KHOA CÔNG NGHỆ THÔNG TIN
TRƯỜNG ĐẠI HỌC KHOA HỌC TỰ NHIÊN
227 Nguyễn Văn Cừ, Phường 4, Quận 5, TP.HCM
Điện Thoại: (08) 38.354.266 - Fax:(08) 38.350.096

Kiến trúc phần mềm
VOU - Marketing with Real-time Games

VOU - Marketing with Real-time Games	Phiên bản: 1.0
Kiến trúc phần mềm	Ngày: 06.09.2024

Tự đánh giá

Nội dung	**Điểm**
Phần báo cáo kiến trúc(thang 10):	10
Phần ứng dụng(thang 10):	10
Phần nâng cao(cộng điểm thêm từ 0->4 điểm):	4

Sinh viên thực hiện:

Phần tự đánh giá % từng thành viên:

MSSV	**Họ Và Tên**	**% đánh giá**
21127158	Lê Hoàng Sang	100%
21127297	Đỗ Phạm Thanh Huy	100%
21127223	Nguyễn Tiến Bách	100%
21127236	Vũ Đình Chương	100%
21127511	Nguyễn Quốc Huy	100%

VOU - Marketing with Real-time Games	Phiên bản: 1.0
Kiến trúc phần mềm	Ngày: 06.09.2024

Mục lục

VOU - Marketing with Real-time Games	Phiên bản: 1.0
Kiến trúc phần mềm	Ngày: 06.09.2024

1. Cơ sở dữ liệu

1. Bảng User (Người dùng)

- **Mô tả**: Bảng này lưu trữ thông tin người dùng trong hệ thống.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - name: Tên người dùng.
  - userName: Tên người dùng (username) cho hệ thống.
  - phoneNumber: Số điện thoại, duy nhất cho mỗi người dùng.
  - hashedPassword: Mật khẩu đã được mã hóa.
  - email: Địa chỉ email, duy nhất.
  - avatar: URL ảnh đại diện.
  - role: Vai trò của người dùng (admin, staff, player).
  - firebaseUID: UID liên kết với Firebase.
  - Các thông tin khác liên quan đến bảo mật như OTP, hashedRefreshToken, passwordResetToken, v.v.
  - **Mối quan hệ**:
    - gameId: Liên kết với bảng Game (mỗi người dùng có thể tham gia một game cụ thể).
    - inventoryId: Liên kết với bảng Inventory (mỗi người dùng có một kho đồ).
    - Liên kết với bảng Transaction (quản lý các giao dịch vật phẩm giữa người dùng).

2. Bảng Event (Sự kiện)

- **Mô tả**: Lưu trữ thông tin về các sự kiện diễn ra trong hệ thống.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - name: Tên sự kiện.
  - description: Mô tả sự kiện.
  - imageUrl: Ảnh đại diện sự kiện.
  - beginAt, endAt: Thời gian bắt đầu và kết thúc sự kiện.
  - status: Trạng thái của sự kiện (ACTIVE, INACTIVE).
  - **Mối quan hệ**:
    - Liên kết với bảng Post (sự kiện có thể có nhiều bài viết liên quan).
    - Liên kết với bảng Favourite (sự kiện có thể được người dùng đánh dấu yêu thích).

3. Bảng Post (Bài viết)

- **Mô tả**: Lưu trữ các bài viết liên quan đến sự kiện.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - title: Tiêu đề bài viết.
  - content: Nội dung bài viết.
  - imageUrl: Ảnh bài viết.
  - status: Trạng thái bài viết (PUBLISHED, DRAFT, ARCHIVED).
  - **Mối quan hệ**:
    - eventId: Liên kết với bảng Event (mỗi bài viết thuộc về một sự kiện).

4. Bảng Favourite (Yêu thích)

- **Mô tả**: Quản lý danh sách sự kiện mà người dùng yêu thích.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - eventId: Liên kết với bảng Event.
  - userId: Liên kết với bảng User.

5. Bảng NotificationKey (Khóa thông báo)

- **Mô tả**: Lưu trữ mã thông báo để gửi thông báo đẩy cho người dùng.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - userId: Liên kết với bảng User.
  - expoPushToken: Mã thông báo của người dùng.

6. Bảng NotificationRecord (Ghi nhận thông báo)

- **Mô tả**: Lưu trữ các thông báo đã gửi cho người dùng.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - userId: Liên kết với bảng User.
  - title, body: Tiêu đề và nội dung của thông báo.

7. Bảng Game (Trò chơi)

- **Mô tả**: Lưu trữ thông tin về các trò chơi trong hệ thống.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - name: Tên trò chơi.
  - lives: Số lượt chơi mặc định của người chơi.
  - company: Tên công ty tổ chức trò chơi.

- **Mối quan hệ**:
  - Liên kết với bảng Item (một game có nhiều vật phẩm).
  - Liên kết với bảng User (một người chơi có thể tham gia một game).

8. Bảng Item (Vật phẩm)

- **Mô tả**: Lưu trữ thông tin về các vật phẩm trong game.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - name: Tên vật phẩm.
  - image: Ảnh đại diện của vật phẩm.
  - description: Mô tả vật phẩm.
  - **Mối quan hệ**:
    - Liên kết với bảng InventoryItem (một vật phẩm có thể có trong nhiều kho đồ).
    - Liên kết với bảng Transaction (vật phẩm có thể được giao dịch giữa người dùng).

9. Bảng Inventory (Kho đồ)

- **Mô tả**: Lưu trữ danh sách vật phẩm mà người dùng sở hữu.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - **Mối quan hệ**:
    - Liên kết với bảng User (một người dùng có một kho đồ).
    - Liên kết với bảng InventoryItem (một kho đồ chứa nhiều vật phẩm).

10. Bảng InventoryItem (Vật phẩm trong kho)

- **Mô tả**: Lưu trữ thông tin chi tiết về các vật phẩm mà người dùng sở hữu trong kho đồ.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - itemId: Liên kết với bảng Item (mỗi vật phẩm trong kho có liên kết với bảng vật phẩm).
  - inventoryId: Liên kết với bảng Inventory (mỗi vật phẩm thuộc về một kho đồ).

11. Bảng Transaction (Giao dịch)

- **Mô tả**: Quản lý các giao dịch vật phẩm giữa người dùng.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - inventoryItemId: Liên kết với bảng InventoryItem.

- senderId, receiverId: Liên kết với bảng User (người gửi và người nhận vật phẩm).

12. Bảng Request (Yêu cầu giao dịch)

- **Mô tả**: Lưu trữ các yêu cầu giao dịch vật phẩm giữa người dùng.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - senderId, receiverId: Liên kết với bảng User (người gửi và người nhận yêu cầu).

13. Bảng QuizGame (Trò chơi câu đố)

- **Mô tả**: Quản lý thông tin về các trò chơi câu đố.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - gameName: Tên trò chơi.
  - playerQuantity: Số lượng người chơi tham gia.

