|  |
| --- |
| TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN  **KHOA CÔNG NGHỆ THÔNG TIN**  **--------------------------**  **DEVELOPER GUIDE**  **Ứng dụng quản lý học sinh**  Thành viên:  Vòng Sau Hùng - 22120118  Lê Nguyễn Hồng Ngọc - 22120232  Phan Tấn Phát - 22120264  **Thành phố Hồ Chí Minh – Năm 2025** |

Mục lục

[**I.** **Coding Standards** 3](#_Toc201347489)

[**II.** **Overview of Architecture** 5](#_Toc201347490)

[**III.** **Source code organization** 9](#_Toc201347491)

[**IV.** **Getting Started with Your App Development** 17](#_Toc201347492)

[**1.** **Yêu cầu hệ thống:** 17](#_Toc201347493)

[**2.** **Cài đặt NodeJS (nếu chưa có):** 18](#_Toc201347494)

[**3.** **Cài đặt pnpm:** 18](#_Toc201347495)

[**4.** **Cài đặt ứng dụng Quản lý Sinh Viên** 19](#_Toc201347496)

[**5.** **Biên dịch và chạy chương trình** 19](#_Toc201347497)

[**V.** **Database Schema** 20](#_Toc201347498)

[**1.** **Tổng quan** 20](#_Toc201347499)

[**2.** **Thông tin sinh viên** 21](#_Toc201347500)

[**3.** **Thông tin môn học** 22](#_Toc201347501)

[**4.** **Thông tin lớp học** 22](#_Toc201347502)

[**5.** **Thông tin đăng ký & kết quả học tập** 22](#_Toc201347503)

[**6.** **Hỗ trợ đa ngôn ngữ** 23](#_Toc201347504)

[**VI.** **Updating an existing entity (How to add a new property)** 23](#_Toc201347505)

[**1.** **Các bước thực hiện** 23](#_Toc201347506)

[**2.** **Ví dụ cụ thể** 24](#_Toc201347507)

[**VII.** **Registering New Routes** 24](#_Toc201347508)

[**1.** **Các bước tạo route mới** 24](#_Toc201347509)

[**2.** **Ví dụ: Thêm route GET /students/:id** 25](#_Toc201347510)

[**3.** **Kiểm tra đăng ký route** 25](#_Toc201347511)

[**VIII.** **Inversion of Control and Dependency Injection** 25](#_Toc201347512)

[**IX.** **Data Validation** 26](#_Toc201347513)

[**X.** **Unit Testing** 29](#_Toc201347514)

[**1.** **Công cụ và Triết lý** 29](#_Toc201347515)

[**XI.** **Web API documentation** 32](#_Toc201347516)

[**XII.** **Reference** 37](#_Toc201347517)

# **Coding Standards**

* 1. **Ngôn ngữ và Framework**
     1. **Ngôn ngữ:** TypeScript. Tận dụng tối đa các tính năng của TypeScript như kiểu dữ liệu tĩnh (static typing), interface, generic để tăng độ an toàn và giảm lỗi runtime.
     2. **Framework:** NestJS. Tuân thủ các mẫu thiết kế và quy ước của NestJS như Modules, Controllers, Providers (Services), và Dependency Injection.
  2. **Cấu trúc Dự án và Kiến trúc**
     1. **Cấu trúc Module:** Mỗi chức năng nghiệp vụ chính phải được đặt trong một module riêng tại src/modules/{tên\_module}.
     2. **Cấu trúc bên trong Module:**

domain/: Chứa logic nghiệp vụ cốt lõi, hoàn toàn độc lập với framework.

port/input/: Định nghĩa các interface cho Service (I{Tên}Service.ts) và lớp triển khai Service ({tên}.service.ts).

port/output/: Định nghĩa các interface cho Repository (I{Tên}Repository.ts).

dto/: Chứa các lớp Data Transfer Object (\*.dto.ts).

adapters/: Chứa các lớp triển khai cụ thể.

driver/: Chứa Controller (\*.controller.ts), là điểm tiếp nhận request từ bên ngoài.

driven/: Chứa lớp triển khai Repository (\*.repository.ts), chịu trách nhiệm giao tiếp với cơ sở dữ liệu.

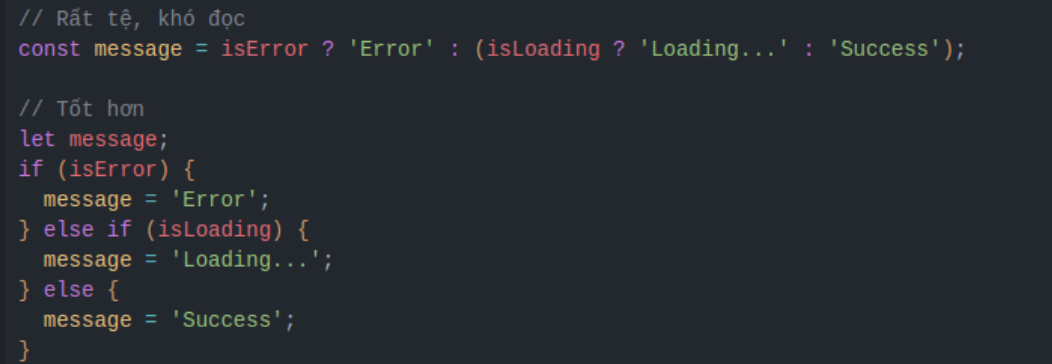
* + 1. **Thư mục shared/:** Chỉ chứa các mã nguồn có thể tái sử dụng trên toàn bộ ứng dụng, ví dụ: PrismaService, Logger, các Interceptor, và các kiểu dữ liệu chung.
    2. **Quy ước Đặt tên (Naming Conventions)**
* Tên file: Sử dụng kebab-case.type.ts (ví dụ: programs.service.ts, create-program.dto.ts).
* Lớp (Class): Sử dụng PascalCase (ví dụ: ProgramsService, CreateProgramDTO).
* Interface: Sử dụng PascalCase với tiền tố I (ví dụ: IProgramsService, IProgramsRepository).
* Biến, Hàm, Thuộc tính: Sử dụng camelCase (ví dụ: programsRepository, findById).
* Hằng số (Constants): Sử dụng UPPER\_SNAKE\_CASE (ví dụ: PROGRAM\_REPOSITORY, TRANSLATABLE\_FIELDS).
* Decorator: Sử dụng PascalCase (ví dụ: @Injectable(), @Controller())
  + 1. **Nguyên tắc Kiến trúc và Lập trình**
* Quy tắc Phụ thuộc (Dependency Rule): Luồng phụ thuộc luôn hướng vào trong: Adapters -> Ports -> Domain. Tuyệt đối không để mã nguồn trong domain phụ thuộc vào bất kỳ thứ gì bên ngoài nó (không import từ NestJS, Prisma trong các file logic nghiệp vụ thuần túy).
* Sử dụng cơ chế Dependency Injection của NestJS một cách triệt để.
* Sử dụng constructor injection để tiêm các phụ thuộc.
* Khi tiêm một Repository, sử dụng token đã định nghĩa (@Inject(PROGRAM\_REPOSITORY)).
* Data Transfer Objects (DTOs):
* Mọi dữ liệu đi vào (request body, query params) hoặc đi ra (response body) khỏi API đều phải được định nghĩa bằng các lớp DTO.
* Sử dụng các thư viện như class-validator và class-transformer để tự động xác thực và chuyển đổi dữ liệu tại tầng Controller hoặc qua ValidationPipe.
* Controller: Chỉ chịu trách nhiệm nhận request, xác thực dữ liệu đầu vào cơ bản (thông qua DTO), và gọi Service tương ứng. Không chứa logic nghiệp vụ.
* Service: Chứa toàn bộ logic nghiệp vụ. Đây là nơi thực hiện các quy trình, tính toán, và điều phối các hoạt động.
* Repository: Chỉ chịu trách nhiệm duy nhất là giao tiếp với nguồn dữ liệu (ví dụ: CSDL). Không chứa logic nghiệp vụ.
  + 1. **Xử lý Lỗi (Error Handling)**
* Sử dụng các lớp exception có sẵn của NestJS (HttpException, NotFoundException, BadRequestException, ForbiddenException...) tại tầng Adapter (Controller) để trả về các mã lỗi HTTP phù hợp.
* Trong các Service, khi một thao tác không thành công (ví dụ: không tìm thấy bản ghi), hãy throw một exception phù hợp để Controller có thể bắt và xử lý.
* Sử dụng try...catch để xử lý các lỗi có thể xảy ra từ các thao tác không an toàn (I/O, gọi API ngoài).
* Luôn sử dụng Logger (@nestjs/common) để ghi lại lỗi với đầy đủ ngữ cảnh.
  + 1. **Lập trình Bất đồng bộ (Asynchronous Programming)**
* Sử dụng async/await: Mọi hoạt động bất đồng bộ (ví dụ: truy vấn cơ sở dữ liệu, gọi API ngoài, đọc file) phải được xử lý bằng cú pháp async/await. Điều này giúp mã nguồn dễ đọc và dễ theo dõi hơn so với việc sử dụng chuỗi .then().catch().



* + 1. **Logic Điều kiện (Conditional Logic)**
* **Toán tử ba ngôi (Ternary Operator ? :):** Nên sử dụng: Cho các phép gán điều kiện đơn giản, ngắn gọn để làm mã nguồn súc tích hơn.



* **Nên tránh:** Lồng các toán tử ba ngôi vào nhau. Việc này làm mã nguồn trở nên cực kỳ khó đọc và khó bảo trì. Hãy sử dụng if/else hoặc switch trong trường hợp này.



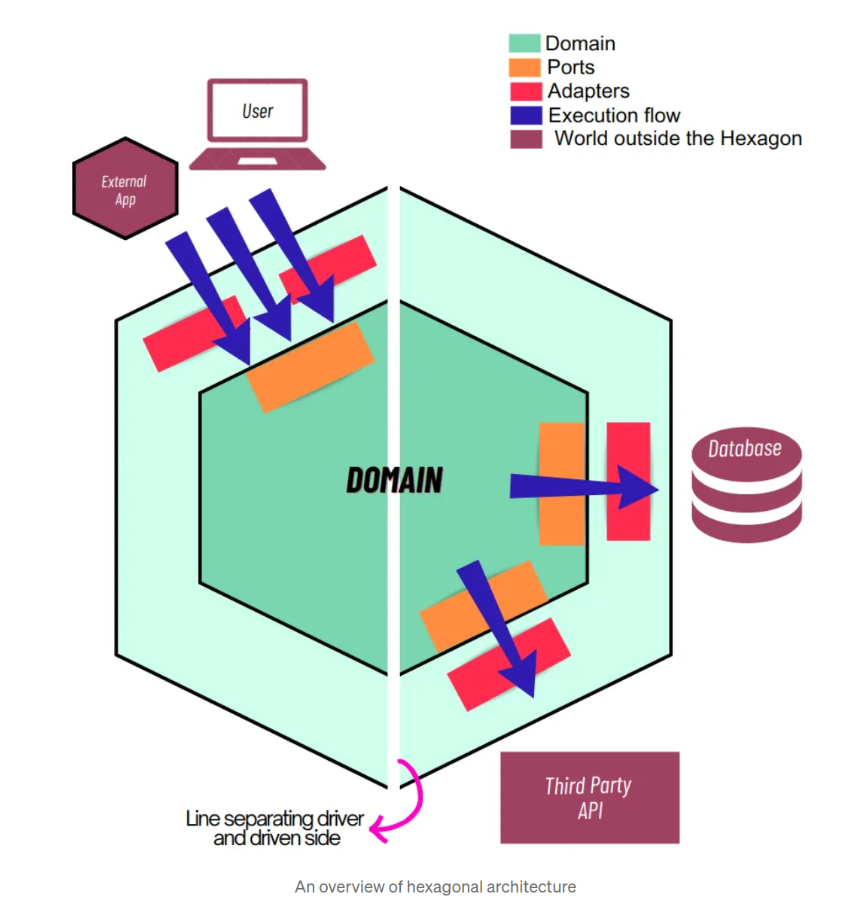
# **Overview of Architecture**

* 1. Kiến trúc của Hệ thống Quản lý Sinh viên được thiết kế với mục tiêu xây dựng một hệ thống linh hoạt, dễ bảo trì, có khả năng mở rộng và hiệu suất cao. Chúng tôi đã lựa chọn một tập hợp các công nghệ và mẫu kiến trúc hiện đại để đạt được những mục tiêu này.
  2. Kiến trúc tổng thể được xây dựng dựa trên ba trụ cột chính:
     1. Kiến trúc Monorepo: Sử dụng Turborepo và pnpm workspaces để quản lý toàn bộ mã nguồn trong một kho chứa duy nhất.
     2. Kiến trúc Backend: Ứng dụng NestJS được cấu trúc theo mẫu Kiến trúc Lục giác (Hexagonal Architecture), hay còn gọi là Ports and Adapters.
     3. Kiến trúc Frontend: Ứng dụng Next.js được xây dựng với các component theo triết lý Atomic Design.
  3. Kiến trúc Monorepo: Toàn bộ dự án được tổ chức dưới dạng monorepo, nghĩa là mã nguồn của cả backend (api), frontend (web), và các gói chia sẻ (packages) đều nằm chung trong một repository. pnpm Workspaces: Quản lý các phụ thuộc một cách hiệu quả, cho phép các gói nội bộ (ví dụ: web) phụ thuộc vào các gói khác (ví dụ: ui) mà không cần phải publish chúng lên một registry công khai.
  4. Turborepo: Một công cụ build system thông minh, giúp tối ưu hóa các tác vụ trong monorepo. Nó có khả năng cache lại kết quả của các tác vụ (như build, lint, test), đảm bảo rằng các tác vụ chỉ được chạy lại khi mã nguồn liên quan có sự thay đổi. Điều này giúp tăng tốc đáng kể chu trình phát triển và CI/CD.

