Kế hoạch di chuyển dự án Node.js sang Deno

Table of Contents

1. Tóm tắt báo cáo

Báo cáo này trình bày kế hoạch chi tiết để di chuyển dự án NestJS hiện tại từ môi trường runtime Node.js sang Deno. Kế hoạch được chia thành các bước chính: Phân tích hiện trạng, Đánh giá tính tương thích, Lập kế hoạch migration và Triển khai chi tiết.

2. Nội dung báo cáo

2.1. Bước 1: Phân tích hiện trạng

2.1.1. 1.1. Kiểm tra và liệt kê tất cả dependencies hiện tại trong `package.json`

Dự án sử dụng các dependencies và devDependencies sau:

- **Dependencies:**
- `@clerk/backend`: Quản lý xác thực người dùng.
- `@elastic/elasticsearch`: Tương tác với Elasticsearch.
- `@nestjs/cache-manager`, `@nestjs/common`, `@nestjs/config`, `@nestjs/core`, `@nestjs/jwt`, `@nestjs/microservices`, `@nestjs/mongoose`, `@nestjs/passport`, `@nestjs/platform-express`, `@nestjs/swagger`, `@nestjs/terminus`, `@nestjs/throttler`, `@nestjs/typeorm`, `@nestjs/websockets`: Các module và framework của NestJS.
- `amqp-connection-manager`, `amqplib`: Tương tác với RabbitMQ.
- `bcryptjs`: Mã hóa mật khẩu.
- `cache-manager`, `cache-manager-redis-yet`, `redis`: Quản lý cache với Redis.
- `class-transformer`, `class-validator`: Chuyển đổi và xác thực dữ liệu.
- `compression`, `helmet`: Middleware bảo mật và tối ưu hóa.
- `mongodb`, `mongoose`: Tương tác với MongoDB.
- `nest-winston`, `winston`: Logging.
- `passport`, `passport-jwt`, `passport-local`: Xác thực.
- `pg`, `typeorm`: Tương tác với PostgreSQL và ORM.
- `raw-body`: Xử lý raw body từ request.
- `resend`: Gửi email.
- `rxjs`: Reactive programming.
- `socket.io`, `@nestjs/websockets`: WebSockets.
- `stripe`: Xử lý thanh toán.
- `svix`: Xử lý webhook.
- `uuid`: Tạo UUID.
- **DevDependencies:**
- `@nestjs/cli`, `@nestjs/schematics`, `@nestjs/testing`: Công cụ phát triển và kiểm thử NestJS.
- `@types/amqplib`, `@types/bcryptjs`, `@types/compression`, `@types/express`, `@types/jest`, `@types/node`, `@types/passport-jwt`, `@types/passport-local`, `@types/supertest`, `@types/uuid`: Type definitions cho TypeScript.
- `@typescript-eslint/eslint-plugin`, `@typescript-eslint/parser`, `eslint`, `eslint-config-prettier`, `eslint-plugin-prettier`: Linting và định dạng code.
- `chalk`: Tạo màu cho console output.
- `jest`, `jest-junit`, `ts-jest`: Framework kiểm thử.

- `prettier`: Định dạng code.
- `source-map-support`: Hỗ trợ source map.
- `supertest`: Kiểm thử HTTP.
- `ts-jest`, `ts-loader`, `ts-node`, `tsconfig-paths`, `typescript`: Công cụ TypeScript.

2.1.2. 1.2. Xác định các tính năng Node.js đang được sử dụng (APIs, modules, built-ins)

Dự án NestJS này sử dụng rộng rãi các tính năng cốt lõi của Node.js và hệ sinh thái npm:

- **Module Built-in:**
- `fs` (File System): Có thể được sử dụng trong các file cấu hình, đọc/ghi file, hoặc quản lý tài nguyên.
- `path`: Để xử lý và chuẩn hóa đường dẫn file.
- `process` (process.env): Truy cập biến môi trường để cấu hình ứng dụng.
- `http`/`https`: NestJS xây dựng trên các module HTTP của Node.js để xử lý các yêu cầu web.
- `stream`: Có thể được sử dụng trong các hoạt động I/O lớn hoặc xử lý dữ liệu theo luồng.
- `buffer`: Xử lý dữ liệu nhị phân, đặc biệt trong các tác vụ mạng hoặc mã hóa.
- `events` (EventEmitter): NestJS sử dụng rộng rãi cơ chế sự kiện.
- `child_process`: Có thể được sử dụng trong các script hoặc công cụ build để thực thi các lệnh hệ thống.
- **APIs và Thư viện phổ biến:**
- `CommonJS modules`: Hầu hết các dependencies hiện tại đều là module CommonJS.
- `npm ecosystem`: Dự án phụ thuộc rất nhiều vào các thư viện từ npm.