14. Bảng QuizGameQuestion (Câu hỏi trò chơi câu đố)

- **Mô tả**: Lưu trữ các câu hỏi trong trò chơi câu đố.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - content: Nội dung câu hỏi.
  - **Mối quan hệ**:
    - Liên kết với bảng QuizGame (mỗi câu hỏi có thể thuộc nhiều trò chơi).

15. Bảng QuizGameAnswer (Câu trả lời cho câu hỏi trò chơi)

- **Mô tả**: Lưu trữ các câu trả lời cho câu hỏi.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - content: Nội dung câu trả lời.
  - **Mối quan hệ**:
    - Liên kết với bảng QuizGameQuestion (mỗi câu trả lời thuộc về một câu hỏi).

16. Bảng Voucher (Phiếu quà tặng)

- **Mô tả**: Quản lý các phiếu quà tặng.
- **Các trường**:

- id: Khóa chính tự động tăng.
- name: Tên phiếu quà tặng.
- value: Giá trị của phiếu.
- status: Trạng thái phiếu (VALID, INVALID, USED).

17. Bảng voucherLine (Dòng phiếu quà tặng)

- **Mô tả**: Lưu trữ thông tin về phiếu quà tặng của người dùng.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - voucherId: Liên kết với bảng Voucher.
  - userId: Liên kết với bảng User.

18. Bảng voucherTrans (Giao dịch phiếu quà tặng)

- **Mô tả**: Quản lý các giao dịch chuyển phiếu quà tặng giữa người dùng.
- **Các trường**:
  - id: Khóa chính tự động tăng.
  - from_user, to_user: Liên kết với bảng User (người gửi và người nhận phiếu quà tặng).

2. Kiến trúc hệ thống

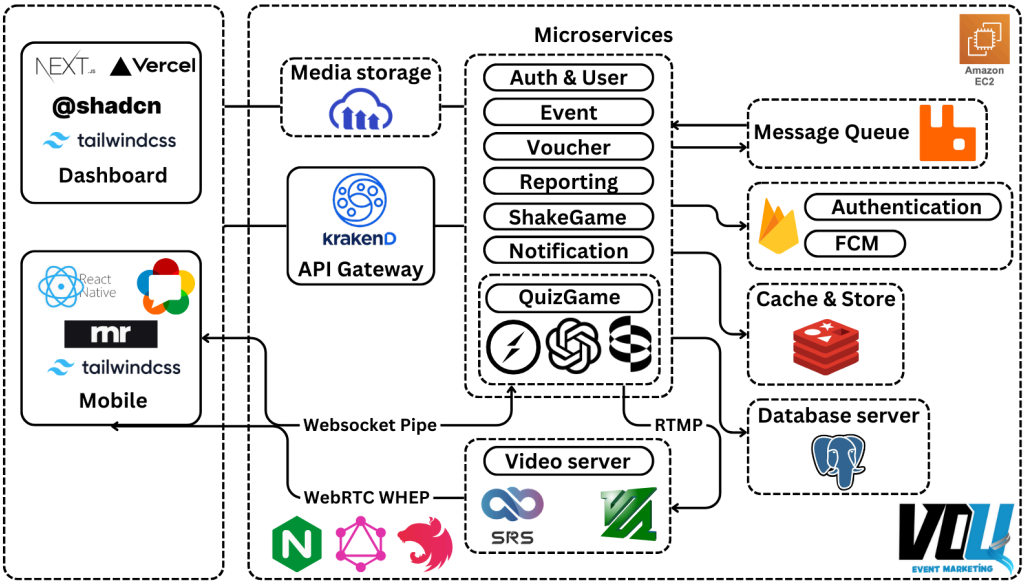
A. Mô tả kiến trúc tổng quan:

1. Công nghệ sử dụng

- **Deployment:** AWS EC2, DigitalOcean Droplets, Vercel
- **Database:** PostgreSQL
- **Back-end:** NestJS, RabbitMQ, Apollo GraphQL server, Prisma, Firebase Authentication, Redis, Socket.IO, KrakenD, OSSRS
- **Front-end:** NextJS, Zustand, React-Query, TailwindCSS, ShadcnUI, Apollo Client
- **CDN:** Cloudinary

2. Sơ đồ kiến trúc:

VOU - Marketing with Real-time Games	Phiên bản: 1.0
Kiến trúc phần mềm	Ngày: 06.09.2024



B. Mô tả các thành phần

1. Client

- **Web dashboard (Next.js + Vercel)**:
  - **Vai trò**: Cung cấp giao diện người dùng (UI) cho các tương tác trên nền tảng web. WebApp được xây dựng bằng **Next.js**, một framework mạnh mẽ dựa trên ReactJS để phát triển các ứng dụng server-side rendering.
  - **Vercel**: Được sử dụng để triển khai và lưu trữ ứng dụng web, giúp dễ dàng quản lý việc phát triển, triển khai và scale.
  - **TailwindCSS**: Được sử dụng để thiết kế giao diện theo phong cách utility-first CSS framework, giúp tăng tốc độ phát triển giao diện người dùng khi kết hợp với **ShadcnUI**.
- **Mobile (React Native + Expo)**:
  - **Vai trò**: Cung cấp giao diện người dùng cho các thiết bị di động. Ứng dụng được xây dựng bằng **React Native**, giúp phát triển đa nền tảng (iOS và Android) từ cùng một codebase.

- **TailwindCSS**: Cũng được tích hợp để tối ưu hóa việc phát triển giao diện trên mobile.

2. API Gateway (KrakenD)

- **Vai trò**:
  - API Gateway đóng vai trò như lớp trung gian giữa client và các microservices phía sau. Nó tổng hợp các API từ nhiều microservice khác nhau và cung cấp cho client một điểm truy cập duy nhất.
  - **KrakenD** là API Gateway sử dụng trong hệ thống, giúp tăng hiệu suất, bảo mật, và khả năng mở rộng bằng cách quản lý tất cả các request từ frontend.
- **Tính năng quan trọng**:
  - Tích hợp, tối ưu hóa lưu lượng, và giúp giảm thiểu gánh nặng cho client bằng cách giảm số lượng request.

3. Microservices

- **Auth & User Service**:
  - Quản lý xác thực và người dùng. Đảm bảo rằng chỉ có người dùng đã được xác thực mới có thể truy cập các dịch vụ khác.
- **Event Service**:
  - Quản lý các sự kiện và sự kiện phát sóng trong hệ thống. Có thể là các sự kiện từ game hoặc voucher.
- **Voucher Service**:
  - Quản lý việc phát hành và sử dụng voucher trong hệ thống.
- **Reporting Service**:
  - Cung cấp các thống kê và báo cáo cho hệ thống, tập hợp dữ liệu từ các microservice khác (event, game, voucher, user).
- **Notification Service**
  - Cung cấp các dịch vụ liên quan đến việc gửi thông báo xuống thiết bị di động của người dùng
- **Game Service**:
  - **Shake Game Service**: Quản lý các trò chơi shake, cung cấp cơ chế chơi và logic cho người dùng.
  - **Quiz Game Service**: Tương tự, quản lý các trò chơi quiz, với hỗ trợ WebSockets cho giao tiếp thời gian thực.