**Kiến trúc Backend (api):**

Backend được xây dựng bằng NestJS, một framework Node.js mạnh mẽ cho các ứng dụng phía máy chủ. Cấu trúc của backend tuân thủ nghiêm ngặt theo mẫu Kiến trúc Hexagonal Architecture (Ports and Adapters). **Hexagonal Architecture** là một mẫu kiến trúc phần mềm nhằm tách biệt các hệ thống bên ngoài khỏi lõi ứng dụng. Miền nghiệp vụ của ứng dụng tồn tại bên trong hình lục giác, nơi tập trung giải quyết vấn đề chính bằng cách định nghĩa một mô hình (entities) liên quan đến một phần cụ thể của logic nghiệp vụ. Đây được xem là hình thức phát triển mã ứng dụng thuần túy nhất. Tuy nhiên, điều này chưa thực sự hữu ích nếu logic nghiệp vụ bị cô lập hoàn toàn bên trong ranh giới của hình lục giác. Chúng ta cần có các cơ chế để nhận đầu vào và trả đầu ra, do đó cần đặt ra một số giả định về cách mà ứng dụng sẽ tương tác với thế giới bên ngoài. Trong kiến trúc Lục giác, tồn tại hai khái niệm là driving và driven, mang ý nghĩa đúng như tên gọi: phía driving là bên khởi xướng hành động, trong khi phía driven là bên tiếp nhận hành động đó và thực hiện các xử lý tương ứng.

* 1. **Mục tiêu:** Tách biệt hoàn toàn logic nghiệp vụ cốt lõi (domain) khỏi các chi tiết về cơ sở hạ tầng (như framework web, cơ sở dữ liệu, dịch vụ bên ngoài).
  2. **Domain** (Miền nghiệp vụ): Là trái tim của ứng dụng, chứa toàn bộ các quy tắc, thực thể và logic nghiệp vụ. Nó hoàn toàn "thuần túy", không chứa bất kỳ mã nguồn nào liên quan đến NestJS hay Prisma.
  3. **Ports (Các cổng):** Một **giao diện** hoặc **hợp đồng** xác định các thuộc tính cần thiết để tương tác với miền nghiệp vụ (domain). Nó cho thế giới bên ngoài biết những gì cần thiết để có thể vận hành được với miền nghiệp vụ. Một cách hiểu khác về “port” (cổng) là: đó là một **thoả thuận** mà bất kỳ bên nào cũng phải tuân thủ nếu muốn tương tác với những gì nằm bên trong “chiếc hộp” — hay trong trường hợp của chúng ta, **bên trong hình lục giác**.
     1. **Input Ports (Driving Ports):** Định nghĩa các chức năng mà ứng dụng cung cấp cho thế giới bên ngoài (ví dụ: IStudentService định nghĩa các hành động như createStudent, getStudentById).
     2. **Output Ports (Driven Ports):** Định nghĩa các dịch vụ mà domain cần từ bên ngoài để hoạt động (ví dụ: IStudentRepository định nghĩa các hành động như save, findById).
  4. **Adapters (Các bộ chuyển đổi):** Một lớp hoặc hàm được tùy chỉnh phù hợp với công nghệ đang được triển khai (nó xác định công việc và các bước chuyển đổi cần thiết để tương tác với thế giới bên ngoài).
     1. **Driving Adapters:** Là các thành phần nhận yêu cầu từ bên ngoài và gọi vào các Input Port. Ví dụ: StudentController là một adapter nhận request HTTP và gọi StudentService.
     2. **Driven Adapters:** Là các thành phần triển khai các Output Port. Ví dụ: PrismaStudentRepository là một adapter triển khai IStudentRepository bằng cách sử dụng Prisma để tương tác với cơ sở dữ liệu.

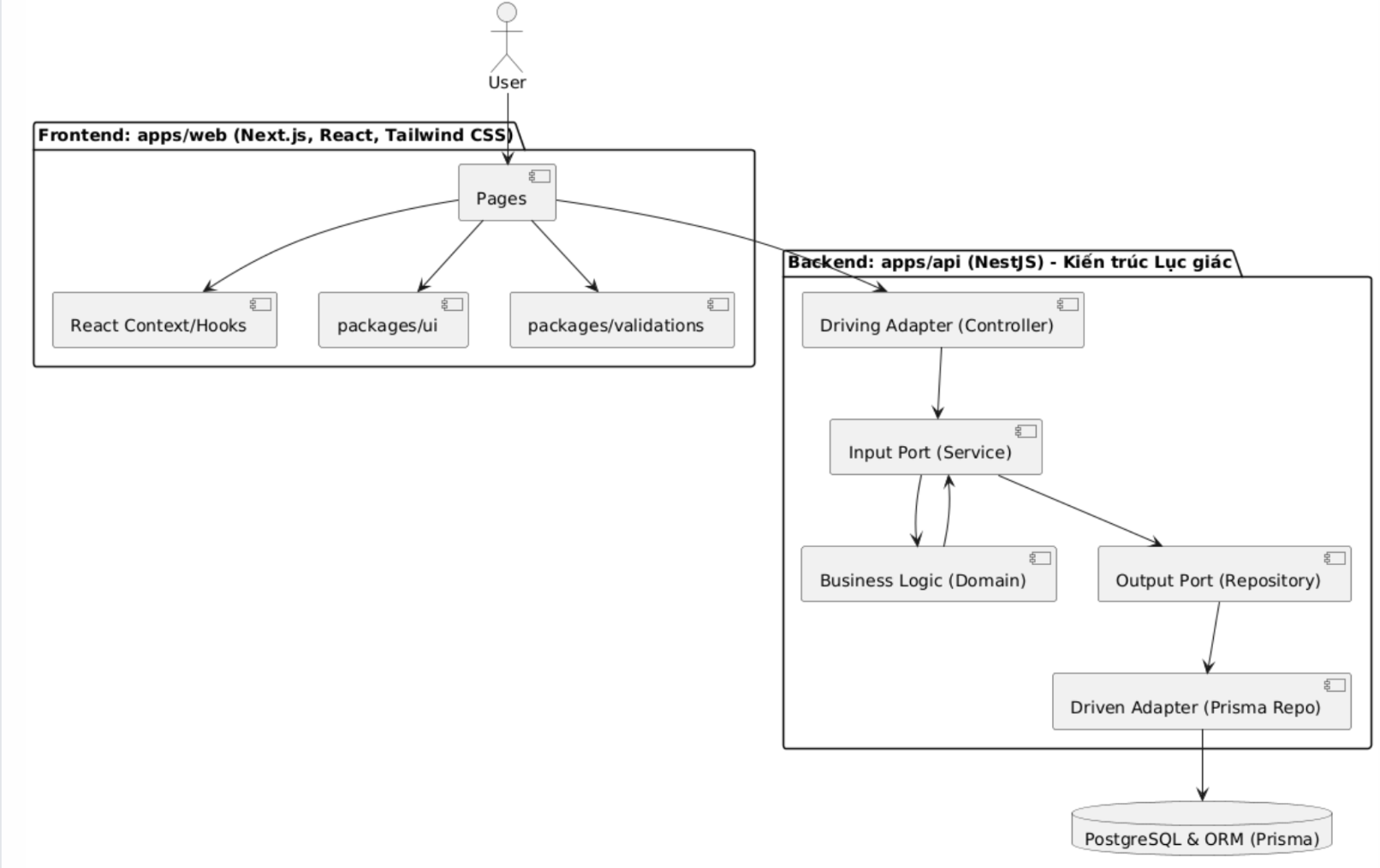


**Các lợi ích chính:**

* **Độc lập với Công nghệ và Framework (Technology & Framework Agnostic):** Logic nghiệp vụ của bạn không hề biết đến sự tồn tại của NestJS, Express, Prisma hay bất kỳ công nghệ cụ thể nào. Nó chỉ giao tiếp qua các interface (Ports).
* **Khả năng Kiểm thử (Testability) Vượt trội:** Vì logic nghiệp vụ hoàn toàn độc lập, bạn có thể viết các bài unit test cho nó một cách cực kỳ dễ dàng và nhanh chóng. Bạn không cần phải khởi tạo server, kết nối CSDL hay mock các request HTTP phức tạp. Ví dụ: Để test StudentService, bạn chỉ cần tạo một "mock" repository trong bộ nhớ (in-memory) để giả lập việc lưu và đọc dữ liệu, giúp test chạy nhanh và đáng tin cậy.
* **Dễ dàng Bảo trì và Mở rộng (Maintainability & Scalability):** Các ranh giới giữa nghiệp vụ và hạ tầng rất rõ ràng. Khi cần sửa lỗi hoặc thêm tính năng, bạn biết chính xác cần phải thay đổi ở đâu. Ví dụ: Nếu cần thêm một cách tương tác mới với hệ thống, ví dụ như qua một giao diện dòng lệnh (CLI) hoặc một message queue (RabbitMQ), bạn chỉ cần tạo một "Driving Adapter" mới mà không ảnh hưởng đến các controller hiện có hay logic nghiệp vụ.
* **Tập trung vào Logic Nghiệp vụ (Focus on Business Logic):** Kiến trúc này buộc các lập trình viên phải suy nghĩ về nghiệp vụ trước tiên, thay vì bị cuốn vào các chi tiết kỹ thuật của framework. Điều này giúp sản phẩm cuối cùng bám sát và giải quyết tốt hơn các vấn đề của người dùng.

**Flow hoạt động**

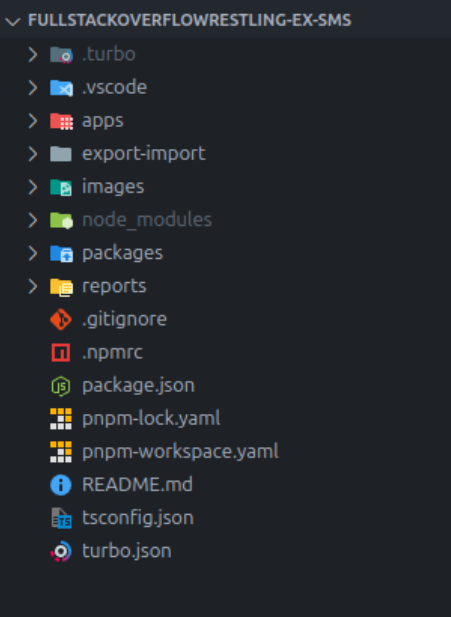
* Hệ thống được thiết kế theo mô hình Hexagonal Architecture (Ports and Adapters) nhằm đảm bảo tính tách biệt giữa logic nghiệp vụ (business logic) và các công nghệ cụ thể bên ngoài. Luồng xử lý bắt đầu từ phía người dùng (User), tương tác trực tiếp với frontend thông qua giao diện web được xây dựng với Next.js, React, và Tailwind CSS trong module apps/web. Trong frontend, người dùng thao tác trên các Pages đại diện cho từng màn hình hoặc chức năng. Trạng thái của ứng dụng được quản lý thông qua React Context hoặc Hooks, trong khi các thành phần giao diện (UI components) được tái sử dụng từ thư viện nội bộ packages/ui. Dữ liệu người dùng nhập sẽ được xác thực ngay phía client bằng các schema từ packages/validations, đảm bảo đúng định dạng trước khi gửi về backend.
* Khi người dùng gửi dữ liệu (chẳng hạn như đăng ký, đăng nhập, tạo nội dung...), frontend sẽ thực hiện HTTP/HTTPS API call đến backend (apps/api). Tại backend, ứng dụng sử dụng NestJS và được tổ chức theo kiến trúc lục giác. Lớp ngoài cùng tiếp nhận yêu cầu là Driving Adapter, cụ thể là các lớp Controller chịu trách nhiệm ánh xạ các route HTTP đến các luồng xử lý bên trong. Controller không xử lý trực tiếp logic nghiệp vụ mà chuyển yêu cầu vào Input Port, chính là các Service, đại diện cho những hành vi mà ứng dụng có thể thực hiện.
* Bên trong Service, logic nghiệp vụ cốt lõi được xử lý thông qua các thực thể và luật trong phần Domain, chính là “trái tim” của ứng dụng, nơi định nghĩa các thực thể, quy tắc, và mối quan hệ nghiệp vụ. Domain không phụ thuộc vào bất kỳ công nghệ cụ thể nào và hoàn toàn độc lập. Nếu Service cần tương tác với bên ngoài (ví dụ: lưu hoặc truy vấn dữ liệu), nó sẽ gọi đến Output Port (Repository Interface) – đây là nơi định nghĩa các hành vi cần thiết mà domain mong đợi từ hạ tầng.
* Các Driven Adapters chịu trách nhiệm hiện thực hóa các Output Port này, cụ thể là các lớp repository triển khai bằng Prisma – công cụ ORM được sử dụng để thao tác với cơ sở dữ liệu. Các Driven Adapter sẽ xử lý chi tiết kỹ thuật như tạo query SQL, kết nối và thực hiện thao tác với cơ sở dữ liệu PostgreSQL. Toàn bộ quá trình này diễn ra theo một chiều rõ ràng: User → Frontend → Controller → Service → Domain → Repository Interface → Repository Implementation → Database, và chiều ngược lại là dữ liệu phản hồi trở về theo thứ tự tương ứng.
* Cách tổ chức theo Hexagonal Architecture giúp cho logic nghiệp vụ luôn được giữ sạch sẽ, dễ kiểm thử, và hoàn toàn tách biệt với các công nghệ bên ngoài như framework, HTTP, hoặc cơ sở dữ liệu. Việc thay đổi adapter (ví dụ từ Prisma sang một ORM khác, hoặc từ REST sang GraphQL) có thể thực hiện mà không ảnh hưởng đến miền nghiệp vụ.



