối quan hệ phức tạp cần join nhiều bảng.
  + **Trong dự án này (Quản lý Người dùng)**: Dữ liệu người dùng có cấu trúc rõ ràng (ID, username, password hash, email, created at) và mối quan hệ đơn giản, nên SQL Server là một lựa chọn phù hợp và đáng tin cậy. Nếu dự án phát triển thành một hệ thống lớn với nhiều loại dữ liệu phi cấu trúc hoặc cần khả năng mở rộng cực kỳ cao, việc cân nhắc NoSQL như MongoDB có thể hợp lý.

## Băm mật khẩu (Password Hashing - BCrypt.Net)

* **Tầm quan trọng**: Việc **không bao giờ lưu trữ mật khẩu dưới dạng văn bản thuần (plaintext)** là một nguyên tắc bảo mật tối quan trọng. Nếu database bị lộ, mật khẩu plaintext sẽ bị rò rỉ ngay lập tức, gây ra rủi ro nghiêm trọng cho người dùng (vì họ có thể dùng cùng mật khẩu ở nhiều dịch vụ khác).
* **BCrypt.Net**: Thay vào đó, sử dụng thuật toán **băm mật khẩu (password hashing)**. Thuật toán này tạo ra một "hash" (một chuỗi ký tự cố định chiều dài) từ mật khẩu gốc. Hash này là một chiều, nghĩa là rất khó (gần như không thể) để đảo ngược hash trở lại mật khẩu gốc.
* **Đặc điểm của BCrypt**:
  + **Thêm muối (Salting)**: BCrypt tự động thêm một chuỗi ngẫu nhiên (salt) vào mật khẩu trước khi băm. Điều này đảm bảo rằng ngay cả hai người dùng có cùng mật khẩu, hash của họ vẫn khác nhau, ngăn chặn các cuộc tấn công bảng cầu vồng (rainbow table attacks).
  + **Số vòng lặp (Cost Factor)**: BCrypt cho phép cấu hình số vòng lặp băm. Càng nhiều vòng lặp, quá trình băm càng chậm, nhưng càng an toàn hơn đối với các cuộc tấn công vét cạn bằng phần cứng chuyên dụng.
* **Cách sử dụng**: Khi người dùng đăng ký, mật khẩu của họ được băm và lưu trữ. Khi đăng nhập, mật khẩu được cung cấp cũng được băm với salt từ hash đã lưu, sau đó hash mới được so sánh với hash đã lưu. Nếu hai hash khớp nhau, mật khẩu là đúng.

## DTO (Data Transfer Object)

* **DTO (Data Transfer Object)** là một đối tượng dữ liệu đơn giản được sử dụng để truyền dữ liệu giữa các lớp hoặc giữa các hệ thống (ví dụ: giữa API và client).
* **Lợi ích**:
  + **Kiểm soát dữ liệu đầu ra**: DTO cho phép chỉ trả về những trường dữ liệu cần thiết và an toàn cho client, ngăn chặn việc lộ thông tin nhạy cảm (như PasswordHash) hoặc không cần thiết. Ví dụ: UserDetailsDto chỉ chứa Id, Username, Email, CreatedAt mà không có PasswordHash.
  + **Kiểm soát dữ liệu đầu vào**: DTO cũng giúp định nghĩa rõ ràng những dữ liệu nào client có thể gửi lên (ví dụ: UserDto cho đăng ký/đăng nhập, UserUpdateDto cho cập nhật).
  + **Đảm bảo tính nhất quán**: Giúp API duy trì một giao diện rõ ràng và ổn định cho việc truyền dữ liệu.
  + **Tách biệt mối quan tâm**: Tách biệt mô hình database (Entity classes) khỏi mô hình API (DTOs), cho phép thay đổi database schema mà không ảnh hưởng trực tiếp đến giao diện API.

# Thiết kế Cơ sở dữ liệu

Dự án Web API Quản lý Người dùng sử dụng cơ sở dữ liệu quan hệ **SQL Server** với một bảng chính để lưu trữ thông tin về người dùng.