4. Video server

- **Vai trò**:
  - Tiếp nhận & xử lý các yêu cầu tạo luồng stream. Sử dụng giao thức WebRTC WHEP có độ trễ thấp để phân phối stream, đảm bảo được độ đồng bộ giữa video và tiến trình thực tế của trò chơi đang được diễn ra.

5. API

- **GraphQL**:
  - Cho phép giao tiếp hiệu quả giữa các microservices và client, đặc biệt với các schema dữ liệu lồng nhau theo nhiều cấp. Tạo sự dễ dàng mở rộng trong tương lai.

6. Message Queue

- **RabbitMQ**:
  - RabbitMQ là một message broker mạnh mẽ, đóng vai trò trung gian trong việc gửi và nhận thông báo từ các microservices đến client hoặc các hệ thống khác. Điều này giúp hệ thống có khả năng mở rộng và phân tán các thông báo một cách dễ dàng.

7. Cache & In-memory store

- **Redis**:
  - Redis đóng vai trò lưu trữ tạm thời và quản lý cache dữ liệu thông báo, giúp cải thiện tốc độ và hiệu quả của hệ thống.

7. Database Layer

- **PostgreSQL**:
  - Cơ sở dữ liệu chính cho hệ thống, quản lý và lưu trữ toàn bộ dữ liệu từ người dùng, sự kiện, voucher, và trò chơi. PostgreSQL là hệ quản trị cơ sở dữ liệu quan hệ mạnh mẽ, hỗ trợ giao dịch và dữ liệu phức tạp.

8. Docker Containerization

- **Vai trò**:

- Docker được sử dụng để đóng gói các microservices thành các container. Điều này giúp đơn giản hóa việc triển khai, quản lý và scale các dịch vụ. Mỗi dịch vụ có thể chạy độc lập và dễ dàng mở rộng hoặc triển khai trên các môi trường khác nhau.

9. Firebase Authentication

- **Vai trò**:
  - Sử dụng Firebase authentication để cung cấp dịch vụ xác thực, đăng ký tài khoản và xác thực OTP. Sử dụng trong microservice auth và module user.

C. Phân tích kiến trúc

1. Hiệu năng:

- **KrakenD**: Đảm bảo rằng các request từ client được xử lý hiệu quả.
- **Redis**: Giúp tăng tốc độ truy vấn, đảm bảo hiệu năng cơ chế caching nhờ bộ nhớ in-memory.

2. Khả năng mở rộng:

- **Microservices Architecture**: Mỗi dịch vụ được tách biệt và triển khai riêng lẻ, dễ dàng mở rộng khi cần thiết mà không ảnh hưởng đến toàn bộ hệ thống.
- **RabbitMQ**: Cho phép hệ thống xử lý lượng lớn thông báo và request một cách ổn định.
- **Docker**: Hỗ trợ khả năng mở rộng dễ dàng bằng cách triển khai các container mới khi cần.

3. Độ tin cậy:

- **RabbitMQ và Firebase**: Đảm bảo các thông báo được phát và nhận một cách đáng tin cậy.
- **PostgreSQL**: Cung cấp các tính năng quản lý dữ liệu an toàn, đảm bảo tính nhất quán và bảo mật dữ liệu.

4. Bảo mật:

- **Auth & User Service**: Đảm bảo việc xác thực và phân quyền chặt chẽ.

- **API Gateway (KrakenD)**: Đóng vai trò quan trọng trong việc kiểm soát truy cập vào các microservices, ngăn chặn các cuộc tấn công từ bên ngoài thông qua việc xác thực, rate-limiting và firewall.

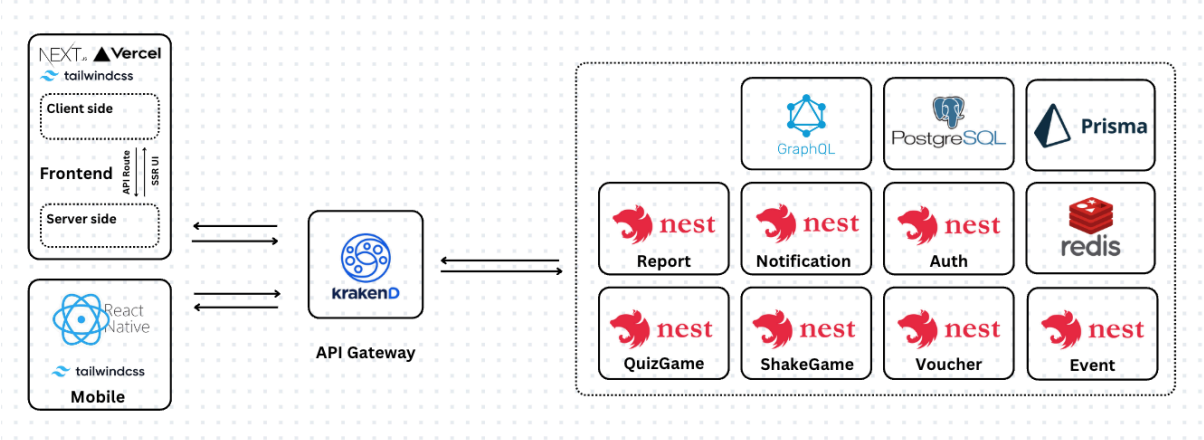
3. Các Thay Đổi Kiến Trúc Để Đáp Ứng Nhu Cầu Tăng Trưởng

Phiên bản 1:

- Mục tiêu: Đáp ứng yêu cầu cơ bản của vấn đề
- Mô tả:
  - *Kiến trúc Microservice:*
    - Phân chia hệ thống lớn thành nhiều dịch vụ (service) nhỏ, độc lập (independent services), giúp dễ dàng quản lý, mở rộng, và bảo trì.
    - Mỗi service thực hiện một chức năng cụ thể, ví dụ như **quản lý người dùng**, **quản lý trò chơi**, **quản lý sự kiện**, **quản lý thông báo**, v.v.
    - Mỗi service có thể được phát triển, triển khai (deploy), và mở rộng (scale) độc lập, giúp hệ thống linh hoạt hơn và giảm thiểu nguy cơ gây ảnh hưởng đến toàn hệ thống nếu có lỗi xảy ra ở một phần nhỏ.
  - *Sử dụng cơ sở dữ liệu quan hệ (PostgreSQL) cho dữ liệu người dùng và cấu hình trò chơi:*
    - Dữ liệu người dùng và cấu hình trò chơi thường có mối liên kết chặt chẽ (như người dùng và điểm số trò chơi), cơ sở dữ liệu quan hệ giúp lưu trữ và truy vấn các mối quan hệ này hiệu quả
    - PostgreSQL dễ dàng mở rộng khi hệ thống phát triển.
  - *Sử dụng Prisma:*
    - Prisma là một ORM (Object-Relational Mapping) mạnh mẽ giúp làm việc với cơ sở dữ liệu quan hệ như PostgreSQL một cách hiệu quả hơn.
    - Prisma hỗ trợ việc tự động hóa việc tạo và quản lý schema, migration database. Giúp dễ dàng truy vấn và thao tác thông qua cú pháp đơn giản
  - *Sử dụng API Gateway KrakenD để giao tiếp:*
    - KrakenD giúp tổng hợp nhiều API từ các microservices phía backend thành một điểm truy cập duy nhất, giúp client không cần gọi nhiều API riêng lẻ.
    - KrakenD có khả năng chuyển đổi dữ liệu từ các microservices, thay đổi định dạng hoặc lọc bỏ thông tin không cần thiết, giúp tối ưu dữ liệu

trước khi gửi về cho client.

- KrakenD hỗ trợ caching, giúp lưu trữ tạm thời các response để giảm số lần truy vấn tới backend.



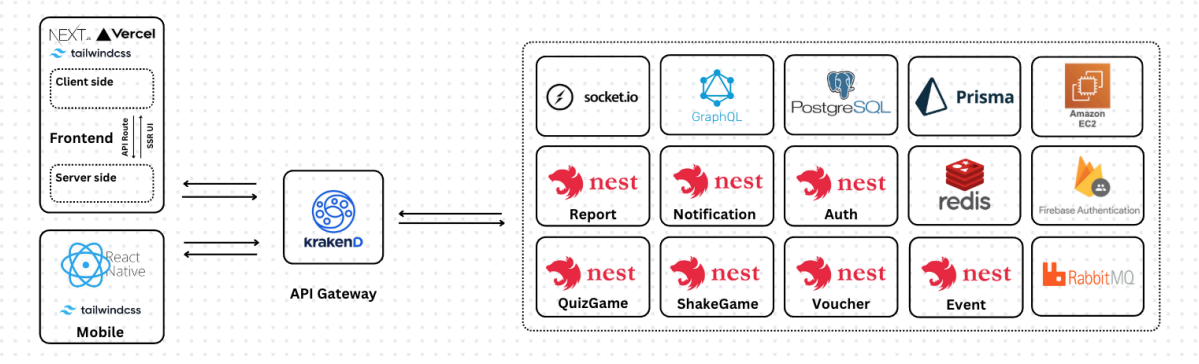
Phiên bản 2:

- ***Mục tiêu:*** *Thực hiện giao tiếp thời gian thực (real-time) giữa các Microservice và Client, và queue giữa các Microservices, đồng thời deploy các Microservice.*
- ***Mô tả:***
  - *Cung cấp giao tiếp real-time giữa client và server:*
    - **WebSocket** rất phù hợp cho các tính năng yêu cầu thời gian thực như trò chơi trực tuyến (quiz game, shake game) hoặc các thông báo đẩy. Ví dụ, trong QuizGame hoặc ShakeGame, WebSocket giúp các người chơi nhận được thông tin và kết quả ngay lập tức mà không cần tải lại trang.
    - **WebSocket.IO** cho phép server không chỉ phản hồi lại yêu cầu của client, mà còn có thể chủ động gửi dữ liệu tới client mà không cần client yêu cầu trước.
    - **AWS EC2** cung cấp một môi trường server mạnh mẽ để triển khai các ứng dụng backend và dịch vụ, trong đó có thể bao gồm các dịch vụ

WebSocket.

- *Tăng cường tính bền vững trong việc giao tiếp giữa các Microservices:*
  - Thay vì các dịch vụ phải giao tiếp trực tiếp với nhau (điều này có thể gây quá tải), **RabbitMQ** đóng vai trò như một **trung gian**, giúp hệ thống trở nên **linh hoạt** và dễ mở rộng hơn.
  - Nó cho phép các thành phần trong hệ thống trao đổi thông tin một cách không đồng bộ và bền vững, đảm bảo rằng tất cả các thông điệp đều được xử lý mà không bị mất dữ liệu.
  - **RabbitMQ** giúp cân bằng tải giữa các microservices bằng cách xếp hàng các thông điệp và phân phối chúng đến các dịch vụ xử lý theo thứ tự và mức độ ưu tiên. Điều này rất hữu ích khi có nhiều yêu cầu xử lý từ nhiều nguồn khác nhau.
- *Cung cấp giải pháp xác thực người dùng:*
  - **Firebase Authentication** giúp đơn giản hóa việc xác thực người dùng trong hệ thống mà không cần phải tự xây dựng từ đầu. **Firebase** hỗ trợ nhiều phương pháp đăng nhập khác nhau, từ truyền thống (email, password) đến hiện đại (**OTP** qua số điện thoại hoặc đăng nhập bằng Google).
  - Việc sử dụng **Firebase Authentication** đảm bảo rằng chỉ những người dùng đã xác thực mới có quyền truy cập vào các tài nguyên bảo mật như thông tin người dùng, trò chơi, hoặc dịch vụ khác.

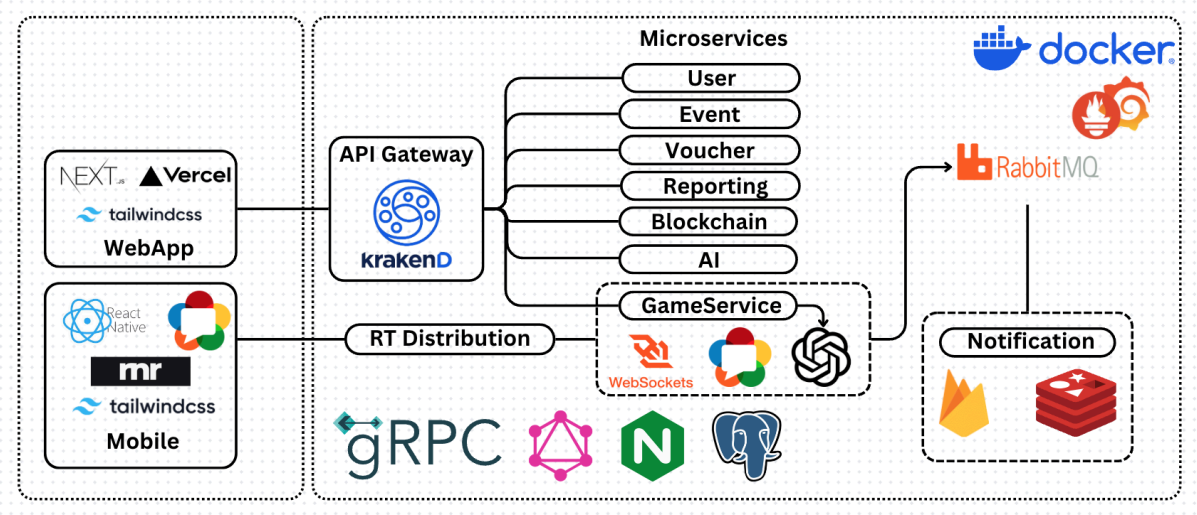
VOU - Marketing with Real-time Games	Phiên bản: 1.0
Kiến trúc phần mềm	Ngày: 06.09.2024



