TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN, ĐẠI HỌC QUỐC GIA TPHCM

KHOA CÔNG NGHỆ THÔNG TIN

Developer Guide

Môn học**: Thiết kế phần mềm**

Giáo viên phụ trách**: Thầy Trần Duy Thảo**

Sinh viên thực hiện**:**

22120252 – Giang Đức Nhật

22120413 – Nguyễn Quốc Tường

22120441 – Nguyễn Trường Vũ

22120450 – Bùi Đình Gia Vỹ

TP. Hồ Chí Minh, tháng 6 năm 2025

Mục lục

I. Tổng quan về Kiến trúc (Overview of Architecture)

- **Backend:** Hệ thống được xây dựng theo kiến trúc 3 lớp (3-Tier Architecture) rõ ràng để tăng tính module hóa và dễ bảo trì:
    - **Entry Points (API Layer):** Lớp này chứa các file `*.route.ts` và `*.middleware.ts`, chịu trách nhiệm định tuyến các request HTTP và thực hiện validation đầu vào.
    - **Domain (Business Logic Layer):** Lớp này chứa các file `*.controller.ts` và `*.service.ts` (ví dụ: `StudentManager.ts`). Controller nhận request đã được validate và gọi đến Service. Service là nơi chứa toàn bộ logic nghiệp vụ chính của ứng dụng.
    - **Data Access Layer:** Sử dụng Prisma ORM để tương tác với database, trừu tượng hóa các câu lệnh SQL và đảm bảo an toàn kiểu dữ liệu.
- **Frontend:** Kiến trúc frontend được xây dựng dựa trên các nguyên tắc hiện đại của React:
    - **Component-Based:** Giao diện được chia thành các component tái sử dụng (`StudentList`, `StudentItem`, `FacultyComponent`...).
    - **Tách biệt Logic và Giao diện:** Logic gọi API được tách biệt hoàn toàn khỏi component giao diện thông qua các file **Service** (ví dụ: `studentAPIServices.ts`). Component chỉ chịu trách nhiệm hiển thị và gọi đến service.
    - **Quản lý State tập trung:** Sử dụng React Context (`CategoryProvider`) để cung cấp các dữ liệu chung (danh sách khoa, trạng thái, chương trình học) cho toàn bộ ứng dụng, tránh việc truyền props phức tạp và đảm bảo dữ liệu nhất quán.

II. Tổ chức Mã nguồn (Source Code Organization)

- **Backend:** Mã nguồn được tổ chức theo từng "component" chức năng (ví dụ: `student-managing`). Bên trong mỗi component, code được chia theo các lớp kiến trúc đã nêu: `entry-points` (chứa api), `domain` (chứa logic), và `models` (Prisma schema). Cấu trúc này giúp dễ dàng tìm kiếm và phát triển các tính năng liên quan.
- **Frontend:** Mã nguồn được tổ chức theo từng "page" hoặc "feature" (ví dụ: `pages/Student`, `pages/Category`). Bên trong mỗi feature, các thành phần được nhóm lại với nhau. Các service và context dùng chung được đặt trong các thư mục riêng (`services/`, `contexts/`) để dễ dàng tái sử dụng.

III. Áp dụng SOLID và Design Patterns

- **Nguyên lý Single Responsibility (SRP):**
    - **Frontend:** Component `StudentList` chỉ chịu trách nhiệm hiển thị danh sách và quản lý các bộ lọc. `StudentAPIServices` chỉ chịu trách nhiệm gọi API. `StudentItem` chỉ chịu trách nhiệm hiển thị và cập nhật chi tiết một sinh viên.
    - **Backend:** `Controller` chỉ điều hướng request, `Service` (`StudentManager`) chứa logic nghiệp vụ, Prisma `model` chỉ định nghĩa dữ liệu. Mỗi phần chỉ làm một việc duy nhất.
- **Nguyên lý Open/Closed (OCP):**

    - Hệ thống Import/Export sử dụng **Strategy Pattern** (`ImportExportStrategyFactory`). Khi cần thêm một định dạng mới (ví dụ: CSV), ta chỉ cần tạo một class Strategy mới mà không cần sửa đổi code của Factory hay logic import chính. Hệ thống "mở" cho việc mở rộng nhưng "đóng" với việc sửa đổi.
- **Design Pattern khác:**
    - **Factory Pattern:** `ImportExportStrategyFactory` là một ví dụ điển hình của Factory Pattern, giúp tạo ra các đối tượng strategy phù hợp dựa trên yêu cầu.