2.1.3. 1.3. Đánh giá cấu trúc dự án và các file cấu hình hiện tại

- **Cấu trúc thư mục:**
- `src/`: Chứa mã nguồn chính của ứng dụng NestJS, được tổ chức theo các module (admin, auth, chat, elasticsearch, emails, health, Infrastructure, payments, queues, users, webhooks).
- `src/common/`: Chứa các thành phần dùng chung như filters, interceptors.
- `src/config/`: Chứa các file cấu hình ứng dụng (database, elasticsearch, env, mongodb, redis).
- `src/database/`: Chứa cấu hình database và các migration.
- `test/`: Chứa các bài kiểm thử (unit, component, e2e).
- `doc/`: Chứa tài liệu dự án.
- `memory-bank/`: Chứa các file ghi nhớ của AI.
- **File cấu hình chính:**
- `nest-cli.json`: Cấu hình cho Nest CLI, định nghĩa cách build và chạy ứng dụng NestJS.
- `tsconfig.json`: Cấu hình TypeScript, bao gồm các tùy chọn biên dịch và đường dẫn alias (`paths`).

- `Dockerfile`, `Dockerfile.dev`, `docker-compose.yml`: Cấu hình cho môi trường Docker, định nghĩa cách đóng gói và chạy ứng dụng trong container.
- `index.js`: Có thể là entry point của ứng dụng hoặc một script phụ trợ.
- `package.json`, `package-lock.json`: Quản lý dependencies và định nghĩa các script.

2.1.4. 1.4. Phân tích các scripts và công cụ build đang sử dụng

- **Scripts trong `package.json`:**
- `build`: `nest build` - Sử dụng Nest CLI để biên dịch mã nguồn TypeScript sang JavaScript.
- `start:dev`, `start:debug`, `start:prod`: `nest start` hoặc `node dist/main` - Chạy ứng dụng ở các môi trường khác nhau.
- `lint`, `format`: `eslint`, `prettier` - Sử dụng ESLint và Prettier để kiểm tra và định dạng mã nguồn.
- `test:unit`, `test:watch`, `test:cov`, `test:debug`, `test:e2e`, `test:component`, `test:all`: `jest` - Sử dụng Jest để chạy các loại kiểm thử khác nhau.
- `typeorm`, `migration:*`: `typeorm-ts-node-commonjs` - Sử dụng TypeORM CLI để quản lý database migrations.
- **Công cụ build và phát triển:**
- `Nest CLI`: Công cụ dòng lệnh chính cho các tác vụ phát triển NestJS.
- `TypeScript Compiler (tsc)`: Được sử dụng bởi Nest CLI để biên dịch mã.
- `Jest`: Framework kiểm thử.
- `ESLint`, `Prettier`: Công cụ phân tích và định dạng mã.
- `ts-node`, `tsconfig-paths`: Cho phép chạy các file TypeScript trực tiếp mà không cần biên dịch trước.
- `Docker`: Để container hóa ứng dụng.

2.2. Bước 2: Đánh giá tính tương thích

2.2.1. 2.1. Kiểm tra xem các dependencies có tương thích với Deno không

Đây là một trong những thách thức lớn nhất. Hầu hết các thư viện Node.js (từ npm) không tương thích trực tiếp với Deno vì: * Deno sử dụng ES Modules (`import ⋯ from "⋯"`) thay vì CommonJS (`require()`). * Deno không có `node_modules`. * Deno có các API built-in khác với Node.js (ví dụ: `Deno.readTextFile` thay vì `fs.readFile`). * Mô hình bảo mật của Deno yêu cầu quyền truy cập rõ ràng.

Các thư viện NestJS (`@nestjs/*`) sẽ không tương thích trực tiếp. NestJS được xây dựng chặt chẽ trên Node.js và Express. Việc di chuyển NestJS sang Deno sẽ đòi hỏi một nỗ lực lớn, có thể là viết lại một phần đáng kể hoặc tìm kiếm các framework tương đương trong Deno.