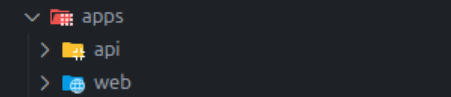
**Luồng cơ bản của kiến trúc Hexagonal Architecture**

# **Source code organization**

* 1. Dự án "Hệ thống Quản lý Sinh viên" được xây dựng theo kiến trúc monorepo, một phương pháp phát triển phần mềm trong đó mã nguồn của nhiều dự án con được lưu trữ trong cùng một kho chứa (repository). Kiến trúc này mang lại nhiều lợi ích như chia sẻ mã nguồn dễ dàng, quản lý phụ thuộc tập trung, và quy trình làm việc nhất quán.
  2. Chúng tôi sử dụng Turborepo để quản lý và tối ưu hóa các tác vụ trong monorepo (như build, lint, test) và pnpm với tính năng workspaces để quản lý các gói (packages) và phụ thuộc (dependencies) một cách hiệu quả.
  3. Cấu trúc tổng thể được thiết kế để phân tách rõ ràng các mối quan tâm (separation of concerns), tăng khả năng bảo trì, mở rộng và tái sử dụng mã nguồn.



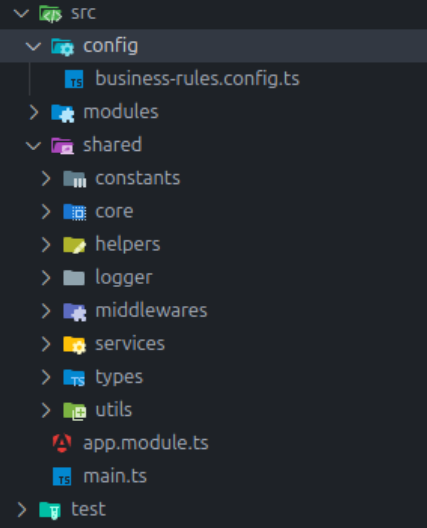
* 1. **pnpm-workspace.yaml:** Tệp này khai báo cho pnpm biết rằng các thư mục con bên trong apps và packages là các phần của một workspace. Điều này cho phép pnpm liên kết các gói cục bộ với nhau và tối ưu hóa việc cài đặt phụ thuộc.
  2. **package.json:** Tệp package.json ở gốc định nghĩa các script chung cho toàn bộ dự án, chẳng hạn như build, dev, lint. Các script này sử dụng turbo để chạy tác vụ tương ứng trên tất cả các apps và packages một cách song song và hiệu quả.
  3. **turbo.json:** Đây là trái tim của việc quản lý monorepo. Nó định nghĩa "pipeline" của các tác vụ. Ví dụ, nó chỉ định rằng tác vụ build của một ứng dụng phụ thuộc vào tác vụ build của các gói mà nó sử dụng ("dependsOn": ["^build"]). Turborepo cũng sử dụng cơ chế caching thông minh để không chạy lại các tác vụ đã được thực thi, giúp tăng tốc đáng kể quá trình build và CI/CD.
  4. **tsconfig.json:** Cung cấp một cấu hình TypeScript cơ sở. Các tsconfig.json trong từng app hoặc package có thể kế thừa và mở rộng cấu hình này.
  5. **Thư mục apps** - Các Ứng dụng Chính:



* + 1. **API (Backend)**

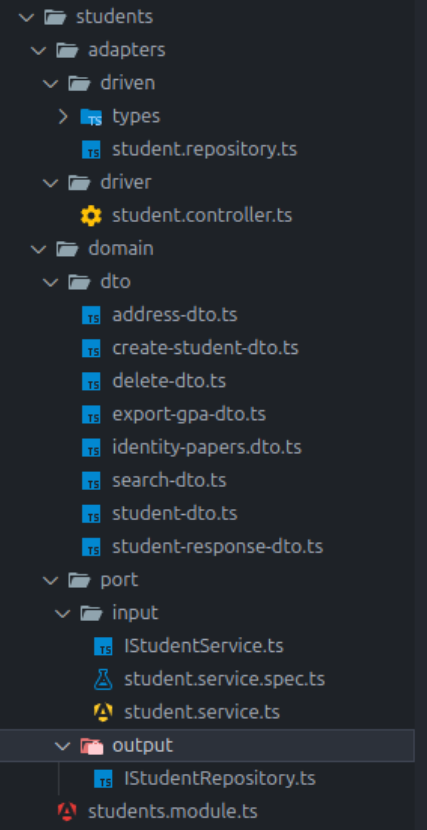
Đây là ứng dụng backend, được xây dựng bằng NestJS, cung cấp các API cho ứng dụng web và các bên thứ ba**.**

* **prisma/schema.prisma:** Định nghĩa schema cho cơ sở dữ liệu bằng Prisma ORM. Đây là nơi duy nhất định nghĩa mô hình dữ liệu.
* **main.ts:** Điểm khởi đầu (entry point) của ứng dụng NestJS.
* **app.module.ts:** Module gốc của ứng dụng.
* **shared/:** Chứa các thành phần dùng chung trong toàn bộ API như PrismaService, interceptors, middlewares, helpers. Mục tiêu chính của thư mục này là tuân thủ nguyên tắc DRY (Don't Repeat Yourself - Đừng lặp lại chính mình), giúp cho mã nguồn trở nên sạch sẽ, dễ bảo trì và nhất quán. Bất cứ khi nào một chức năng hoặc một đoạn mã có khả năng được sử dụng ở nhiều nơi, nó nên được đặt vào đây.
  + **constants/**: mục đích lưu trữ các giá trị hằng số không thay đổi trong suốt quá trình chạy ứng dụng. Việc tập trung các hằng số vào một nơi giúp dễ dàng quản lý, thay đổi và tránh "magic strings" hoặc "magic numbers" (các giá trị không rõ ý nghĩa xuất hiện trực tiếp trong code). Ví dụ (student.constant.ts): Có thể chứa các hằng số liên quan đến sinh viên như vai trò mặc định, trạng thái học tập ('studying', 'graduated', 'suspended'), hoặc các khóa (key) dùng cho caching.
  + **core/:** mục đích chứa các thành phần cốt lõi, mang tính "cross-cutting concern" (mối quan tâm xuyên suốt) của framework NestJS. Ví dụ (interceptors/): Chứa các NestJS Interceptor. Interceptor có thể "chặn" các yêu cầu (request) và phản hồi (response) để thêm vào các logic chung.
  + **helpers/:** mục đích chứa các lớp hoặc hàm trợ giúp (helper) để thực hiện các tác vụ cụ thể, thường là để đơn giản hóa các logic phức tạp. api-response.ts: Cung cấp các hàm hoặc class để tạo ra các đối tượng phản hồi API một cách nhất quán. Ví dụ, một hàm SuccessResponse(data) và ErrorResponse(message, code). error.ts: Định nghĩa các lớp lỗi tùy chỉnh (custom exceptions) kế thừa từ HttpException của NestJS. Ví dụ: UserNotFoundException, InvalidInputException để các service có thể throw những lỗi có ngữ nghĩa rõ ràng.
  + **logger/:** mục đích thiết lập và cấu hình hệ thống ghi log cho toàn bộ ứng dụng. Ghi log là cực kỳ quan trọng để theo dõi hoạt động, gỡ lỗi và giám sát hệ thống. Ví dụ (logger.module.ts): Module này có thể cấu hình một thư viện ghi log mạnh mẽ như Winston hoặc Pino. Nó sẽ tạo ra một LoggerService có thể được inject (tiêm) vào bất kỳ controller hoặc service nào khác trong ứng dụng để ghi log một cách có cấu trúc (ví dụ: this.logger.error('Failed to connect to database', context)).
  + **middlewares/:** mục đích chứa các service dùng chung. Khác với service trong một module chức năng (chỉ phục vụ cho module đó), shared service cung cấp các chức năng cho toàn bộ ứng dụng. database/: Thường chứa PrismaService, một service đóng gói PrismaClient. Service này quản lý việc kết nối và ngắt kết nối tới cơ sở dữ liệu và có thể được inject vào các repository để thực hiện truy vấn. shared.module.ts: Đây là một file rất quan trọng. Nó là một NestJS Module có nhiệm vụ export tất cả các provider (services, helpers...) trong thư mục shared. Bất kỳ feature module nào (ví dụ StudentsModule) muốn sử dụng PrismaService hoặc LoggerService chỉ cần import SharedModule này vào là có thể sử dụng được.
  + **types/:** mục đích định nghĩa các kiểu dữ liệu (types) và interface TypeScript được sử dụng ở nhiều nơi trong ứng dụng. Ví dụ PaginatedResponse.ts: Định nghĩa một kiểu generic cho các phản hồi có phân trang.
  + **utils/:** mục đích chứa các hàm tiện ích nhỏ, thuần túy (pure functions), không phụ thuộc vào framework hay trạng thái của ứng dụng. Ví dụ các hàm xử lý chuỗi, định dạng ngày tháng, tính toán, tạo mã ngẫu nhiên...

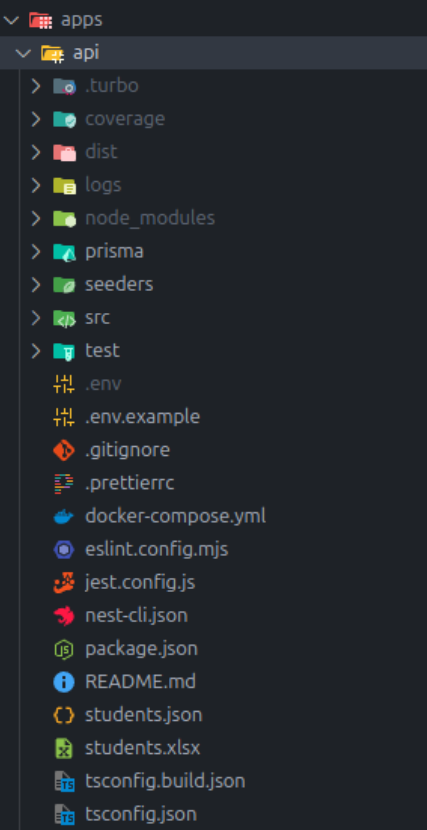


*Ảnh chi tiết của shared*

* **modules/:** Mỗi thư mục con là một module chức năng (feature module) tương ứng với một miền nghiệp vụ (ví dụ: students, faculties, programs).
* Cấu trúc của một module (ví dụ students):
  + **students.module.ts:** Khai báo module, kết nối các thành phần lại với nhau.
  + **domain/:** Logic nghiệp vụ cốt lõi.
  + **port/input/IStudentService.ts:** Interface cho service, định nghĩa các chức năng như createStudent, findStudentById.
  + **port/input/student.service.ts:** Lớp service triển khai IStudentService, chứa logic nghiệp vụ chính.
  + **port/output/IStudentRepository.ts:** Interface cho repository, định nghĩa các phương thức truy vấn CSDL như save, findById.
  + **dto/:** Data Transfer Objects, các đối tượng dùng để truyền dữ liệu giữa các lớp.
  + **adapters/:** Các lớp "cầu nối"
  + **driver/student.controller.ts:** Controller nhận các request HTTP, xác thực đầu vào và gọi đến StudentService.
  + **driven/student.repository.ts:** Repository triển khai IStudentRepository, sử dụng PrismaService để thực hiện các thao tác với CSDL.



*Ảnh chi tiết của một module*

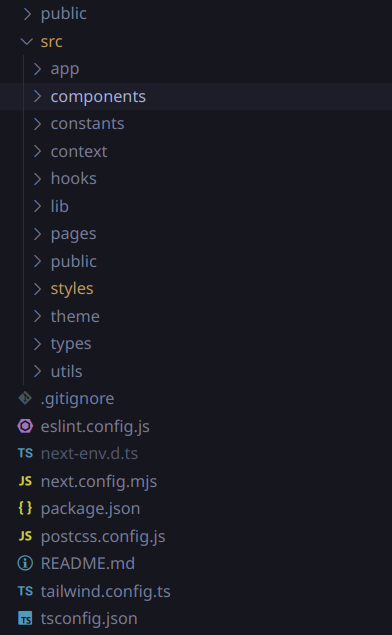
****

*Ảnh cấu trúc tổng quát API Backend*

* + 1. **Web (Frontend)**

Đây là ứng dụng frontend, được xây dựng bằng NextJS, cung cấp giao diện cho ứng dụng.