IV. Cải tiến và Thiết kế Web API

- **RESTful Principles:** Các API được thiết kế theo chuẩn RESTful, sử dụng các phương thức HTTP một cách hợp lý (`GET`, `POST`, `PATCH`/`PUT`, `DELETE`) và cấu trúc URL rõ ràng (ví dụ: `/students`, `/students/:id`).
- **Phương thức `PATCH`:** Đã tối ưu logic `PATCH` ở cả frontend và backend. Frontend giờ đây chỉ gửi những trường dữ liệu thực sự thay đổi, và backend (trong `StudentManager.ts`) được viết lại để có thể xử lý các payload cập nhật một phần này một cách linh hoạt.
- **Nhất quán Response:** Các response từ API (dù chưa hoàn toàn đồng bộ) đã được cấu trúc lại để chứa dữ liệu trong một trường `metadata`, giúp frontend dễ dàng xử lý.

V. Dependency Injection và Clean Architecture

- Mặc dù không sử dụng một framework DI chuyên dụng, nguyên tắc **Dependency Inversion** (chữ D trong SOLID) đã được áp dụng. Ví dụ: `ImportExportStrategyFactory` phụ thuộc vào một abstraction (`ImportExportStrategy`) chứ không phải một implementation cụ thể. Điều này giúp giảm sự phụ thuộc và tăng tính linh hoạt.
- Kiến trúc 3 lớp ở backend là một bước tiến gần đến **Clean Architecture**, nơi các lớp bên ngoài (API) phụ thuộc vào các lớp bên trong (Domain/Service), và logic nghiệp vụ cốt lõi không phụ thuộc vào framework hay database.

VI. Data Validation (ở Backend)

- **Middleware:** Đã xây dựng một chuỗi middleware validation mạnh mẽ cho endpoint `PATCH /students/:id` bằng `express-validator` (thông qua hàm `body()`).
- **Xử lý `PATCH` Request:** Các middleware (`checkEmailPattern`, `checkPhoneNumberPattern`...) đã được sửa lại hoàn toàn để xử lý đúng trường hợp `PATCH` - chúng sẽ bỏ qua validation nếu trường đó không được gửi lên trong request body.
- **Xử lý Lỗi:** Đã sửa lỗi `next()` và `return next()` để ngăn chặn lỗi `ERR_HTTP_HEADERS_SENT`. Logic được bọc trong `try...catch` và các lỗi được chuyển đến error handler một cách nhất quán bằng `next(error)`.
- **Validation Nghiệp vụ:** Middleware `checkStatusTransition` được tạo ra để kiểm tra các quy tắc nghiệp vụ phức tạp (ví dụ: một sinh viên không thể chuyển từ trạng thái "Thôi học" sang "Tốt nghiệp"), tách biệt hoàn toàn logic này ra khỏi controller.

VII. Unit Testing

- Đã rà soát và viết lại file test `studentAPIServices.test.ts` để phản ánh đúng logic mới của ứng dụng.
- **Tách biệt Mock Data:** Phân biệt rõ ràng giữa "dữ liệu thô từ API" (plain object) và "kết quả mong đợi" (instance của class `Student`). Điều này làm cho bài test trở nên chính xác và đáng tin cậy hơn.
- **Kiểm tra các Transformation:** Bài test cho `addStudent` giờ đây đã kiểm tra được rằng phương thức `toJSON()` được gọi, đảm bảo dữ liệu được "làm phẳng" đúng cách trước khi gửi đi.

VIII. Hướng dẫn cho Lập trình viên (Developer Guide)

- **Coding Standards:** Thống nhất sử dụng TypeScript cho cả frontend và backend, đặt tên biến/hàm theo camelCase, tên class theo PascalCase. Sử dụng Prettier/ESLint để tự động format code.
- **Getting Started:** Hướng dẫn clone repo, chạy `npm install` ở cả thư mục `frontend` và `backend`. Hướng dẫn tạo file `.env` để cấu hình chuỗi kết nối database cho Prisma. Chạy `npx prisma migrate dev` để tạo database. Chạy `npm run dev` để khởi động cả hai server.
- **Database Schema:** Toàn bộ schema được quản lý trong file `prisma/schema.prisma`. Đây là "nguồn chân lý" duy nhất cho cấu trúc database. Prisma tự động tạo các migration từ file này.
- **Updating an existing entity:** Hướng dẫn chi tiết cách hàm `update` trong `StudentManager.ts` đã được viết lại để ánh xạ một payload `PATCH` đơn giản (chứa các ID dạng string) sang cấu trúc `connect` hoặc `update` phức tạp của Prisma. Đây là một ví dụ điển hình khi làm việc với ORM.
- **Registering New Routes:** Để thêm một bộ API mới (ví dụ: cho `Module`), cần tạo 4 file: `module.route.ts` (định nghĩa endpoints), `module.controller.ts` (điều hướng), `module.service.ts` (logic nghiệp vụ), và đăng ký router mới trong file `app.ts` (hoặc file server chính).