Các thư viện khác như `typeorm`, `mongoose`, `redis`, `stripe`, `resend`, `svix`, `amqplib`, `bcryptjs`, `class-validator`, `class-transformer`, `uuid` cần được kiểm tra từng cái một. Một số có thể có phiên bản tương thích với Deno hoặc có thể được thay thế bằng các thư viện Deno native.

2.2.2. 2.2. Xác định các thay thế cho dependencies không tương thích

- **Framework Web:** NestJS không tương thích. Cần tìm một framework web tương tự trong Deno, ví dụ:
- https://deno.land/x/oak (tương tự Koa)
- https://deno.land/x/hono (nhẹ và nhanh)
- Hoặc xây dựng API trực tiếp bằng Deno's native HTTP server.
- **ORM/ODM:**
- `TypeORM`: Có thể có phiên bản tương thích Deno hoặc cần thay thế bằng một ORM/ODM khác cho Deno (ví dụ: `deno_mongo` cho MongoDB, hoặc các thư viện PostgreSQL client native Deno).
- `Mongoose`: Không tương thích. Cần sử dụng `deno_mongo` hoặc một thư viện MongoDB client native Deno khác.
- **Xác thực:** `@clerk/backend`, `passport-jwt`, `bcryptjs` - cần tìm các giải pháp xác thực tương đương hoặc tích hợp trực tiếp với Deno. `bcryptjs` có thể có phiên bản tương thích hoặc có thể sử dụng Web Crypto API của Deno.
- **Cache:** `cache-manager`, `redis` - cần tìm các thư viện Redis client tương thích Deno.
- **Message Queue:** `amqplib` - cần tìm thư viện AMQP client tương thích Deno.
- **Email:** `resend` - cần kiểm tra xem có phiên bản Deno SDK không.
- **Payments:** `stripe` - cần kiểm tra xem có phiên bản Deno SDK không.
- **WebSockets:** `socket.io` - cần tìm thư viện WebSocket tương thích Deno hoặc sử dụng WebSocket API native của Deno.
- **Logging:** `winston` - cần tìm thư viện logging tương thích Deno.
- **Validation/Transformation:** `class-validator`, `class-transformer` - cần tìm các thư viện tương đương trong Deno.
- **UUID:** `uuid` - có thể có phiên bản tương thích Deno hoặc sử dụng Web Crypto API.
- **Testing:** `Jest` không tương thích. Deno có built-in test runner. Cần viết lại các bài kiểm thử.
- **Linting/Formatting:** `ESLint`, `Prettier` - Deno có `deno fmt` và `deno lint` built-in.

2.2.3. 2.3. Liệt kê các thay đổi cần thiết trong code để tương thích với Deno

- **Import Statements:** Thay đổi tất cả các `require()` thành `import ⋯ from "⋯"` và cập nhật đường dẫn import để sử dụng URL hoặc đường dẫn tương đối/tuyệt đối của Deno.
- **Node.js Built-in APIs:** Thay thế các API như `fs`, `path`, `process`, `http`, `stream`, `buffer`, `events` bằng các API tương ứng của Deno (ví dụ: `Deno.readTextFile`, `Deno.env`, `Deno.serve`).
- **Biến môi trường:** Deno sử dụng `Deno.env.get()` thay vì `process.env`.
- **Cấu hình:** Cập nhật các file cấu hình để phù hợp với cách Deno quản lý module và biến môi trường.
- **Kiểm thử:** Viết lại các bài kiểm thử để sử dụng Deno's built-in test runner.
- **TypeScript:** Deno hỗ trợ TypeScript native, nên không cần `tsconfig.json` hay `ts-node` nữa. Tuy nhiên, cần đảm bảo code tuân thủ các quy tắc của Deno.

- **Mô hình bảo mật:** Thêm các cờ quyền (`--allow-net`, `--allow-read`, `--allow-write`, `--allow-env`, v.v.) khi chạy ứng dụng Deno.

2.2.4. 2.4. Đánh giá các tính năng Deno có thể tận dụng

- **TypeScript Native:** Không cần biên dịch trước, giúp tăng tốc độ phát triển.
- **Security Model:** Mô hình bảo mật dựa trên quyền truy cập giúp tăng cường bảo mật cho ứng dụng.
- **Built-in Tools:** `deno fmt`, `deno lint`, `deno test`, `deno doc`, `deno bundle` giúp đơn giản hóa quy trình phát triển và loại bỏ nhiều devDependencies.
- **Single Executable:** Khả năng đóng gói ứng dụng thành một file thực thi duy nhất.
- **Web Standard APIs:** Sử dụng các API chuẩn web (Fetch API, Web Crypto API, URL API) giúp code dễ di chuyển hơn giữa các môi trường.