Phiên bản 3:

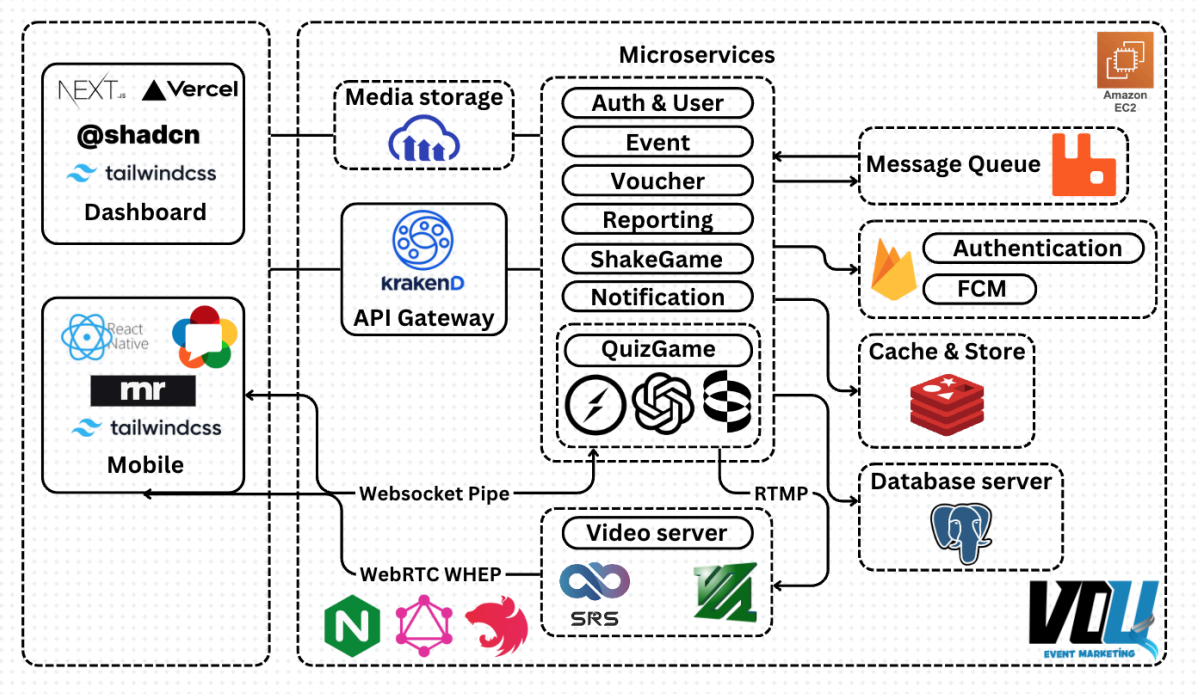
- ***Mục tiêu:*** *Thực hiện giao tiếp thời gian thực (real-time) giữa các Microservice và Client, và queue giữa các Microservices.*
- ***Mô tả:***
  - Hỗ trợ tương tác hiệu quả hơn giữa các dịch vụ:
    - **gRPC** hỗ trợ giao tiếp **song song** và **kết nối lâu dài**, phù hợp với việc cần lấy thông tin thời gian thực, như đồng bộ thời gian giữa các hệ thống hoặc dịch vụ trò chơi.
    - Sử dụng protocol buffers giúp gRPC **truyền tải dữ liệu nhanh hơn**, tiêu thụ ít băng thông và hiệu quả hơn so với việc sử dụng JSON hay XML, đặc biệt khi hệ thống có số lượng request lớn.
  - *Cân bằng tải, phân phối yêu cầu giữa các máy chủ:*
    - **NGINX** giúp **phân phối lưu lượng** từ các client đến nhiều instance của các dịch vụ backend, đảm bảo hệ thống hoạt động **ổn định** và **hiệu quả** khi có nhiều người dùng truy cập đồng thời.
    - **NGINX** làm việc như một **reverse proxy** để chuyển tiếp các yêu cầu từ client tới các backend services như API Gateway (**KrakenD**), giúp tối ưu hóa lưu lượng và cải thiện hiệu năng

- *Đóng gói ứng dụng thành các container:*
  - Docker cho phép **đóng gói** các microservices thành từng container, dễ dàng triển khai và quản lý trên các môi trường khác nhau như EC2, Vercel.
  - Giúp dễ dàng scale các dịch vụ theo nhu cầu mà không ảnh hưởng đến các container khác.
- *Tăng cường việc cache dữ liệu:*
  - Redis giúp cache dữ liệu được truy vấn thường xuyên như kết quả của API, thông tin người dùng, hoặc cấu hình trò chơi. Điều này giúp giảm tải cho cơ sở dữ liệu chính và tăng tốc độ phản hồi của hệ thống.
  - Redis có thể được sử dụng để **lưu trữ session** người dùng, giúp duy trì thông tin phiên đăng nhập mà không cần phải liên tục xác thực lại với cơ sở dữ liệu chính.
- *Hỗ trợ các microservice trò chơi tạo luồng stream:*
  - **WebRTC** rất phù hợp để **truyền tải dữ liệu video và âm thanh** trong các trò chơi trực tuyến hoặc các tình huống cần tương tác thời gian thực giữa các người chơi.
  - WebRTC cung cấp khả năng **truyền tải dữ liệu với độ trễ thấp**, giúp trải nghiệm trò chơi trở nên mượt mà hơn, điều này cực kỳ quan trọng trong các trò chơi nhiều người chơi.
- Hỗ trợ các microservice các tác vụ về ngôn ngữ:
  - Mô hình **OpenAI** có khả năng chuyển **văn bản thành giọng nói** (text-to-speech), giúp tạo ra trải nghiệm **tương tác bằng giọng nói** trong trò chơi hoặc ứng dụng. Điều này có thể ứng dụng cho việc cung cấp hướng dẫn, thông báo, hoặc tường thuật trong trò chơi.