* **app/:** Routing và layout chính cho ứng dụng. Đây là folder giúp NextJS route request của người dùng đến giao diện thích hợp
* **components/:** Đứa các components để xây dựng giao diện chung cho ứng dụng. Các component này là các component tái sử dụng, được sử dụng ở nhiều trang khác nhau. Cấu trúc được chia theo Atomic Design gồm 4 thư mục:
  + Atoms: các thành phần nhỏ nhất cấu thành nên giao diện như Button, Label, Card,...
  + Molecules: cụm các Atom kết hợp với nhau hoặc các thành phần phức tạp hơn một chút như Dialog, Form nhỏ, Section,...
  + Organisim: các component lớn, gồm nhiều các Molecules và Atoms, chứa một vài logic phức tạp hơn như Form lớn, mutable data component
  + Pages: layout cho 1 trang của ứng dụng. Đặt các thành phần nhỏ hơn để tạo nên 1 trang web cụ thể và thêm các chức năng cần thiết
* **pages/:** Chứa các component chuyên dụng cho 1 page cụ thể. Các component này chỉ hoạt động độc lập tại 1 page nhất định và không dùng cho các page khác
* **constants/:** Chứa các file định nghĩa các giá trị config, file dịch nội dung tĩnh
* **context**/: React Context, lưu trữ các context của React
* **hooks**/: chứa các custom hook của React
* **lib**/: chứa các service để gọi về Backend cũng như các thư viện phụ thuộc bên ngoài
* **types**/: chứa các file định nghĩa các kiểu dữ liệu cho Typescript
* **styles**/: css cho ứng dụng
* **theme**/: chứa theme provider để cập nhật theme cho ứng dụng khi người dùng thay đổi



*Ảnh cấu trúc tổng quát Frontend*

# **Getting Started with Your App Development**

## **Yêu cầu hệ thống:**

* **Hệ điều hành:** Windows 10/11, macOS 10.14 trở lên, hoặc Linux (Ubuntu 18.04 trở lên)
* **NodeJS**: Phiên bản 16.0.0 trở lên
* **RAM**: Tối thiểu 2 GB
* **Dung lượng ổ cứng trống**: 1 GB
* **Trình duyệt web**: Chrome, Firefox, Edge hoặc Safari phiên bản mới nhất

## **Cài đặt NodeJS (nếu chưa có):**

* + - * **Kiểm tra Node.js đã được cài chưa**
* Mở Command Prompt (Windows) hoặc Terminal (Mac/Linux)
* Nhập lệnh: node -v
* Nếu hiển thị phiên bản Node.js (ví dụ: v16.15.0), bạn đã cài đặt rồi
* Nếu báo lỗi, tiếp tục các bước sau
  + - * **Tải NodeJS**
* Truy cập trang web: *https://nodejs.org/*
* Chọn phiên bản LTS (Long Term Support) để tải xuống
* Nhấp vào nút tải xuống phù hợp với hệ điều hành của bạn và lưu file cài đặt
  + - * **Cài đặt Node.js**
* Windows*:*
  + Mở file .**msi** vừa tải xuống
  + Nhấn "Next" và làm theo hướng dẫn trên màn hình
  + Chọn tất cả các tính năng mặc định và tiếp tục
  + Đợi quá trình cài đặt hoàn tất và nhấn "Finish"
* *macOS:*
  + Mở file .*pkg* vừa tải xuống
  + Làm theo các bước trên màn hình để hoàn tất cài đặt
* *Linux***:**

|  |
| --- |
| **curl fsSL https://deb.nodesource.com/setup\_16.x | sudo -E bash - sudo apt-get install -y nodejs** |

* + - * **Kiểm tra lại cài đặt**
* Mở lại Command Prompt hoặc Terminal
* Nhập lệnh: node -v
* Nhập lệnh: npm -v
* Nếu cả hai lệnh đều hiển thị phiên bản, cài đặt đã thành công

## **Cài đặt pnpm:**

* + - * **Mở Command Prompt (Windows) hoặc Terminal (Mac/Linux)**
      * **Cài đặt pnpm bằng một trong các cách sau:**
* Sử dụng npm:

|  |
| --- |
| npm install -g pnpm |

* Hoặc trên Windows với PowerShell:

|  |
| --- |
| iwr https://get.pnpm.io/install.ps1 -useb | iex |

* Hoặc trên macOS/Linux:

|  |
| --- |
| curl -fsSL https://get.pnpm.io/install.sh | sh - |

* + - * **Kiểm tra cài đặt:**

|  |
| --- |
| pnpm --version |

## **Cài đặt ứng dụng Quản lý Sinh Viên**

* + - * **Tải** **ứng dụng**
* Nếu bạn sử dụng Git:
  + Mở Command Prompt hoặc Terminal
  + Điều hướng đến thư mục bạn muốn lưu ứng dụng:

|  |
| --- |
| cd đường-dẫn-đến-thư-mục |

* + Clone repository:

|  |
| --- |
| git clone https://github.com/Shungisme/Fullstackoverflowrestling-Ex-SMS |

* + Di chuyển vào thư mục dự án:

|  |
| --- |
| cd Fullstackoverflowrestling-Ex-SMS |

* Hoặc tải xuống dưới dạng ZIP:
  + Tải file ZIP từ đường link: [[Link tải ứng dụng]](https://github.com/Shungisme/Fullstackoverflowrestling-Ex-SMS/archive/refs/heads/main.zip)
  + Giải nén vào thư mục bạn muốn lưu ứng dụng
  + Mở Command Prompt hoặc Terminal và điều hướng đến thư mục vừa giải nén
    - * **Cài đặt các thư viện phụ thuộc**
* Mở Command Prompt hoặc Terminal trong thư mục dự án
* Chạy lệnh:

|  |
| --- |
| pnpm install |

* Đợi cho quá trình cài đặt hoàn tất (có thể mất vài phút tùy thuộc vào tốc độ internet)

## **Biên dịch và chạy chương trình**

* + - * **Thiết lập môi trường cho dự án**
        + Bổ sung các file.env
        + Generate Prisma Client

Mở Command Prompt hoặc Terminal

Điều hướng đến thư mục bạn đã lưu ứng dụng:

|  |
| --- |
| cd đường-dẫn-đến-thư-mục |

Điều hướng đến thư mục chứa mã nguồn backend:

|  |
| --- |
| cd apps/api |

Khởi tạo Prisma Client

|  |
| --- |
| pnpm prisma generate |

* + - * **Biên dịch dự án**
        + Mở Command Prompt hoặc Terminal trong thư mục gốc của dự án
        + Chạy lệnh:

|  |
| --- |
| pnpm build |

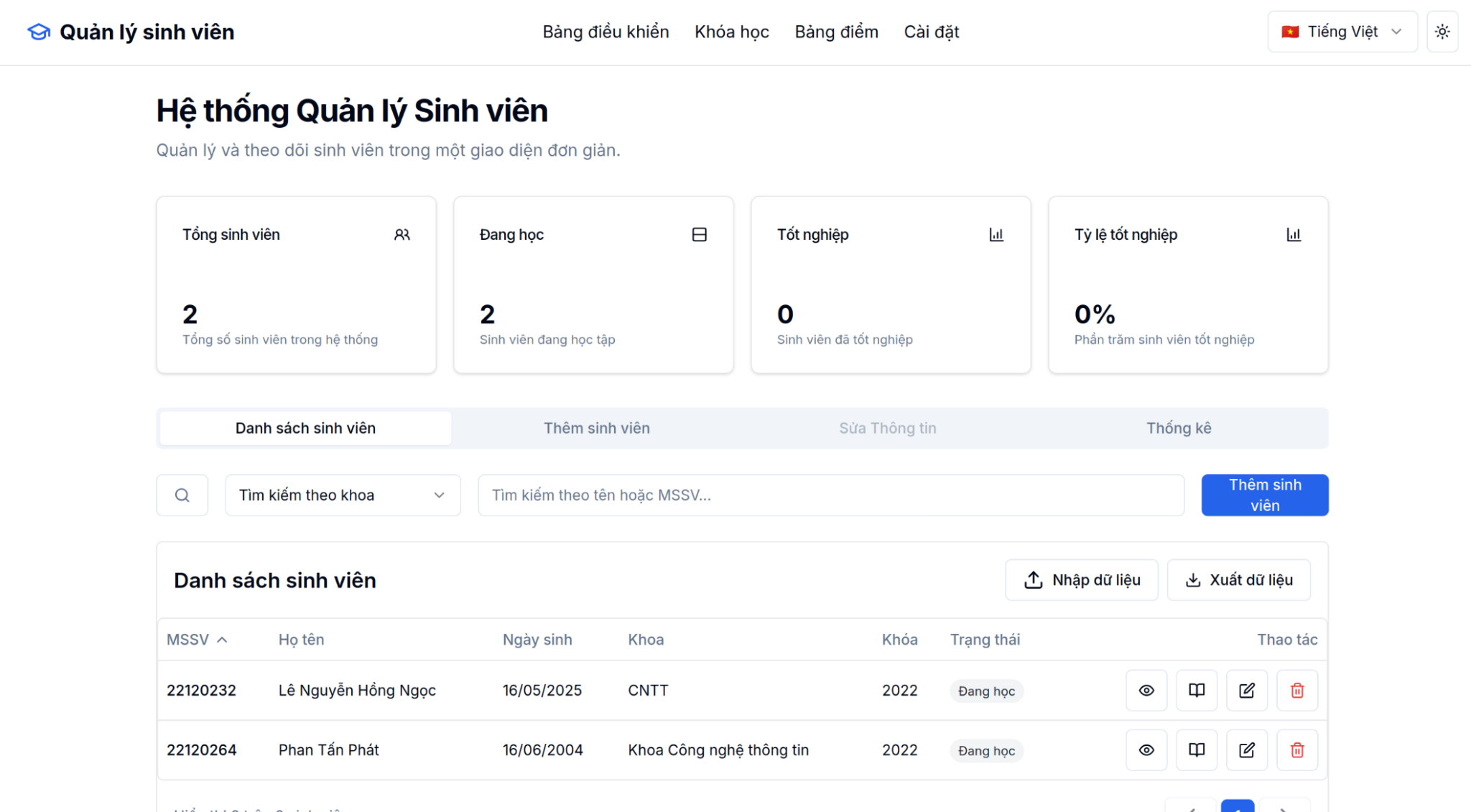
* + - * + Đợi cho quá trình biên dịch hoàn tất. Bạn sẽ thấy thông báo thành công.
      * **Chạy ứng dụng**
        + Sau khi biên dịch thành công, chạy lệnh:

|  |
| --- |
| pnpm dev |

* + - * + Truy cập ứng dụng:

Mở trình duyệt web

Truy cập địa chỉ: <http://localhost:3000>



# **Database Schema**

## **Tổng quan**

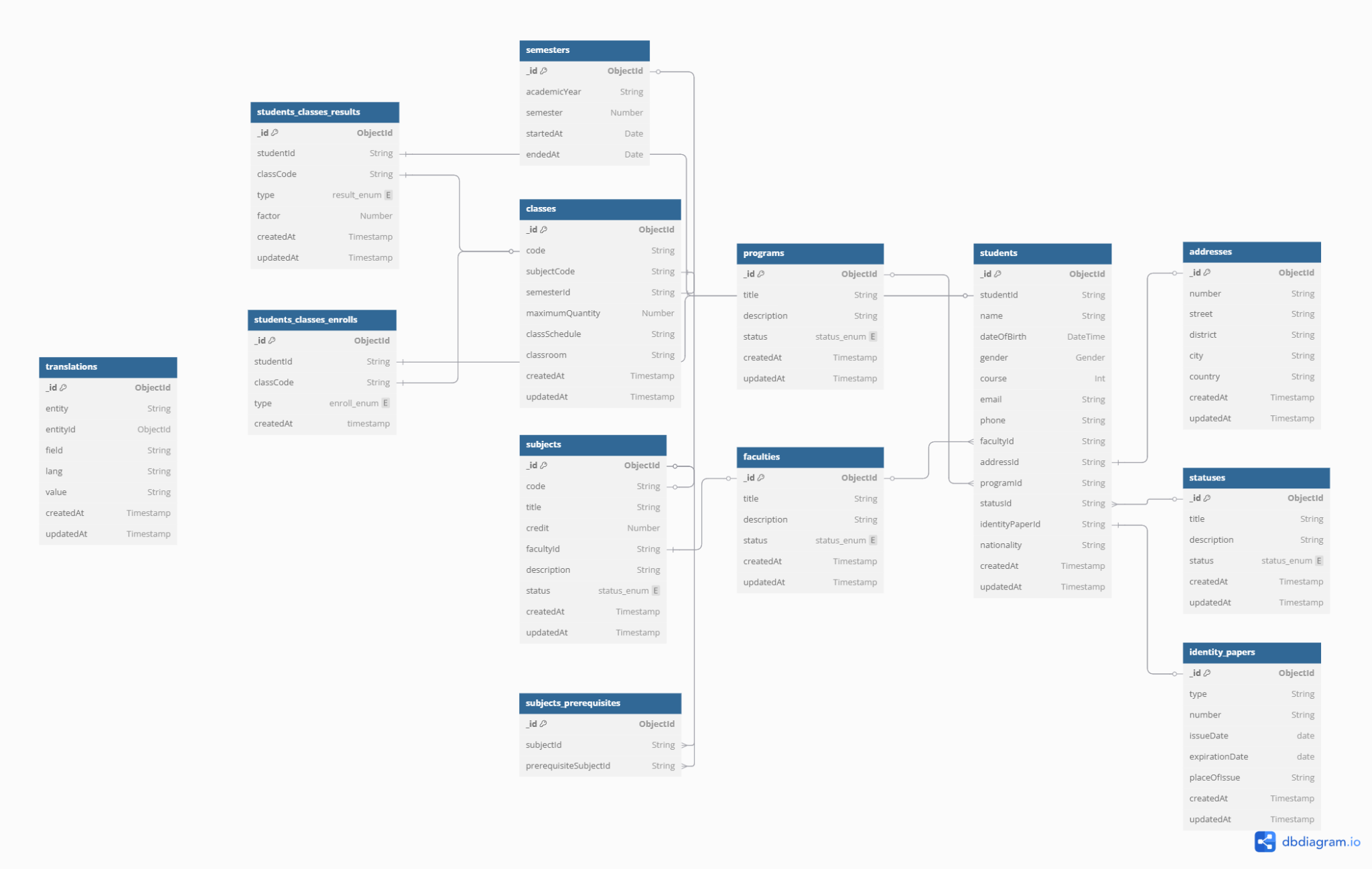
Phần này mô tả chi tiết **cấu trúc cơ sở dữ liệu mặc định** được cài đặt trong quá trình khởi tạo hệ thống. Schema này bao gồm tất cả các cấu trúc dữ liệu cần thiết để hỗ trợ các chức năng cốt lõi: **quản lý sinh viên, tổ chức học tập, đăng ký lớp và theo dõi kết quả**.