2.3. Bước 3: Lập kế hoạch migration

2.3.1. 3.1. Chia nhỏ quá trình migration thành các phase cụ thể

- **Phase 0: Chuẩn bị và Nghiên cứu**
- Đọc và hiểu sâu về Deno, các API và best practices của nó.
- Xác định các thư viện Deno thay thế cho tất cả các dependencies Node.js hiện tại.
- Thiết lập môi trường phát triển Deno cơ bản.
- Tạo một dự án Deno nhỏ để thử nghiệm các khái niệm cơ bản.
- **Phase 1: Chuyển đổi Core Services và Cấu hình**
- Tạo cấu trúc dự án Deno mới.
- Chuyển đổi các file cấu hình (`.env`, database configs) sang định dạng tương thích Deno.
- Thay thế các Node.js built-in APIs bằng Deno APIs trong các module cốt lõi (ví dụ: quản lý biến môi trường, I/O cơ bản).
- Thiết lập một HTTP server cơ bản bằng Deno (ví dụ: Oak hoặc Hono).
- **Phase 2: Di chuyển Database và ORM/ODM**
- Thay thế TypeORM/Mongoose bằng các thư viện Deno native cho PostgreSQL và MongoDB.
- Chuyển đổi các entity/schema và repository/model sang định dạng mới.
- Đảm bảo các migration hoạt động với Deno.
- **Phase 3: Di chuyển các Module và Dịch vụ khác**
- Di chuyển các module xác thực (Clerk, Passport) sang giải pháp Deno tương đương.
- Chuyển đổi các module cache (Redis).
- Di chuyển các module message queue (RabbitMQ).
- Chuyển đổi các module thanh toán (Stripe), email (Resend), webhook (Svix).
- Di chuyển các module Elasticsearch, WebSockets.

- **Phase 4: Kiểm thử và Tối ưu hóa**
- Viết lại tất cả các bài kiểm thử để sử dụng Deno's built-in test runner.
- Thực hiện kiểm thử đơn vị, tích hợp và E2E.
- Tối ưu hóa hiệu suất và tài nguyên cho môi trường Deno.
- Cập nhật Dockerfile để sử dụng Deno image.

2.3.2. 3.2. Xác định thứ tự ưu tiên cho từng component/module

1. **Cấu hình và Môi trường:** Đảm bảo ứng dụng có thể đọc cấu hình và biến môi trường trong Deno.
2. **HTTP Server cơ bản:** Thiết lập một điểm cuối API đơn giản để xác nhận Deno có thể xử lý request.
3. **Database Connectivity:** Kết nối thành công với PostgreSQL và MongoDB.
4. **Xác thực:** Đây là một module quan trọng, cần được di chuyển sớm để các module khác có thể sử dụng.
5. **Các module độc lập:** Các module ít phụ thuộc vào các module khác (ví dụ: Emails, UUID) có thể được di chuyển sớm.
6. **Các module phức tạp/phụ thuộc nhiều:** Các module như Chat, Payments, Webhooks, Elasticsearch cần được di chuyển sau khi các dependencies của chúng đã ổn định.
7. **Kiểm thử:** Viết lại kiểm thử song song với quá trình di chuyển để đảm bảo chất lượng.

2.3.3. 3.3. Lên kế hoạch testing và validation cho từng bước

- **Kiểm thử đơn vị (Unit Tests):** Sau khi di chuyển từng module, viết lại và chạy các bài kiểm thử đơn vị tương ứng bằng Deno's built-in test runner.
- **Kiểm thử tích hợp (Integration Tests):** Sau khi di chuyển các nhóm module liên quan (ví dụ: Auth + Users), chạy kiểm thử tích hợp để đảm bảo chúng hoạt động cùng nhau.
- **Kiểm thử E2E (End-to-End Tests):** Sau mỗi phase lớn, chạy các bài kiểm thử E2E để xác nhận luồng người dùng chính hoạt động đúng.
- **Kiểm thử hiệu suất (Performance Tests):** So sánh hiệu suất của ứng dụng Deno với phiên bản Node.js.
- **Kiểm thử bảo mật (Security Tests):** Đảm bảo mô hình bảo mật của Deno được áp dụng đúng và không có lỗ hổng mới.
- **Code Review:** Thực hiện code review thường xuyên để đảm bảo code tuân thủ các best practices của Deno.