## Sơ đồ bảng (User Table)

Trong cơ sở dữ liệu, có một bảng duy nhất được đặt tên là Users. Bảng này chứa các thông tin cần thiết để định danh và xác thực người dùng. Cấu trúc chi tiết của bảng Users như sau:

|  |  |  |
| --- | --- | --- |
| Tên cột (Column Name) | Kiểu dữ liệu (Data Type) | Mô tả (Description) |
| Id | int | **Khóa chính (Primary Key)**, tự động tăng (IDENTITY). Định danh duy nhất cho mỗi người dùng. |
| Username | nvarchar(255) | Tên đăng nhập của người dùng. **Yêu cầu duy nhất** và không được phép null. |
| PasswordHash | nvarchar(255) | Chuỗi băm (hash) của mật khẩu người dùng. **Không phải mật khẩu gốc**. Kết quả của quá trình băm bằng BCrypt.Net. |
| Email | nvarchar(255) | Địa chỉ email của người dùng. Có thể là null. |
| CreatedAt | datetime2 | Thời điểm người dùng tạo tài khoản. Tự động được ghi lại khi một bản ghi mới được thêm vào. |

**Mô tả thêm về bảng Users:**

* **Id**: Là khóa chính của bảng, đảm bảo mỗi người dùng có một định danh số duy nhất. Việc tự động tăng giúp quản lý ID dễ dàng.
* **Username**: Đây là trường quan trọng để xác thực người dùng. Việc bắt buộc nó phải **duy nhất** đảm bảo không có hai tài khoản trùng tên đăng nhập.
* **PasswordHash**: Đây là một cột cực kỳ quan trọng về bảo mật. Nó lưu trữ phiên bản đã được băm (hash) của mật khẩu. Không bao giờ lưu mật khẩu dưới dạng văn bản gốc để bảo vệ thông tin người dùng trong trường hợp database bị lộ.
* **Email**: Cung cấp một phương thức liên lạc bổ sung với người dùng. Việc cho phép null mang lại sự linh hoạt nếu email không phải là bắt buộc ngay từ đầu.
* **CreatedAt**: Giúp theo dõi thời gian tạo tài khoản, hữu ích cho việc quản lý và phân tích dữ liệu.