Hệ thống sử dụng cơ sở dữ liệu MongoDB kết hợp với Prisma ORM để truy vấn dữ liệu. Tất cả bảng (collections) được chia nhóm logic để tăng khả năng mở rộng, bảo trì và dễ tích hợp vào các hệ thống dịch vụ.

* **Cơ sở dữ liệu**: MongoDB
* **Truy cập dữ liệu**: Prisma ORM
* **Quy tắc đặt tên**: tên cột sử dụng **camelCase, tên bảng sử dụng snakecase**

Tổng cộng có **13 bảng chính (collections)**, được chia thành các nhóm chức năng như sau:

* Thông tin sinh viên
* Thông tin môn học
* Thông tin lớp học
* Thông tin đăng ký & kết quả học tập
* Hỗ trợ đa ngôn ngữ



## **Thông tin sinh viên**

Khối chức năng này đóng vai trò trung tâm trong toàn bộ hệ thống, chịu trách nhiệm lưu trữ và quản lý toàn diện các thông tin liên quan đến sinh viên. Thiết kế dữ liệu ở đây đảm bảo tính chi tiết, đầy đủ, nhưng vẫn phân tách rõ ràng để thuận tiện cho việc mở rộng hoặc bảo trì.

Hồ sơ sinh viên bao gồm các thuộc tính cơ bản như mã số sinh viên, họ tên, ngày sinh, giới tính, quốc tịch, khóa học, email và số điện thoại. Ngoài ra, mỗi sinh viên còn được gắn với các thông tin phụ trợ như:

* **Khoa** (Faculty): đơn vị quản lý hành chính, giúp tổ chức sinh viên theo bộ môn, phòng ban hoặc chuyên ngành.
* **Chương trình đào tạo** (Program): phản ánh lộ trình học tập, từ hệ cử nhân đến thạc sĩ.
* **Trạng thái học tập** (Status): cho biết sinh viên đang học, bảo lưu, thôi học hoặc đã tốt nghiệp.
* **Giấy tờ định danh** (Identity): bao gồm CMND, CCCD, hoặc hộ chiếu – giúp quản lý danh tính sinh viên theo chuẩn pháp lý.
* **Địa chỉ**: tách riêng thành ba loại – thường trú, tạm trú, và địa chỉ liên lạc – phục vụ cho các nghiệp vụ như gửi thư mời, thông báo học vụ, hoặc xác minh thông tin.

Việc tách riêng các thực thể phụ này ra khỏi bảng sinh viên chính không chỉ giúp dữ liệu có tính chuẩn hóa mà còn hỗ trợ mở rộng trong tương lai, chẳng hạn: một sinh viên có thể thay đổi địa chỉ hoặc giấy tờ mà không ảnh hưởng đến toàn bộ hồ sơ.

## **Thông tin môn học**

Khối dữ liệu này mô hình hóa các môn học – thành phần cốt lõi trong chương trình đào tạo. Mỗi môn học không chỉ có tên và mã số, mà còn đi kèm thông tin về tín chỉ, mô tả, khoa quản lý và trạng thái hoạt động (đang sử dụng, tạm dừng, ngừng hoạt động…).

Đặc biệt, hệ thống còn hỗ trợ **cấu trúc tiên quyết** giữa các môn học – tức là sinh viên phải hoàn thành một số môn trước khi học môn khác. Mối quan hệ này được xây dựng qua bảng ánh xạ tự tham chiếu, cho phép thiết kế lộ trình học tập một cách rõ ràng và có kiểm soát.

## **Thông tin lớp học**

Mỗi môn học có thể được mở thành nhiều lớp khác nhau tùy theo học kỳ, giảng viên, lịch học, hoặc nhóm sinh viên. Vì vậy, lớp học là khái niệm động, mang tính tình huống và tổ chức thực tế trong từng kỳ học.

Hệ thống xây dựng lớp học dưới dạng thực thể riêng biệt, trong đó mỗi lớp liên kết với một môn học cụ thể và một học kỳ nhất định. Lớp học có thể có tên giảng viên, thời gian học (schedule), phòng học (classroom), số lượng sinh viên tối đa và trạng thái hiện tại.

Mô hình học kỳ cũng được trừu tượng hóa thành một thực thể riêng, chứa thông tin năm học, số thứ tự học kỳ (ví dụ: học kỳ 1, 2, hè), ngày bắt đầu và kết thúc. Điều này giúp phân tách dữ liệu học thuật theo thời gian, hỗ trợ tốt cho việc thống kê, báo cáo và truy vấn lịch sử lớp học.

## **Thông tin đăng ký & kết quả học tập**

Đây là khối dữ liệu phản ánh quá trình học tập của sinh viên – từ lúc đăng ký vào một lớp học cho đến khi có kết quả điểm số cuối cùng.

Hệ thống thiết kế tách biệt giữa **đăng ký lớp học** và **kết quả học tập**, giúp theo dõi và xử lý các tình huống linh hoạt như: đăng ký nhưng bỏ lớp, điểm chưa cập nhật, học lại, hoặc có nhiều cột điểm trong cùng một lớp.

Thông tin kết quả được lưu theo từng loại điểm (giữa kỳ, cuối kỳ, khác), mỗi loại có thể gắn hệ số riêng để phục vụ tính điểm trung bình. Điều này phù hợp với cơ chế chấm điểm tại các trường đại học, nơi điểm số không đơn thuần là một con số, mà có trọng số, quy đổi và ảnh hưởng đến học lực tổng thể.

## **Hỗ trợ đa ngôn ngữ**

Với nhu cầu mở rộng quốc tế, hệ thống hỗ trợ đa ngôn ngữ thông qua một cơ chế linh hoạt, không phụ thuộc vào số lượng ngôn ngữ cài sẵn. Cụ thể, mọi bản ghi có thể được dịch bằng cách ghi lại thông tin:

* Tên bảng
* Khóa thực thể
* Tên trường
* Ngôn ngữ đích
* Giá trị dịch

Nhờ cấu trúc này, frontend có thể dễ dàng hiển thị nội dung theo ngôn ngữ người dùng mà không cần sửa đổi bảng gốc hoặc schema. Đây là thiết kế mở, phù hợp với các hệ thống quản lý hiện đại, đặc biệt là nơi có nhu cầu sử dụng nhiều frontend, nhiều vùng ngôn ngữ (ví dụ: tiếng Việt, tiếng Anh, tiếng Pháp...).

# **Updating an existing entity (How to add a new property)**

Việc thêm một thuộc tính mới vào model trong Prisma rất linh hoạt và nhanh chóng, đặc biệt khi sử dụng MongoDB. Dưới đây là các bước hướng dẫn và ví dụ minh hoạ cụ thể.

## **Các bước thực hiện**

1. **Mở file schema Prisma (schema.prisma)**
   * Xác định model cần cập nhật.
2. **Thêm thuộc tính mới vào model**
   * Đặt tên thuộc tính, xác định kiểu dữ liệu và các chỉ định ràng buộc nếu có **(?**, **@default**, **@unique**,...).
3. **Chạy lệnh cập nhật Prisma**
   * Do MongoDB không dùng migration như SQL, nên bạn không cần chạy **prisma migrate**.
   * Chạy lệnh sau để cập nhật lại Prisma Client:  
      **npx prisma generate**
4. **Cập nhật logic xử lý trong backend (nếu có)**
   * Kiểm tra và xử lý phần tạo mới (**create**), cập nhật (**update**), và ánh xạ dữ liệu JSON nếu liên quan.

## **Ví dụ cụ thể**

Giả sử bạn muốn thêm trường **avatarUrl** vào model **Student** để lưu ảnh đại diện sinh viên.

**Trước khi cập nhật:**

*model Student {*

*id String @id @default(auto()) @map("\_id") @db.ObjectId*

*name String*

*email String*

*...*

*}*

**Sau khi thêm trường:**

*model Student {*

*id String @id @default(auto()) @map("\_id") @db.ObjectId*

*name String*

*email String*

*avatarUrl String // Thêm mới*

*...*

*}*

# **Registering New Routes**

Trong dự án sử dụng **NestJS**, việc đăng ký route mới tuân theo cấu trúc module-based rõ ràng và có tổ chức. Mỗi route mới thường được định nghĩa thông qua **Controller** tương ứng với logic nghiệp vụ ở **Service** và được gom nhóm trong **Module**.

## **Các bước tạo route mới**

1. **Xác định module liên quan**
   * Nếu đã có module tương ứng (VD: **students**, **classes**), bạn sẽ thêm controller hoặc method mới vào đó.
   * Nếu chưa có module, tạo module mới bằng CLI:

*nest g module example*

*nest g controller example*

*nest g service example*

1. **Thêm method trong Controller**
   * Định nghĩa route mới tại file **\*.controller.ts**, sử dụng các decorator như **@Get()**, **@Post()**, **@Patch()**, **@Delete()** để ánh xạ HTTP method.
2. **Kết nối đến Service xử lý logic**
   * Viết hàm xử lý logic tại **\*.service.ts**, controller gọi vào đây để tách biệt logic khỏi tầng định tuyến.
3. **Đảm bảo Module đã đăng ký controller và service**
   * Mỗi module cần **controllers: [XxxController] và providers: [XxxService].**

## **Ví dụ: Thêm route GET /students/:id**

**student.controller.ts**

*@Controller('students')*

*export class StudentController {*

*constructor(private readonly studentService: StudentService) {}*

*@Get(':id')*

*async getStudentById(@Param('id') id: string) {*

*return this.studentService.findById(id);*

*}*

*}*

**student.service.ts**

*@Injectable()*

*export class StudentService {*

*constructor(private readonly prisma: PrismaService) {}*

*async findById(id: string) {*

*return this.prisma.student.findUnique({ where: { id } });*

*}*

*}*

## **Kiểm tra đăng ký route**

Chạy server và truy cập **http://localhost:3000/students/{id}** hoặc dùng Postman để test.

# **Inversion of Control and Dependency Injection**

NestJS cung cấp một **IoC container** tích hợp giúp tự động quản lý vòng đời của các service, controller và các thành phần khác.

* Các lớp (classes) được đánh dấu bằng decorator **@Injectable()** hoặc **@Controller()** được tự động quản lý và inject phụ thuộc cần thiết.
* DI giúp loại bỏ việc khởi tạo thủ công các service trong controller, tăng tính module hóa và dễ dàng mock khi viết test.

**Service và Controller với DI:**

*@Injectable()*

*export class StudentService {*

*constructor(private readonly prisma: PrismaService) {}*

*findById(id: string) {*

*return this.prisma.student.findUnique({ where: { id } });*

*}*

*}*

*@Controller('students')*

*export class StudentController {*

*constructor(private readonly studentService: StudentService) {}*

*@Get(':id')*

*getStudent(@Param('id') id: string) {*

*return this.studentService.findById(id);*

*}*

*}*

* **StudentService** được **inject** **PrismaService** qua constructor.
* **StudentController** được **inject** **StudentService** qua constructor.
* NestJS IoC container sẽ tự động tạo và quản lý các instance này.

**Lưu ý:**

* Các class cần sử dụng DI phải được đăng ký trong **providers** của module.
* Các controller được đăng ký trong **controllers**.
* Sử dụng decorator **@Injectable()** cho service hoặc provider.

# **Data Validation**

Hệ thống sử dụng thư việnZod để xử lý xác thực dữ liệu đầu vào, kết hợp với **nestjs-zod** để tích hợp với hệ sinh thái NestJS. Việc xác thực giúp đảm bảo dữ liệu đầu vào từ client luôn chính xác, đủ thông tin và đúng định dạng trước khi được xử lý trong service hoặc lưu vào cơ sở dữ liệu.

Khác với cách tiếp cận truyền thống sử dụng **class-validator** và **class-transformer**, Zod cho phép định nghĩa schema kiểm tra dữ liệu bằng cách sử dụng hàm thuần, hỗ trợ tốt hơn cho TypeScript, dễ viết test và kiểm soát logic xác thực một cách rõ ràng hơn.

Trong hệ thống, các thực thể như **Student**, **IdentityPaper**, **Address** đều có các schema Zod riêng đi kèm, được tái sử dụng để sinh DTO đầu vào (**create, update**) và cả kiểu dữ liệu dùng trong service.