Phiên bản cuối cùng:

- ***Mục tiêu:*** *Xây thành công luồng stream video AI cho Quiz, phát triển thành công các Microservice và ổn định các công nghệ đang sử dụng.*
- ***Mô tả:***
  - Giảm thiểu các độ trễ và phức tạp ở phía Front-end:
    - **gRPC** không được **hỗ trợ tốt trong các trình duyệt web** vì nó sử dụng giao thức HTTP/2, và điều này khiến việc tích hợp với các frontend như **Next.js** hoặc **React Native** trở nên phức tạp hơn.
    - RESTful APIs và GraphQL là lựa chọn hợp lý hơn cho các ứng dụng frontend, do chúng dễ dàng tương thích với trình duyệt và các thư viện frontend hiện có như Apollo Client.
  - Tiếp tục ổn định các công nghệ đang sử dụng
  - Tiếp tục xây dựng các API cho các Microservice.



4. Áp Dụng Design Patterns

A. Singleton pattern

- **Mô tả:**
  - **NestJS** sử dụng **Singleton** cho các dịch vụ (@Injectable), điều đó có nghĩa là chỉ có một instance duy nhất của một dịch vụ sẽ được tạo ra và sử dụng trong toàn bộ ứng dụng.
  - **Firebase Authentication**, **Redis**, **RabbitMQ**: Đây là những dịch vụ có thể được sử dụng xuyên suốt hệ thống, việc áp dụng **Singleton** sẽ đảm bảo chỉ có một instance của chúng được khởi tạo.
- **Lợi ích:**
  - Tiết kiệm tài nguyên vì chỉ tạo một instance duy nhất.
  - Dễ dàng quản lý trạng thái chia sẻ nếu cần (như cache, database connection).

VOU - Marketing with Real-time Games	Phiên bản: 1.0
Kiến trúc phần mềm	Ngày: 06.09.2024

```
import { Injectable } from '@nestjs/common';
import {  EventService,
          QuizService,
          ShakeService,
          UserService,
          VoucherService}
        from 'src/prisma/prisma.service';

@Injectable()
export class ReportService {
  constructor(private readonly shakeService: ShakeService,
              private readonly eventService: EventService,
              private readonly quizService: QuizService,
              private readonly userService: UserService,
              private readonly voucherService: VoucherService
  ) {}

  async countTotalPlayers(): Promise<number> {
    return this.userService.user.count();
  }

  async countTotalGames(): Promise<number> {
    return await this.shakeService.game.count() + await this.quizService.quizGame.count();
  }
```

B. Factory Method pattern

- **Mô tả:**
  - **NestJS** hỗ trợ sử dụng Factory Pattern để tạo các đối tượng tùy chỉnh hoặc dịch vụ một cách linh hoạt, thường thông qua các factory providers.
  - **Prisma** và **Database (PostgreSQL)**: Việc tạo các đối tượng kết nối đến database thông qua Factory Method giúp dễ dàng quản lý các kết nối.
  - **Apollo Client**: Khởi tạo các client giao tiếp với **GraphQL** có thể được trừu tượng hóa bằng Factory Method.
- **Lợi ích:**
  - Tạo đối tượng linh hoạt hơn dựa trên cấu hình hoặc tham số.
  - Có thể dễ dàng thay đổi logic tạo đối tượng mà không ảnh hưởng đến

mã nguồn chính.

- Cho phép tạo các đối tượng phức tạp một cách dễ dàng, đồng thời giúp mã dễ bảo trì hơn.

```
function getApolloLink(pairs: LinkConditionPair[]): ApolloLink {
  if (pairs.length === 1) {
    return pairs[0].link;
  } else {
    const [firstPair, ...restPairs] = pairs;
    return ApolloLink.from([
      // removeTypenameLink,
      // logoutLink,
      // middlewareLink,
      ApolloLink.split(
        firstPair.condition,
        firstPair.link,
        getApolloLink(restPairs)
      ),
    ]);
  }
}

const createHttpLink = (servicePath: string) => {
  console.log("servicePath", servicePath);
  return new HttpLink({
    // uri: `${BASE_URL}/${servicePath}`,
    uri: servicePath,
    fetchOptions
```

- **createHttpLink**: Phương thức này tạo ra các instance của **HttpLink** theo từng endpoint cụ thể. Điều này có một phần giống với **Factory Method**, vì nó sinh ra các đối tượng **HttpLink** dựa trên một tham số đầu vào (servicePath).

C. Adapter pattern

- **Mô tả:**
  - **RabbitMQ** và **Redis**: Khi cần giao tiếp giữa các thành phần với nhau (ví dụ như từ hệ thống nội bộ đến các dịch vụ bên ngoài), có thể sử dụng **Adapter** để chuẩn hóa các tương tác.
  - **KrakenD**: Khi cần tích hợp các API khác nhau thông qua một API Gateway, có thể sử dụng Adapter để chuyển đổi các API cũ hoặc không đồng nhất.
- **Lợi ích:**
  - Tăng khả năng tương thích giữa các hệ thống khác nhau mà không cần thay đổi mã gốc

```typescript
import { Injectable, Inject } from '@nestjs/common';
import { CACHE_MANAGER, Cache } from '@nestjs/cache-manager';

@Injectable()
export class RedisCacheService {
  constructor(@Inject(CACHE_MANAGER) private readonly cacheManager: Cache) {}

  async setCache(key: string, value: any, ttl: number): Promise<void> {
    await this.cacheManager.set(key, value, ttl);
  }

  async getCache<T>(key: string): Promise<T | undefined> {
    return await this.cacheManager.get<T>(key);
  }

  async deleteCache(key: string): Promise<void> {
    await this.cacheManager.del(key);
  }
}

```

- Lớp **RedisCacheService** đóng vai trò như một **Adapter**, nó chuyển đổi (hoặc

"thích ứng") giao diện của **CACHE_MANAGER** thành một giao diện đơn giản hơn mà client mong đợi. Các phương thức như **setCache**, **getCache**, và **deleteCache** trong lớp này là những phương thức bao bọc các phương thức của **CACHE_MANAGER**, làm cho hệ thống cache **Redis** dễ sử dụng hơn đối với các thành phần khác của ứng dụng mà không cần biết chi tiết triển khai bên trong **Redis**.

D. Decorator pattern

- **Mô tả:**
  - **NestJS** tận dụng Decorator Pattern rất mạnh mẽ, đặc biệt là trong việc sử dụng decorators để xác định metadata cho controller, service, module, v.v. Các decorators này không thay đổi lớp mà chúng trang trí, mà chỉ thêm metadata hoặc logic bổ sung.
- **Lợi ích:**
  - Giúp mã nguồn dễ đọc và rõ ràng hơn.
  - Cải thiện khả năng tái sử dụng mã, tăng cường sự linh hoạt trong việc mở rộng các class hoặc chức năng mà không cần thay đổi code gốc.