## Mối quan hệ

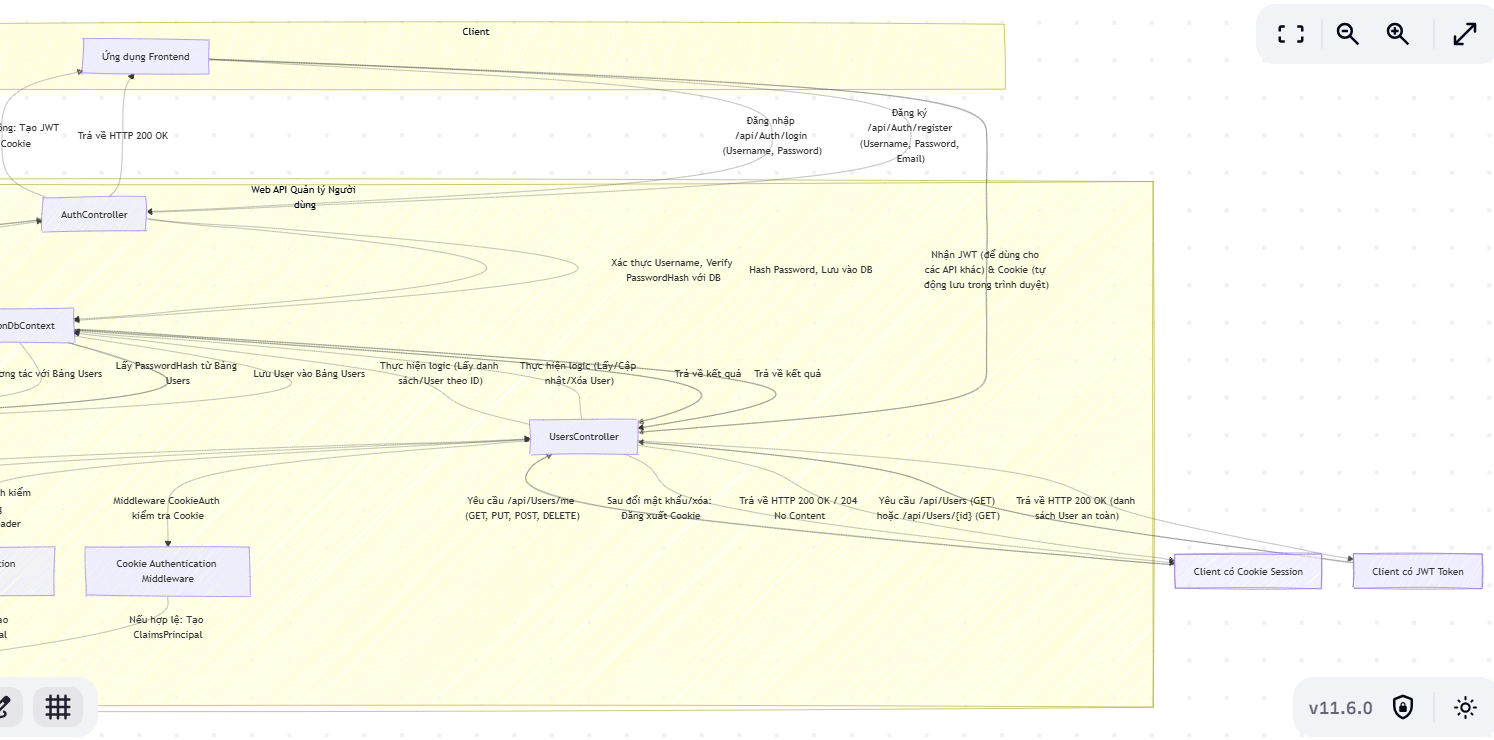
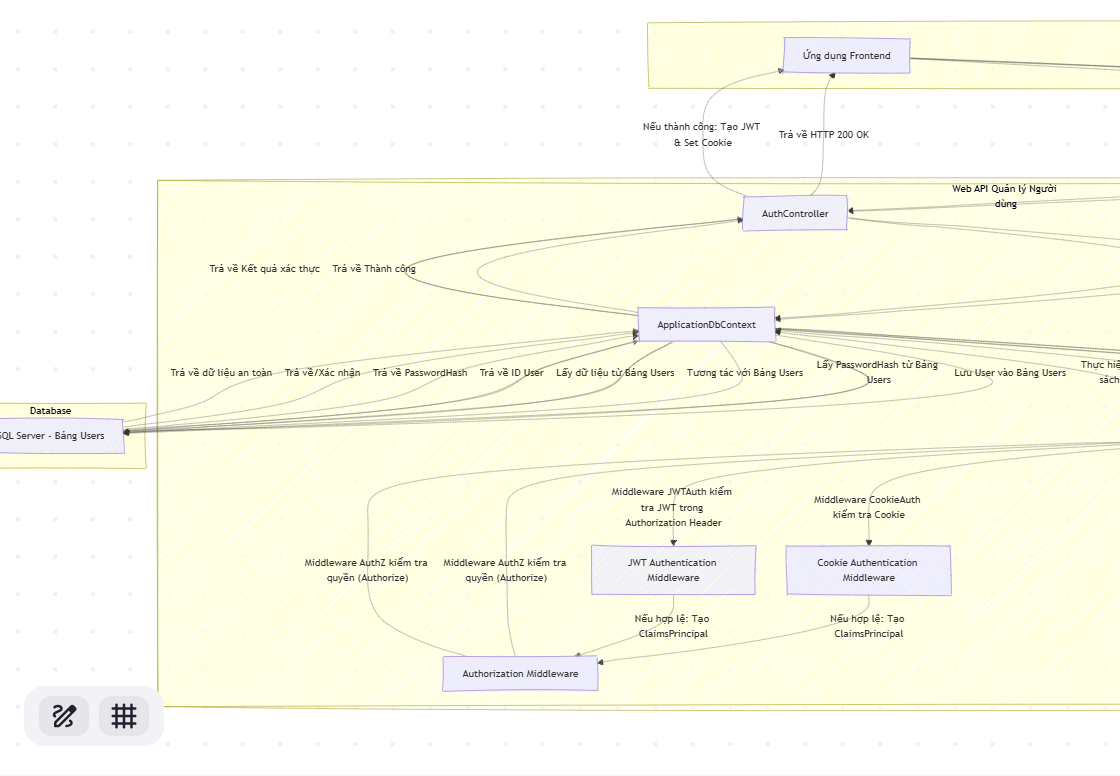
Trong dự án hiện tại, chỉ có một bảng Users, do đó **không có mối quan hệ trực tiếp** giữa các bảng. Đây là một cơ sở dữ liệu đơn giản cho mục đích quản lý người dùng cơ bản.