Ví dụ dưới đây mô tả quá trình xác thực dữ liệu cho thực thể **Student**:

*export const StudentSchema = z.object({*

*id: z*

*.string()*

*.refine((id) => ObjectId.isValid(id), { message: 'Invalid ObjectId' })*

*.optional(),*

*studentId: z.string().min(1, 'studentId cannot be empty'),*

*name: z.string().min(1, 'Name cannot be empty'),*

*dateOfBirth: z*

*.string()*

*.refine((date) => !isNaN(Date.parse(date)), {*

*message: 'Invalid date of birth format',*

*})*

*.transform((date) => new Date(date)),*

*gender: z.nativeEnum(Gender),*

*course: z.number().int().min(1, 'Course must be a positive integer'),*

*email: z*

*.string()*

*.email('Invalid email format')*

*.refine(*

*(email) => {*

*const emailDomain = email.split('@')[1];*

*return allowedEmailDomains.includes(emailDomain);*

*},*

*{*

*message: Email must belong to one of the following domains: ${allowedEmailDomains.join(', ')},*

*},*

*),*

*phone: z*

*.string()*

*.min(10)*

*.max(15)*

*.refine((phone) => phone.match(allowedPhoneNumber), {*

*message:*

*'Invalid phone number format. Must follow Vietnam format: +84xxxxxxxxx or 0xxxxxxxxx',*

*}),*

*nationality: z.string().min(1, 'Nationality cannot be empty'),*

*facultyId: z.string().min(1, 'Faculty cannot be empty'),*

*permanentAddressId: z.string().optional().nullable(),*

*temporaryAddressId: z.string().optional().nullable(),*

*mailingAddressId: z.string().min(1, 'Mailing address cannot be empty'),*

*identityPaperId: z.string().min(1, 'Identity paper cannot be empty'),*

*programId: z.string().min(1, 'Program cannot be empty'),*

*statusId: z.string().min(1, 'Status cannot be empty'),*

*});*

Một số kiểm tra quan trọng:

* **ObjectId**: tất cả các **id** liên quan đến MongoDB (ví dụ **facultyId**, **programId**, **identityPaperId**) đều được xác thực bằng **.refine((id) => ObjectId.isValid(id))**.
* **email**: ngoài việc dùng **.email()** để kiểm tra định dạng, còn xác thực rằng domain nằm trong danh sách cho phép được cấu hình qua **businessRulesConfig**.
* **phone**: xác thực số điện thoại theo định dạng Việt Nam với regex được cấu hình động.
* **dateOfBirth**: kiểm tra định dạng chuỗi ngày và chuyển đổi về Date object.

Sau khi khai báo schema, các DTO đầu vào được sinh ra từ schema này như sau:

*export class StudentRequestDTO extends createZodDto(*

*StudentSchema.omit({ id: true }),*

*) {}*

*export class UpdateStudentRequestDTO extends createZodDto(*

*StudentSchema.partial(),*

*) {}*

Việc sử dụng **createZodDto()** cho phép bạn truyền trực tiếp DTO này vào controller như sau:

*@Post()*

*@HttpCode(HttpStatus.CREATED)*

*@ZodSerializerDto(StudentDTO)*

*async create(@Body() dto: StudentRequestDTO) {*

*return this.studentService.create(dto);*

*}*

Nếu dữ liệu đầu vào không hợp lệ, hệ thống sẽ tự động trả về lỗi 400 với thông báo cụ thể được định nghĩa sẵn trong từng refine() hoặc rule của Zod:

*{*

*"statusCode": 400,*

*"message": [*

*"Email must belong to one of the following domains: student.edu.vn, myuniversity.vn"*

*],*

*"error": "Bad Request"*

*}*

Ngoài ra, bạn có thể cấu hình riêng các quy tắc nghiệp vụ như domain email, regex điện thoại… thông qua file **business-rules.config.ts**, giúp tách biệt logic validation khỏi logic controller và dễ dàng cập nhật khi cần thay đổi mà không ảnh hưởng code chính.

*const businessRulesConfig = new BusinessRulesConfig();*

*const allowedEmailDomains = businessRulesConfig.get('emailDomains');*

*const allowedPhoneNumber = businessRulesConfig.get('phoneRegex');*

Nhờ vào Zod, bạn có thể dễ dàng:

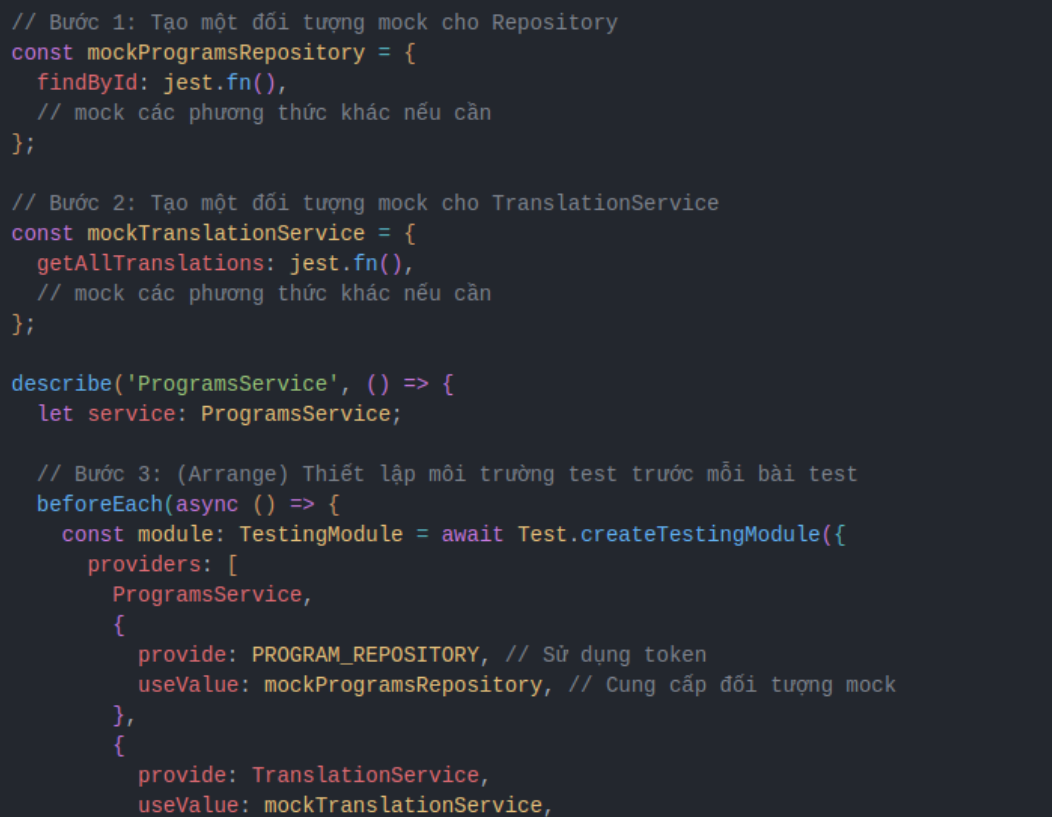
* Kiểm tra dữ liệu chính xác trước khi truy cập service
* Dễ dàng viết test cho schema
* Tái sử dụng schema ở nhiều nơi (client validation, DTO, transformer...)
* Tự động generate schema Swagger (khi dùng **@ZodSerializerDto**)

Phần xác thực dữ liệu này có thể mở rộng tương tự cho các module khác như: **IdentityPaper**, **Address**, **Class**, **Enrollment**, **Result**, bằng cách viết schema riêng cho từng loại dữ liệu và chuyển sang DTO bằng **createZodDto**.

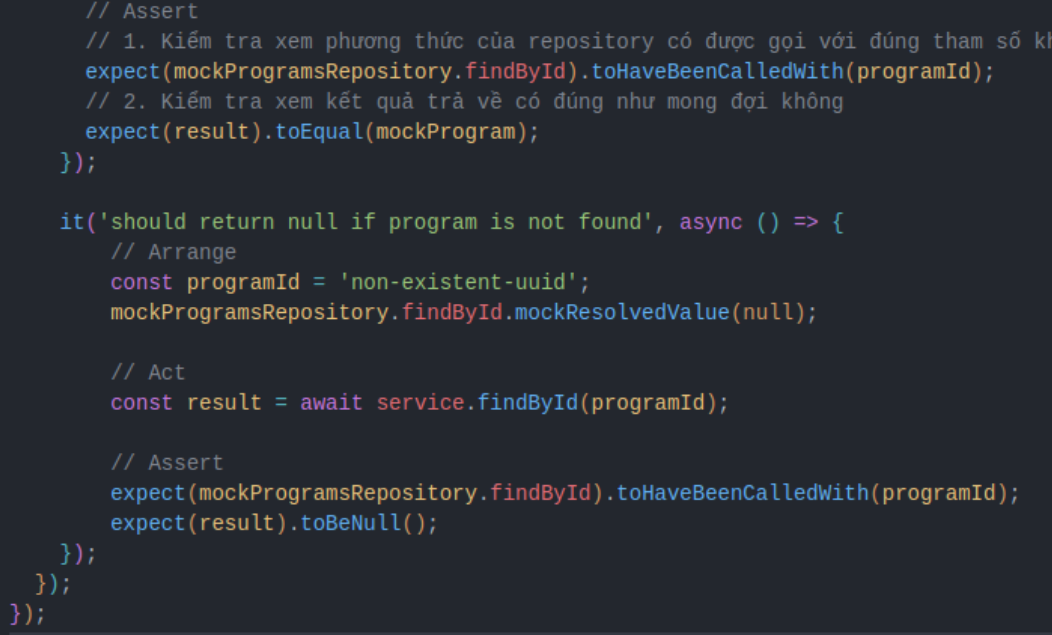
# **Unit Testing**

## **Công cụ và Triết lý**

* + 1. **Framework:** sử dụng Jest, một framework kiểm thử JavaScript phổ biến, kết hợp với thư viện @nestjs/testing để tạo môi trường kiểm thử tích hợp tốt với NestJS.
    2. **Triết lý:** Với Kiến trúc Lục giác, chiến lược kiểm thử của chúng tôi tập trung vào việc:
       - Kiểm thử Logic Nghiệp vụ trong Services: Các lớp Service (ví dụ: ProgramsService) là mục tiêu chính của unit test. Chúng chứa logic nghiệp vụ cốt lõi và phải được kiểm thử kỹ lưỡng trong môi trường biệt lập.
       - Mocking Dependencies: Mọi phụ thuộc bên ngoài của một unit đang được kiểm thử đều phải được "giả lập" (mock). Ví dụ, khi kiểm thử ProgramsService, chúng ta sẽ mock IProgramsRepository và TranslationService. Điều này đảm bảo rằng chúng ta chỉ kiểm thử logic của ProgramsService chứ không phải logic của repository hay CSDL.
    3. **Cấu trúc một bài Test (AAA)**
       - Arrange (Sắp đặt): Chuẩn bị mọi thứ cần thiết cho bài test. Điều này bao gồm việc tạo một testing module, khởi tạo service cần test, và tạo các đối tượng mock cho các phụ thuộc.
       - Act (Hành động): Gọi phương thức mà bạn muốn kiểm thử.
       - Assert (Xác nhận): Kiểm tra xem kết quả của hành động có đúng như mong đợi hay không. Sử dụng các hàm expect của Jest để thực hiện việc này.
    4. **Ví dụ thực tế: Kiểm thử ProgramsService**

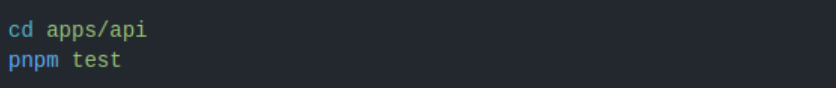
****

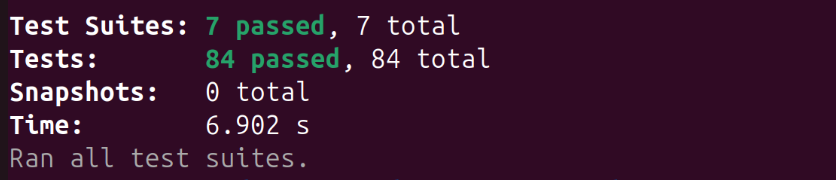
****

****

* + 1. **Chạy Test**

**Chạy Test:** Để thực thi tất cả các unit test trong ứng dụng api, di chuyển vào thư mục và chạy lệnh test được định nghĩa trong package.json.