VOU - Marketing with Real-time Games	Phiên bản: 1.0
Kiến trúc phần mềm	Ngày: 06.09.2024

```
import {
  ExecutionContext,
  ForbiddenException,
  Global,
  Injectable,
} from '@nestjs/common';
import { AuthGuard, PassportStrategy } from '@nestjs/passport';
import { ExtractJwt, Strategy } from 'passport-jwt';
import { PrismaService } from '../prisma/prisma.service';
import { ConfigService } from '@nestjs/config';
import { GqlExecutionContext } from '@nestjs/graphql';

@Injectable()
@Global()
export class JwtStrategy extends PassportStrategy(Strategy) {
  constructor(
    private readonly prisma: PrismaService,
    configService: ConfigService,
  ) {
    super({
      jwtFromRequest: ExtractJwt.fromAuthHeaderAsBearerToken(),
      ignoreExpiration: true, // TODO: set to false
      secretOrKey: configService.get('ACCESS_TOKEN_SECRET'),
    });
  }

  async validate(payload: any) {
    const user = await this.prisma.user.findUniqueOrThrow({
      where: { id: payload.userId },
    });

    if (!user.hashedRefreshToken) {
      throw new ForbiddenException('User has signed out');
    }

    return user;
  }
}

```

E. Module pattern

- **Mô tả:**
  - NestJS sử dụng **Module Pattern** để tổ chức mã nguồn thành các module độc lập. Mỗi module có thể chứa các controller, service, và các provider của riêng nó.
- **Lợi ích:**
  - Giúp chia nhỏ hệ thống thành các phần dễ quản lý và bảo trì.
  - Dễ dàng tái sử dụng các module trong các dự án khác.
  - Hỗ trợ việc phát triển theo nhóm một cách hiệu quả, khi mỗi nhóm có thể làm việc độc lập trên các module khác nhau.

F. Proxy pattern

- **Mô tả:**
  - **KrakenD API Gateway**: KrakenD hoạt động như một Proxy giữa client và các service bên trong. Nó xử lý các request từ client và forward đến các microservice phù hợp.
  - **AWS EC2**: Khi triển khai trên EC2, có thể dùng Proxy để điều hướng lưu lượng truy cập thông qua các server khác nhau.
- **Lợi ích:**
  - Tăng tính bảo mật và tối ưu hóa lưu lượng truy cập, đồng thời giúp che giấu các hệ thống bên trong.

```
import { Injectable, Inject } from '@nestjs/common';
import { firstValueFrom, timeout } from 'rxjs';
import { ClientProxy } from '@nestjs/microservices';

@Injectable()
export class AuthService {
  constructor(@Inject('SHAKE_SERVICE') private rabbitClient: ClientProxy) {}

  async validateToken(token: string): Promise<any> {
    const response = firstValueFrom(this.rabbitClient.send({ cmd: 'validate_token' }, { token }).pipe(timeout(50000)));

    return response;
  }
}
```

G. Dependency Injection (DI) pattern

- **Mô tả:**
  - Dependency Injection là mẫu thiết kế cốt lõi trong NestJS. Nó cho phép các lớp phụ thuộc được cung cấp tự động thay vì khởi tạo thủ công, giúp giảm sự phụ thuộc chặt chẽ giữa các thành phần.
- **Lợi ích:**
  - Giảm sự phụ thuộc giữa các module và dịch vụ.
  - Tăng khả năng kiểm tra (unit test) vì có thể dễ dàng mock các phụ thuộc.
  - Dễ dàng mở rộng và thay đổi dịch vụ mà không cần thay đổi logic của lớp phụ thuộc.

```
import { Resolver, Query, Args } from '@nestjs/graphql';
import { ReportService } from './report.service';
import { UserEntity } from 'src/report/entities/user.entity';
import { QuizGameEntity } from 'src/report/entities/quiz.entity';
import { Game } from 'src/report/entities/game.entity';
import { Voucher } from 'src/report/entities/voucher.entity';
import { Event } from 'src/report/entities/event.entity';

@Resolver()
export class ReportResolver {
  constructor(private readonly reportService: ReportService) { }

  @Query(() => Number)
  async countTotalPlayers(): Promise<number> {
    return this.reportService.countTotalPlayers();
  }

```

- **ReportService** được tiêm vào thông qua constructor. NestJS quản lý **ReportService** như một provider và cung cấp nó cho **ReportResolver** khi khởi tạo. Điều này giúp giảm sự phụ thuộc cứng nhắc giữa **ReportResolver** và **ReportService**.

5. Phần ứng dụng

1. Danh sách các chức năng cho phân hệ Admin

STT	Chức năng	Mô tả	Mức độ hoàn thiện	Ghi chú
1	**Sign Up/ Sign In**	Đăng nhập, đăng ký tài khoản cho admin, có xác thực OTP	100%	
2	**Forgot password**	Reset password, đổi mật khẩu khi người dùng quên mật khẩu	100%	

3	**Sign Out**	Đăng xuất tài khoản admin	100%	
4	**Get account**	Lấy thông tin tài khoản dựa trên ID truyền vào	100%	
5	**Delete account**	Xóa tài khoản người dùng	100%	
6	**Activate account**	Kích hoạt tài khoản người dùng	100%	
7	**Update account**	Chỉnh sửa thông tin tài khoản	100%	
8	**Statistic & report**	- Thống kê dữ liệu người chơi - Thống kê dữ liệu game - Thống kê dữ liệu voucher - Thống kê dữ liệu đối tác - Thống kê các dữ liệu theo ngày và lọc cao nhất	100%	
9	**CRUD Quiz game**	- Thêm xóa sửa quiz game - Thêm xóa sửa bộ câu hỏi cho quiz game - Thêm xóa sửa bộ câu trả lời cho quiz game	100%	
10	**Shake Game**	Logic game & các tính năng chuyên biệt của game	100%	

2. Danh sách các chức năng cho phân hệ Khách hàng (User)

STT	Chức năng	Mô tả	Mức độ hoàn thiện	Ghi chú
1	**Shake game**	Các tính năng của game & quản lý nội dung shake game. Các tính năng về việc request và accept request lượt chơi với người dùng khác, chia sẻ lên MXH để nhận thêm lượt chơi. Các tính năng liên quan đến việc nhận và thu thập, chia sẻ item để nhận voucher.	100%	
2	**Quiz game**	Các tính năng cần thiết liên quan đến chơi quiz game Real-time và nhận voucher khi trở thành người chiến thắng.	100%	
3	**Sign in, sign up,**	Đăng nhập, đăng ký và đăng xuất tài khoản	100%	

	sign out	người dùng.		
4	**Forgot password**	Đổi mật khẩu khi người dùng quên mật khẩu	100%	
5	**Voucher**	Thực hiện nhận voucher và các chức năng liên quan đến việc sử dụng voucher online & offline. Các tính năng xem tổng quát số voucher của người dùng, xem chi tiết từng voucher và cách sử dụng.	100%	
6	**Event**	Các tính năng xem tổng quát, xem chi tiết từng Event, thực hiện việc chơi game của từng Event Các chức năng xử lý Event yêu thích và thông báo cho người dùng khi sự kiện đến gần	100%	
7	**User profile**	Các tính năng xem thông tin và sửa đổi thông tin người dùng, ảnh đại diện.	100%	
8	**Share Facebook**	Tính năng share link lên Facebook thông qua ứng dụng	100%	