2.3.4. 3.4. Chuẩn bị rollback plan nếu cần thiết

- **Version Control:** Sử dụng Git để quản lý phiên bản. Tạo các branch riêng cho quá trình migration.
- **Incremental Commits:** Thực hiện các commit nhỏ, thường xuyên sau mỗi thay đổi thành công.

- **Feature Flags:** Nếu có thể, sử dụng feature flags để bật/tắt các tính năng đã di chuyển, cho phép triển khai dần dần.
- **Backup Database:** Sao lưu toàn bộ database trước khi thực hiện các migration lớn.
- **Song song môi trường:** Duy trì môi trường Node.js hiện tại hoạt động song song với môi trường Deno mới trong quá trình chuyển đổi.

2.4. Bước 4: Triển khai chi tiết

2.4.1. 4.1. Tạo cấu trúc dự án mới cho Deno

- Tạo một thư mục gốc mới cho dự án Deno (ví dụ: `theshoebolt-deno`).
- Di chuyển các file mã nguồn (`.ts`) từ `src/` sang cấu trúc mới, có thể điều chỉnh lại để phù hợp với Deno's conventions.
- Tạo file `deps.ts` để quản lý các dependencies bên ngoài (tương tự `package.json` nhưng cho Deno).
- Tạo file `main.ts` hoặc `app.ts` làm entry point chính của ứng dụng Deno.
- Tạo file `deno.json` để cấu hình Deno (tasks, lint, fmt, test).

2.4.2. 4.2. Cập nhật import/export statements

- Thay đổi tất cả các `require()` thành `import ⋯ from "⋯"`.
- Cập nhật đường dẫn import:
- Đối với các module cục bộ: Sử dụng đường dẫn tương đối (ví dụ: `import { User } from "./users/user.entity.ts";`).
- Đối với các module từ Deno.land/x: Sử dụng URL đầy đủ (ví dụ: `import { Application } from "https://deno.land/x/oak/mod.ts";`).
- Đối với các module Node.js tương thích Deno: Sử dụng tiền tố `node:` (ví dụ: `import { Buffer } from "node:buffer";`).
- Đảm bảo tất cả các file TypeScript đều có phần mở rộng `.ts` trong import statements.

2.4.3. 4.3. Thay thế Node.js APIs bằng Deno APIs tương ứng

- **File System:**
- `fs.readFile` → `Deno.readTextFile`
- `fs.writeFile` → `Deno.writeTextFile`
- `fs.existsSync` → `Deno.stat` (kiểm tra lỗi)
- **Environment Variables:**
- `process.env.VAR_NAME` → `Deno.env.get("VAR_NAME")`
- **HTTP Server:**
- Sử dụng `Deno.serve` hoặc một framework như Oak/Hono thay vì Express/NestJS.
- **Path:**

- Sử dụng `path` module từ `deno.land/std/path` hoặc các hàm tương ứng của Deno.
- **Buffer:**
- Sử dụng `Uint8Array` hoặc `Buffer` từ `node:buffer` nếu cần.
- **Events:**
- Sử dụng `EventTarget` hoặc các cơ chế sự kiện của framework Deno.

2.4.4. 4.4. Cập nhật scripts và công cụ development

- **Scripts trong `deno.json`:**
- Thay thế `nest build` bằng `deno compile` hoặc `deno run` trực tiếp.
- Thay thế `jest` bằng `deno test`.
- Thay thế `eslint`, `prettier` bằng `deno lint`, `deno fmt`.
- Cập nhật các script migration của TypeORM hoặc thay thế bằng công cụ migration Deno native.
- **Dockerfile:**
- Sử dụng Deno base image (ví dụ: `denoland/deno:latest`).
- Cập nhật các lệnh để chạy ứng dụng Deno.
- **CI/CD:** Cập nhật các pipeline CI/CD để sử dụng các lệnh Deno thay vì Node.js/npm.

3. Kết Luận

Quá trình di chuyển từ Node.js sang Deno là một nỗ lực đáng kể, đặc biệt đối với một dự án NestJS phức tạp với nhiều dependencies. Kế hoạch này cung cấp một lộ trình chi tiết, chia nhỏ quá trình thành các bước quản lý được, từ phân tích hiện trạng đến triển khai chi tiết và kiểm thử. Việc tuân thủ kế hoạch này và thực hiện kiểm thử kỹ lưỡng ở mỗi giai đoạn sẽ giúp giảm thiểu rủi ro và đảm bảo quá trình chuyển đổi thành công.