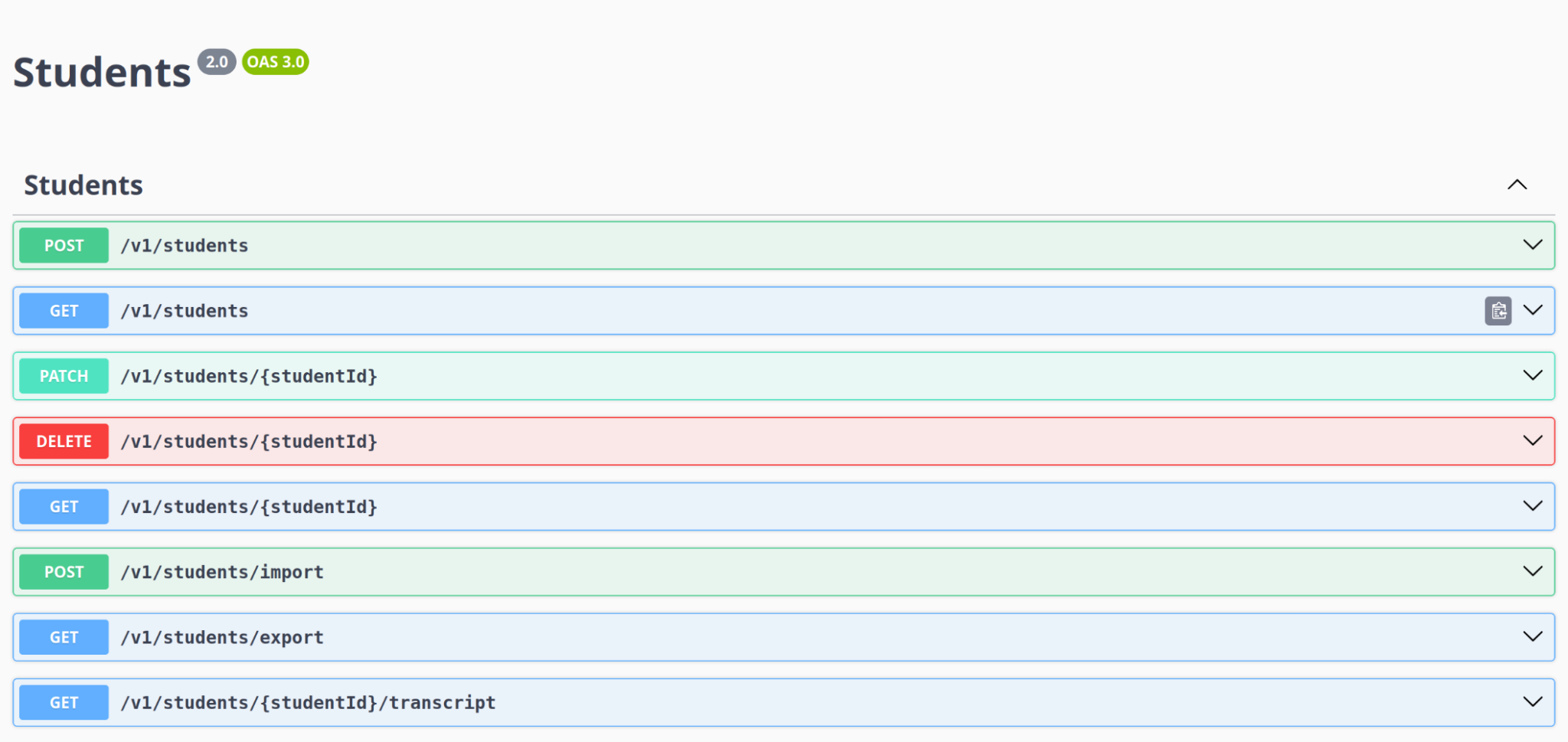


# **Web API documentation**

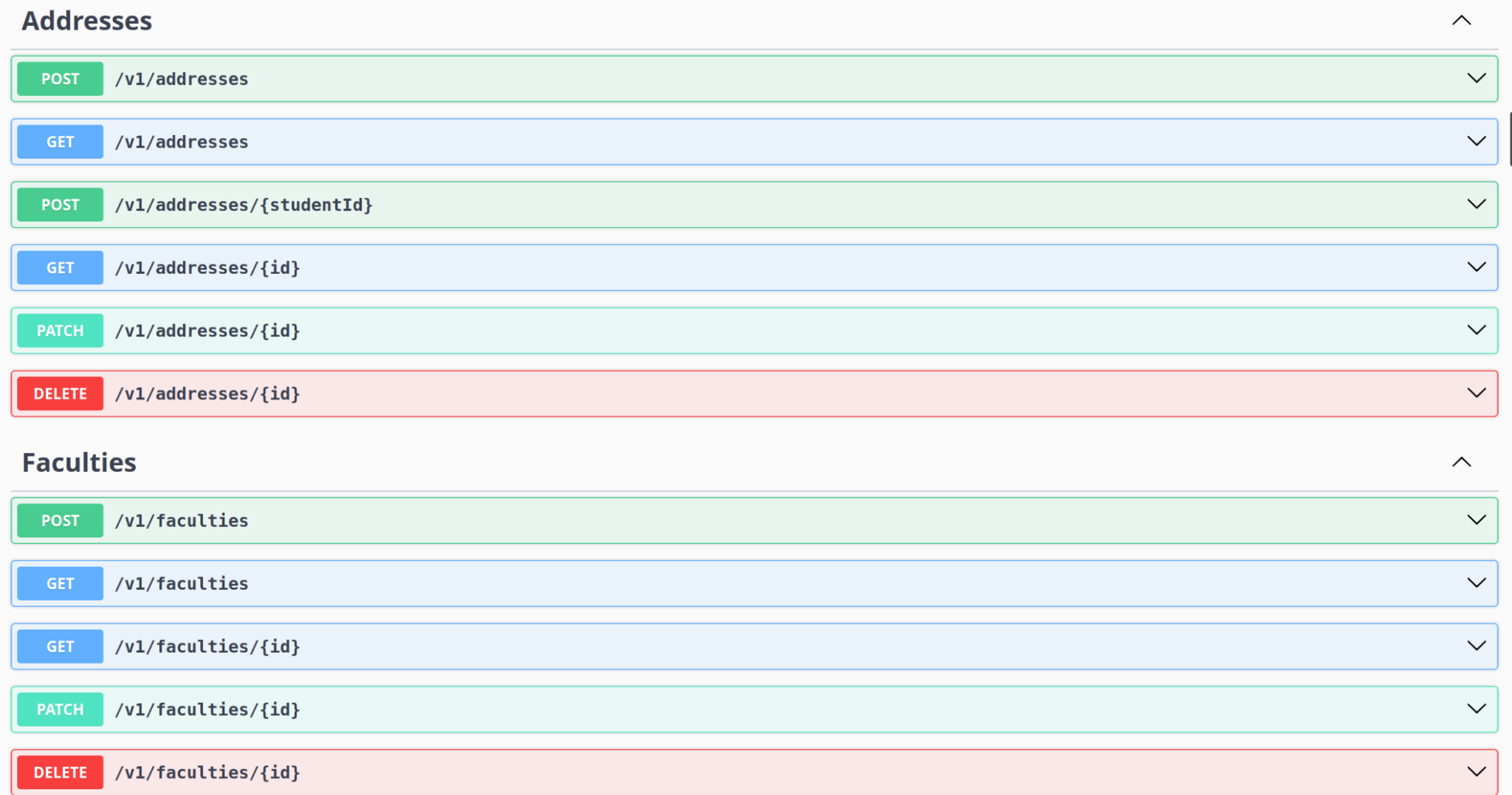
* 1. **Dự án của không sử dụng các tệp tài liệu API tĩnh (static files) như Word, Markdown hay Postman Collection. Thay vào đó, chúng em đã áp dụng phương pháp "Code-First" để tự động tạo ra một tài liệu API tương tác và luôn cập nhật bằng cách sử dụng thư viện @nestjs/swagger. Phương pháp này đảm bảo rằng tài liệu API luôn phản ánh chính xác trạng thái mới nhất của mã nguồn, loại bỏ hoàn toàn rủi ro tài liệu bị lỗi thời.**
  2. **Cơ chế hoạt động dựa trên việc các lập trình viên sử dụng các Decorators của Swagger trực tiếp trong mã nguồn TypeScript để "chú thích" và mô tả các thành phần của API. Controllers và Endpoints: Mỗi endpoint được mô tả bằng các decorators như @ApiTags, @ApiOperation, @ApiResponse. Data Models (DTOs): Mỗi thuộc tính trong các lớp Data Transfer Object (DTO) được mô tả bằng decorator @ApiProperty.**

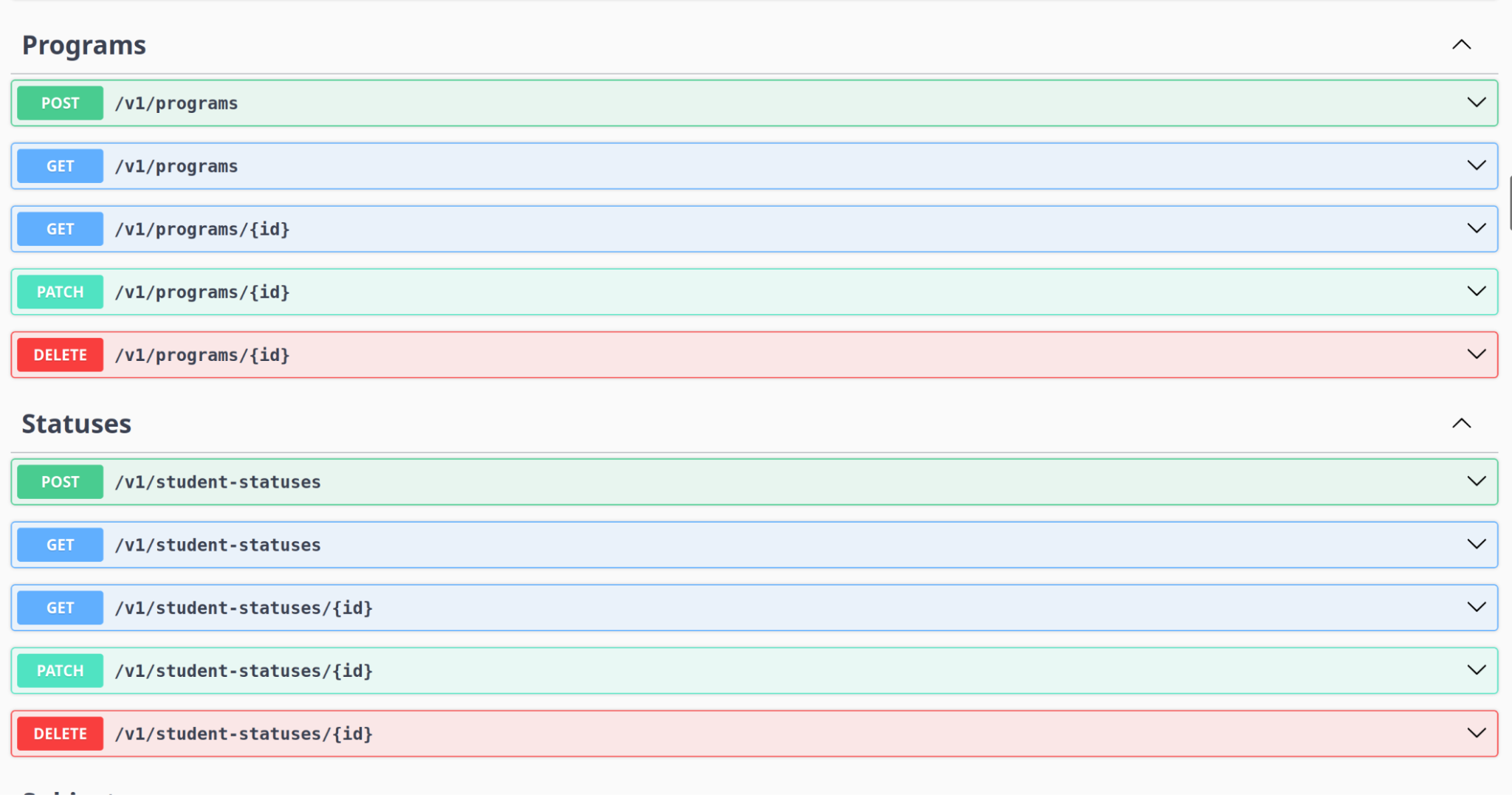


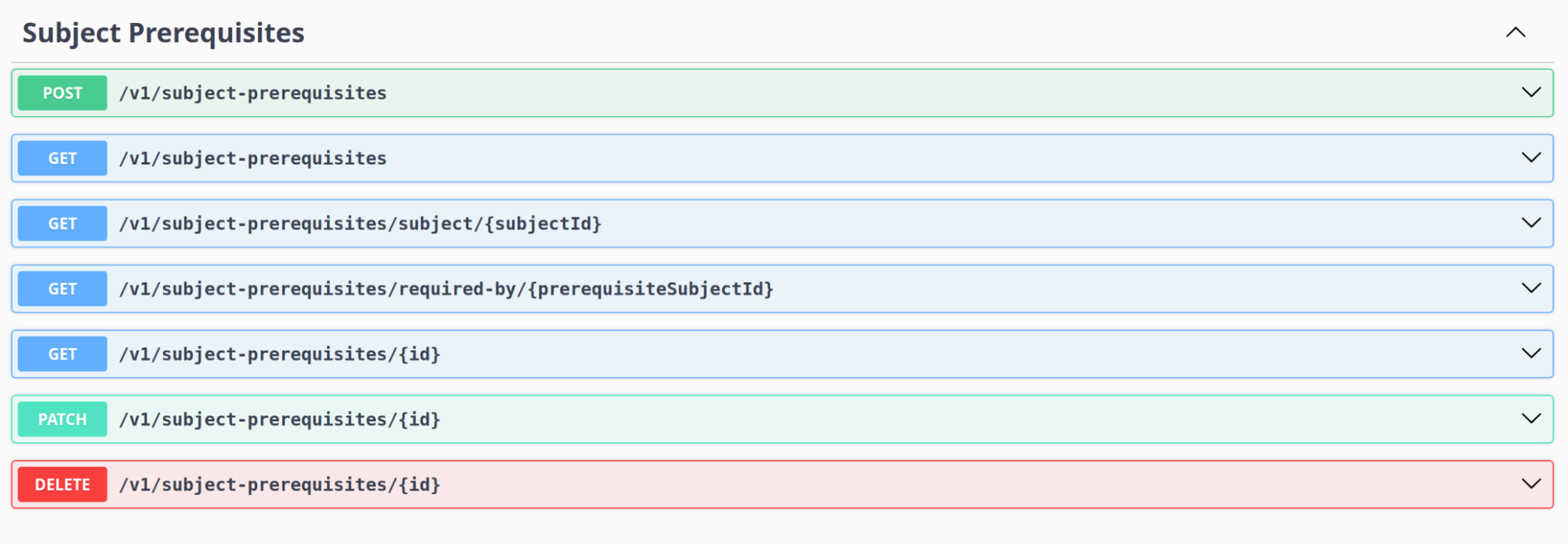
* 1. **Khi ứng dụng backend khởi động, NestJS sẽ quét toàn bộ mã nguồn, đọc các decorator này và tự động xây dựng một trang web tài liệu API tương tác theo chuẩn OpenAPI (Swagger).**
  2. **Hướng dẫn Truy cập Tài liệu API:** 
     1. Pnpm run dev
     2. http://localhost:6050/swagger#/