Nếu dự án được mở rộng trong tương lai (ví dụ: thêm vai trò người dùng, thông tin hồ sơ chi tiết, nhật ký hoạt động), tôi sẽ giới thiệu thêm các bảng mới (ví dụ: Roles, UserRoles, Profiles, Logs) và thiết lập các mối quan hệ khóa ngoại (Foreign Key) giữa chúng để duy trì tính toàn vẹn của dữ liệu và cho phép truy vấn dữ liệu phức tạp hơn

# Luồng hoạt động chính của API

Để hiểu rõ cách dự án hoạt động, việc nắm bắt luồng trao đổi giữa client và server là rất quan trọng. Dưới đây là sơ đồ và giải thích các luồng hoạt động chính của API:

## Sơ đồ luồng hoạt động tổng thể



## Giải thích các luồng hoạt động chi tiết

1. **Luồng Đăng ký Người dùng (API: POST /api/Auth/register)**
   * **Client**: Gửi yêu cầu POST đến /api/Auth/register với thông tin **Username**, **Password** (plain text) và **Email**.
   * **AuthController**: Nhận yêu cầu.
     + Kiểm tra tính hợp lệ của dữ liệu đầu vào (ví dụ: email đúng định dạng, username không trùng).
     + Sử dụng thư viện **BCrypt.Net** để **băm (hash)** mật khẩu được cung cấp.
     + Tạo đối tượng User và lưu vào **ApplicationDbContext**.
     + Gọi \_context.SaveChangesAsync() để lưu dữ liệu vào **Bảng Users** trong **SQL Server**.
   * **Server**: Trả về phản hồi HTTP 200 OK hoặc HTTP 400 Bad Request nếu có lỗi (ví dụ: username đã tồn tại).
2. **Luồng Đăng nhập Người dùng (API: POST /api/Auth/login)**
   * **Client**: Gửi yêu cầu POST đến /api/Auth/login với **Username** và **Password** (plain text).
   * **AuthController**: Nhận yêu cầu.
     + Truy vấn **ApplicationDbContext** để tìm người dùng với Username đã cho.
     + Nếu tìm thấy, sử dụng **BCrypt.Net.Verify()** để so sánh mật khẩu được cung cấp với PasswordHash đã lưu trong **Database**.
     + **Nếu xác thực thành công**:
       - Tạo một **JSON Web Token (JWT)** chứa các claims về người dùng (ID, Username). Token này được ký bằng khóa bí mật của server và có thời gian hết hạn.
       - Tạo một **cookie session** chứa thông tin xác thực của người dùng và thiết lập nó trong phản hồi HTTP. Trình duyệt client sẽ tự động lưu trữ và gửi cookie này trong các yêu cầu tiếp theo đến domain của API.
     + **Nếu xác thực thất bại**: Trả về HTTP 400 Bad Request hoặc HTTP 401 Unauthorized.
   * **Client**: Nhận JWT (thường là trong JSON response) và cookie (tự động bởi trình duyệt). JWT sẽ được lưu trữ cục bộ (ví dụ: localStorage hoặc sessionStorage) để sử dụng cho các API yêu cầu JWT, còn cookie sẽ được trình duyệt quản lý.
3. **Luồng Truy cập Endpoint Cá nhân (/api/Users/me - GET, PUT, POST, DELETE) (Yêu cầu Cookie Session)**
   * **Client (đã đăng nhập qua cookie)**: Gửi yêu cầu (GET, PUT, POST, DELETE) đến /api/Users/me hoặc /api/Users/me/change-password. Trình duyệt tự động đính kèm cookie session trong header Cookie.
   * **API (Middleware Authentication)**:
     + **Cookie Authentication Middleware** (được cấu hình trong Program.cs) kiểm tra cookie trong yêu cầu.
     + Nếu cookie hợp lệ, middleware sẽ giải mã nó và tạo ra một **ClaimsPrincipal** (đối tượng đại diện cho người dùng đã xác thực) và gán nó vào HttpContext.User.
   * **API (Middleware Authorization)**:
     + **Authorization Middleware** (được kích hoạt bởi [Authorize] attribute với CookieAuthenticationDefaults.AuthenticationScheme) kiểm tra ClaimsPrincipal. Nếu người dùng đã xác thực, yêu cầu sẽ được chuyển đến **UsersController**.
   * **UsersController**:
     + Trích xuất **ID người dùng** từ ClaimsPrincipal (qua User.FindFirst(ClaimTypes.NameIdentifier)).
     + Thực hiện logic nghiệp vụ:
       - **GET**: Truy vấn thông tin người dùng từ **Database** và trả về **UserDetailsDto** (chỉ các trường an toàn).
       - **PUT**: Cập nhật Username/Email của người dùng trong **Database**.
       - **POST /change-password**: Xác thực mật khẩu cũ, băm mật khẩu mới, cập nhật vào **Database**, sau đó **đăng xuất phiên hiện tại** (HttpContext.SignOutAsync) để tăng cường bảo mật.
       - **DELETE**: Xóa người dùng khỏi **Database**, sau đó **đăng xuất phiên hiện tại**.
   * **Server**: Trả về HTTP 200 OK (hoặc HTTP 204 No Content cho DELETE) nếu thành công, hoặc HTTP 401 Unauthorized, HTTP 404 Not Found, HTTP 400 Bad Request nếu có lỗi.
4. **Luồng Truy cập Endpoint Công khai (/api/Users và /api/Users/{id}) (Yêu cầu JWT)**
   * **Client (có JWT)**: Gửi yêu cầu GET đến /api/Users (để lấy tất cả người dùng) hoặc /api/Users/{id} (để lấy người dùng cụ thể). Client phải đính kèm JWT vào header Authorization dưới dạng Bearer Token (ví dụ: Authorization: Bearer <your\_jwt>).
   * **API (Middleware Authentication)**:
     + **JWT Bearer Authentication Middleware** (được cấu hình trong Program.cs) kiểm tra header Authorization.
     + Nó xác minh chữ ký của JWT bằng khóa bí mật của server, kiểm tra thời gian hết hạn và các claim khác.
     + Nếu JWT hợp lệ, middleware sẽ tạo ra một **ClaimsPrincipal** và gán nó vào HttpContext.User.
   * **API (Middleware Authorization)**:
     + **Authorization Middleware** (được kích hoạt bởi [Authorize] attribute với JwtBearerDefaults.AuthenticationScheme) kiểm tra ClaimsPrincipal. Nếu người dùng đã xác thực, yêu cầu sẽ được chuyển đến **UsersController**.
   * **UsersController**:
     + **/api/Users**: Truy vấn tất cả người dùng từ **Database**. Sử dụng LINQ Select để ánh xạ dữ liệu sang **UserDetailsForListDto** (chỉ ID, Username, Email) để đảm bảo chỉ trả về thông tin an toàn.
     + **/api/Users/{id}**: Truy vấn người dùng cụ thể theo ID từ **Database**. Trả về các thông tin an toàn (ID, Username, Email, CreatedAt).
   * **Server**: Trả về HTTP 200 OK với dữ liệu người dùng, hoặc HTTP 401 Unauthorized nếu JWT không hợp lệ/không có, hoặc HTTP 404 Not Found nếu không tìm thấy người dùng.