3. Danh sách các chức năng cho phân hệ Brand

STT	Chức năng	Mô tả	Mức độ hoàn thiện	Ghi chú
1	**Authentication**	Đăng nhập, đăng ký, đăng xuất, xác thực tài khoản cho phân hệ thương hiệu (Brands)	100%	
2	**Forgot password**	Đổi mật khẩu tài khoản có xác thực OTP?	100%	
4	**CRU Event**	Tạo, đọc và cập nhật Event	100%	
5	**CRUD Voucher**	Tạo, đọc, xóa và cập nhật Voucher	100%	
6	**Chọn trò chơi cho Event**	Brand có thể chọn trò chơi cho Event	100%	

6. Các phần nâng cao khác (nếu có)

A. GraphQL:

a. **Tránh over-fetching và under-fetching**: Bạn chỉ lấy đúng dữ liệu cần thiết, không thừa hoặc thiếu, giúp tối ưu hóa hiệu suất và trải nghiệm người dùng.
b. **Truy vấn lồng nhau (Nested Data)**: Thay vì gọi nhiều endpoint như REST, GraphQL cho phép truy vấn dữ liệu ở nhiều cấp độ chỉ trong một lần yêu cầu, đơn giản hóa việc lấy dữ liệu phức tạp.
c. **Tính mở rộng và linh hoạt**: API có thể dễ dàng mở rộng mà không làm ảnh hưởng đến các phần khác của hệ thống, giúp phát triển nhanh hơn và dễ bảo trì.
d. **Single Endpoint**: Tất cả yêu cầu đều đi qua một endpoint duy nhất, giúp quản lý dễ dàng hơn và giảm thiểu số lượng yêu cầu không cần thiết.

B. Stream video AI:

a. **Công nghệ D-ID**: D-ID sử dụng trí tuệ nhân tạo để tạo ra các video với khuôn mặt nhân tạo dựa trên hình ảnh tĩnh. Hình ảnh khuôn mặt có thể được biến đổi thành các video sinh động với chuyển động tự nhiên của môi và các biểu cảm phù hợp với lời nói.
b. **OSSRS** (Open Source SRS) là một nền tảng phát trực tuyến mã nguồn mở chuyên hỗ trợ **RTMP (Real-Time Messaging Protocol)**, **HLS (HTTP Live Streaming)**, và **WebRTC**. OSSRS được biết đến với khả năng hiệu suất cao và độ trễ thấp trong truyền tải video.
c. **OpenAI TTS**: Công nghệ chuyển văn bản thành giọng nói của OpenAI sử dụng mô hình AI tiên tiến để tạo ra giọng nói tự nhiên và chân thực. Giọng nói được tạo ra không chỉ có ngữ điệu tự nhiên mà còn có thể tùy chỉnh về tốc độ, nhấn âm, và cảm xúc.

C. Websocket:

a. **Giao tiếp hai chiều (full-duplex)**: Cả server và client có thể gửi và nhận dữ liệu đồng thời mà không phải chờ yêu cầu từ phía đối phương.
b. **Real-time**: Dữ liệu được truyền ngay lập tức khi có sự kiện, lý tưởng cho các ứng dụng như chat, trò chơi trực tuyến, và cập nhật dữ liệu liên tục (bảng giá chứng khoán, thông báo, v.v.).
c. **Hiệu quả**: WebSocket sử dụng ít tài nguyên hơn vì không cần phải thiết lập lại kết nối HTTP cho mỗi lần truyền tải, giúp giảm độ trễ và tiết kiệm băng thông.

7. Kết Luận và Đánh Giá

- **Một số bài học kinh nghiệm từ quá trình thực hiện:**
  - Hiệu năng: Sử dụng caching với Redis và quản lý lưu lượng thông qua API Gateway giúp tối ưu hóa hiệu năng hệ thống, đặc biệt là khi hệ thống xử lý lượng dữ liệu lớn hoặc yêu cầu thời gian thực.
  - Tính linh hoạt: GraphQL cho phép chúng tôi truy vấn dữ liệu một cách linh hoạt, tránh tình trạng over-fetching và under-fetching, đồng thời tăng khả năng mở rộng cho hệ thống trong tương lai.
  - Quản lý sự kiện thời gian thực: WebSocket và RabbitMQ đã được áp dụng để xử lý các tác vụ yêu cầu thời gian thực, như game, thông báo và sự kiện, mang lại hiệu quả rõ rệt.
- **Tuy nhiên, chúng em cũng gặp một số khó khăn:**
  - Việc đồng bộ dữ liệu giữa các microservices và quản lý tính nhất quán dữ liệu trong môi trường phân tán yêu cầu phải thiết kế kỹ lưỡng và sử dụng message broker như RabbitMQ để đảm bảo độ tin cậy.
  - Tích hợp nhiều dịch vụ từ nhiều nguồn khác nhau đôi khi dẫn đến các vấn đề về tương thích và cần có các giải pháp adapter phù hợp.
- **Nhìn chung, kiến trúc hệ thống hiện tại đã chứng minh được hiệu quả trong việc đáp ứng nhu cầu của dự án và có thể dễ dàng mở rộng trong tương lai. Để tiếp tục phát triển, chúng em đề xuất:**
  - Cải tiến bảo mật: Tăng cường các biện pháp bảo mật, đặc biệt trong việc xác thực và phân quyền **Role-based Access Control** và **Attribute-based Access Control**.
  - Nâng cao hiệu suất: Tiếp tục tối ưu hóa caching (các chiến lược) và cân nhắc mở rộng hệ thống bằng cách triển khai các giải pháp cân bằng tải và tự động scale để đảm bảo khả năng đáp ứng khi lưu lượng truy cập tăng đột biến.

8. Tài Liệu Tham Khảo

- https://docs.nestjs.com/graphql/quick-start
- https://docs.nestjs.com/
- https://docs.nestjs.com/recipes/prisma
- https://www.krakend.io/docs/overview/
- https://dev.to/vue-storefront/building-microservices-with-nestjs-is-that-simple-4afm
- https://amplication.com/blog/working-with-microservices-with-nestjs
- https://blog.vietnamlab.vn/xay-dung-ung-dung-chat-realtime-voi-nestjs/
- https://astconsulting.in/blog/2023/08/29/real-time-web-applications-nestjs-websockets/
- https://softwaremill.com/caches-in-microservice-architecture/
- https://www.linkedin.com/pulse/exploring-caching-patterns-microservices-architecture-saeed-anabtawi/