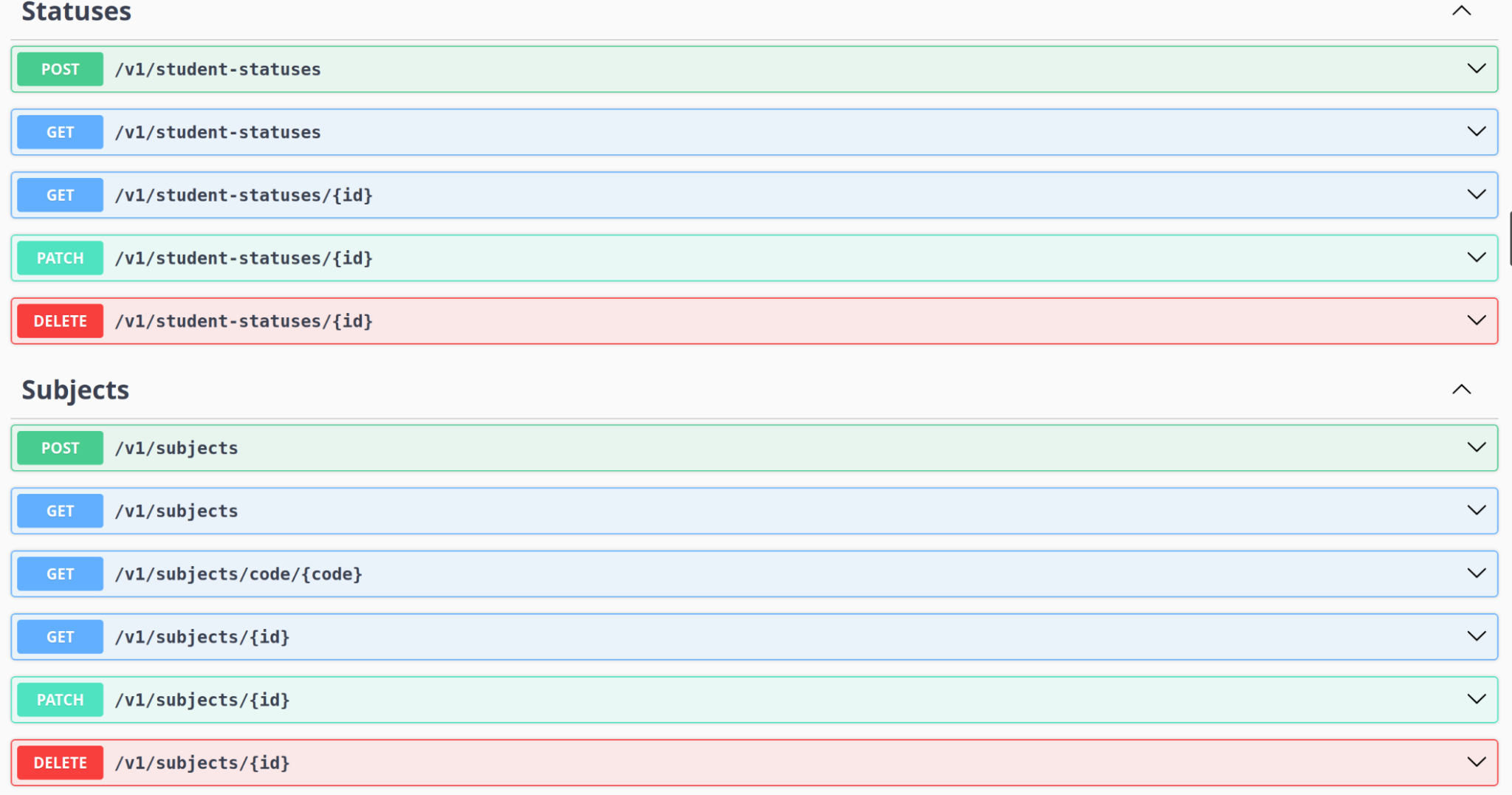






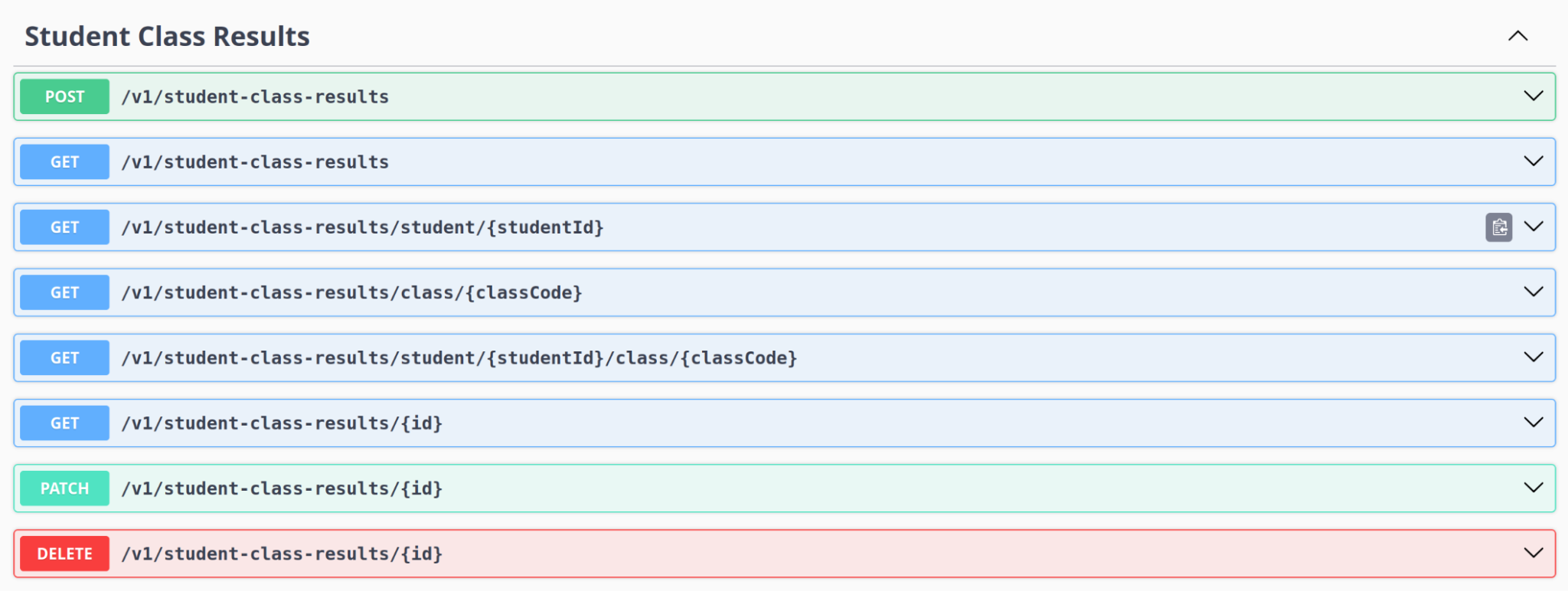


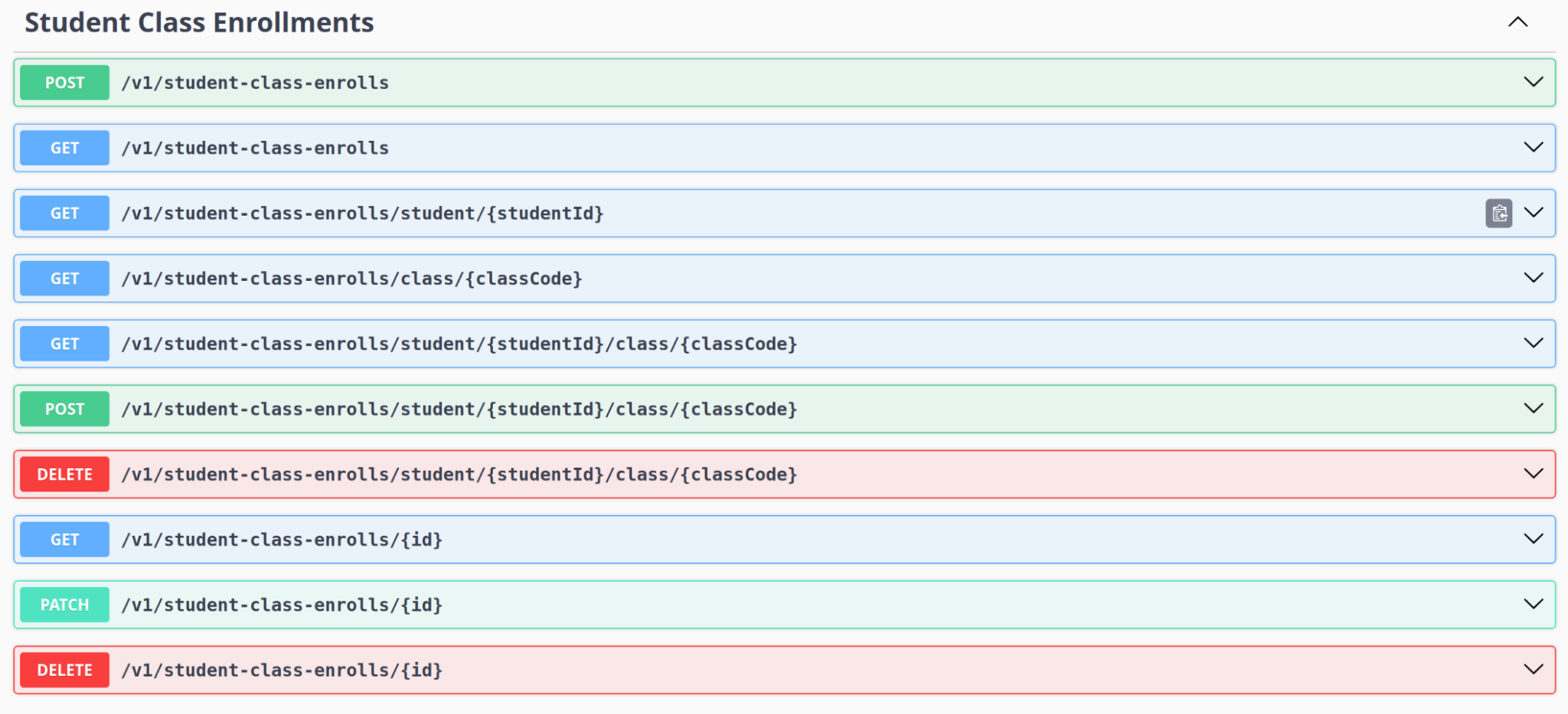












* 1. **Các Tính năng chính của Trang Tài liệu (Swagger UI)**
     1. Danh sách Endpoints: Liệt kê toàn bộ các API của hệ thống, được nhóm một cách logic theo từng tài nguyên (Programs, Faculties, Students...).
     2. Thông tin Chi tiết: Với mỗi API, người xem có thể thấy rõ:
     3. Phương thức HTTP (GET, POST, PATCH, DELETE).
     4. Đường dẫn (URL).
     5. Mô tả chi tiết về chức năng.
     6. Danh sách các tham số yêu cầu (path, query, body).
     7. Mô hình Dữ liệu (Schemas): Hiển thị cấu trúc dữ liệu chính xác của các đối tượng request và response, giúp phía frontend hiểu rõ cần gửi lên và sẽ nhận về những gì.
     8. Tương tác trực tiếp: Tính năng mạnh mẽ nhất là cho phép người dùng thực thi các lời gọi API ngay trên trình duyệt. Người dùng có thể điền các tham số và nhấn "Execute" để gửi request đến server và xem kết quả trả về.
  2. **Ưu điểm của Phương pháp này**
     1. Luôn đồng bộ: Tài liệu được sinh ra từ chính mã nguồn, do đó nó không bao giờ có thể sai lệch hay lỗi thời so với API thực tế.
     2. Nguồn chân lý duy nhất (Single Source of Truth): Lập trình viên chỉ cần tập trung vào việc viết mã và chú thích mã. Không cần phải duy trì một tài liệu riêng biệt.
     3. Tăng hiệu suất: Giảm đáng kể thời gian và công sức cho việc viết và cập nhật tài liệu.
     4. Hỗ trợ hiệu quả cho các nhóm: Team frontend, team kiểm thử (QA) có thể hiểu và bắt đầu làm việc với API một cách nhanh chóng mà không cần hỏi lại team backend.

# **Reference**

* 1. [Hexagonal Architecture](https://medium.com/@amna.azam/decoupling-concerns-applying-hexagonal-architecture-in-nestjs-d59d6199c345)
  2. [NestJs](https://nestjs.com/)
  3. [Zod](https://zod.dev/)
  4. [Prisma](https://www.prisma.io/)
  5. [JestJs](https://jestjs.io/)