# Kết luận và Hướng phát triển

## Tóm tắt những gì đã đạt được

Dự án **Web API Quản lý Người dùng** đã được xây dựng thành công trên nền tảng ASP.NET Core, cung cấp một bộ API RESTful mạnh mẽ và bảo mật cho việc quản lý thông tin người dùng. Trong quá trình phát triển đã đạt được các mục tiêu quan trọng sau:

* **Triển khai các chức năng nghiệp vụ cốt lõi**: Bao gồm đăng ký, đăng nhập, xem, cập nhật, thay đổi mật khẩu và xóa tài khoản cá nhân.
* **Áp dụng bảo mật nâng cao**: Sử dụng thư viện **BCrypt.Net** để băm mật khẩu một cách an toàn, ngăn chặn rò rỉ mật khẩu gốc.
* **Hỗ trợ đa dạng phương thức xác thực**: Kết hợp cả **Cookie-based Authentication (session)** cho các tác vụ quản lý hồ sơ cá nhân trên trình duyệt và **JWT (JSON Web Token) Authentication** cho các API không trạng thái, phù hợp với nhiều loại ứng dụng client khác nhau (web, mobile, desktop).
* **Quản lý cơ sở dữ liệu hiệu quả**: Sử dụng **Entity Framework Core** để tương tác với **SQL Server**, đơn giản hóa việc truy vấn, thêm, sửa, xóa dữ liệu thông qua các đối tượng C#.
* **Cấu trúc dự án rõ ràng và dễ bảo trì**: Áp dụng các nguyên tắc thiết kế như **Dependency Injection** và sử dụng **DTOs (Data Transfer Objects)** để tách biệt mối quan tâm và kiểm soát dữ liệu đầu vào/đầu ra.
* **Hỗ trợ tích hợp frontend**: Cấu hình **CORS** để cho phép ứng dụng frontend chạy trên các nguồn gốc khác có thể kết nối và sử dụng API một cách an toàn.
* **Tài liệu hóa API**: Tích hợp **Swagger/OpenAPI** cung cấp tài liệu API tương tác, giúp việc kiểm thử và hiểu các endpoint trở nên dễ dàng hơn.

Dự án này là một nền tảng vững chắc, minh chứng cho khả năng xây dựng các dịch vụ backend hiện đại, bảo mật và có khả năng mở rộng bằng ASP.NET Core.

## Hướng phát triển trong tương lai

Để mở rộng và hoàn thiện dự án, có một số hướng phát triển tiềm năng có thể được xem xét:

1. **Quản lý Vai trò và Phân quyền (Role-Based Access Control - RBAC)**:
   * Thêm bảng Roles và UserRoles vào cơ sở dữ liệu.
   * Tích hợp tính năng quản lý vai trò (ví dụ: Admin, User) và gán quyền hạn khác nhau cho các vai trò đó.
   * Sử dụng [Authorize(Roles = "Admin")] để bảo vệ các endpoint yêu cầu quyền quản trị.
2. **Reset Mật khẩu (Forgot Password)**:
   * Triển khai cơ chế reset mật khẩu thông qua email, bao gồm việc tạo và xác minh token reset mật khẩu.
3. **Xác thực hai yếu tố (Two-Factor Authentication - 2FA)**:
   * Thêm một lớp bảo mật nữa bằng cách yêu cầu người dùng xác minh danh tính qua một yếu tố thứ hai (ví dụ: mã OTP từ ứng dụng xác thực hoặc SMS).
4. **Logging và Xử lý lỗi nâng cao**:
   * Tích hợp một framework logging mạnh mẽ (ví dụ: Serilog, NLog) để ghi lại các sự kiện và lỗi của ứng dụng.
   * Triển khai Middleware xử lý lỗi toàn cục (Exception Handling Middleware) để bắt và xử lý các ngoại lệ một cách tập trung, cung cấp phản hồi lỗi nhất quán cho client.
5. **Giới hạn tốc độ yêu cầu (Rate Limiting)**:
   * Áp dụng rate limiting để ngăn chặn các cuộc tấn công vét cạn hoặc DoS bằng cách giới hạn số lượng yêu cầu mà một client có thể gửi trong một khoảng thời gian nhất định.
6. **Triển khai vào Container (Docker)**:
   * Đóng gói ứng dụng API vào Docker container để dễ dàng triển khai, quản lý và mở rộng trên các môi trường khác nhau (ví dụ: Kubernetes).
7. **Sử dụng Response Caching/Redis**:
   * Đối với các endpoint đọc dữ liệu thường xuyên được truy cập, có thể áp dụng caching để cải thiện hiệu suất và giảm tải cho cơ sở dữ liệu.