VIỆT FOOD - ỨNG DỤNG WEB ĐẶT ĐỒ ĂN VIỆT NAM

Sinh viên thực tập

Ngày 1 tháng 6 năm 2025

Mục lục

Chương 1

Giới thiệu tổng quan

1.1 Giới thiệu dự án

Việt Food là một ứng dụng web fullstack với kiến trúc client-server nhằm cung cấp nền tảng đặt món ăn Việt Nam trực tuyến. Ứng dụng được phát triển với mục đích tạo ra trải nghiệm người dùng hiện đại, thân thiện, đồng thời đảm bảo hiệu suất cao và khả năng mở rộng. Với giao diện người dùng trực quan và hệ thống backend mạnh mẽ, Việt Food nhằm mục đích kết nối người dùng với các món ăn truyền thống và hiện đại của ẩm thực Việt Nam.

1.2 Công nghệ sử dụng

Dự án được xây dựng bằng các công nghệ hiện đại:

1.2.1 Backend

- **Node.js** và **Express**: Framework cho phát triển ứng dụng web server-side.
- **TypeScript**: Ngôn ngữ lập trình đảm bảo tính mạnh mẽ và an toàn cho mã nguồn.
- **MongoDB**: Cơ sở dữ liệu NoSQL phục vụ lưu trữ dữ liệu phi quan hệ.
- **Redis**: Hệ thống cache bộ nhớ để tối ưu hóa hiệu suất truy vấn.
- **Redis Stream**: Xử lý các tin nhắn và sự kiện real-time.
- **Compression**: Middleware giảm kích thước phản hồi HTTP.
- **Sharp**: Thư viện xử lý và tối ưu hóa hình ảnh.
- **Elasticsearch**: Công cụ tìm kiếm và đánh chỉ mục dữ liệu nhanh chóng.
- **Gemini AI**: Tích hợp trí tuệ nhân tạo cho các tính năng thông minh.

1.2.2 Frontend

- **React**: Thư viện JavaScript để xây dựng giao diện người dùng tương tác.
- **TypeScript**: Đảm bảo type safety và khả năng bảo trì cho mã nguồn frontend.
- **Tailwind CSS**: Framework CSS tiện ích giúp phát triển UI nhanh chóng và đồng nhất.
- **React Query**: Thư viện quản lý trạng thái và fetch data từ server.
- **Radix UI**: Bộ components primitives có thể tùy chỉnh cao.
- **Chart.js & Recharts**: Thư viện tạo biểu đồ cho phép hiển thị dữ liệu trực quan.
- **React Hook Form**: Quản lý biểu mẫu với hiệu suất cao.
- **Zod**: Thư viện xác thực dữ liệu với TypeScript.
- **Framer Motion**: Thư viện animation mạnh mẽ và linh hoạt.
- **Mapbox GL**: Tích hợp bản đồ tương tác để hiển thị địa điểm.

1.3 Phạm vi báo cáo

Báo cáo này tập trung vào phân tích toàn diện dự án Việt Food, bao gồm cả frontend và backend. Các nội dung chính được phân tích:

- **Kiến trúc tổng thể**: Cấu trúc dự án, mô hình phân lớp và luồng dữ liệu.
- **Frontend**: Cấu trúc mã nguồn, các components chính, quản lý trạng thái và routing.
- **Backend**: API endpoints, xử lý dữ liệu, tương tác với cơ sở dữ liệu.
- **Các kỹ thuật tối ưu**: Caching với Redis, xử lý tin nhắn với Redis Stream, nén dữ liệu, và tối ưu hóa hình ảnh.
- **Trải nghiệm người dùng**: UI/UX design, tính năng tương tác và phản hồi.
- **Khả năng mở rộng**: Cách dự án được thiết kế để xử lý việc tăng trưởng quy mô.

Việc phân tích toàn diện giúp hiểu rõ cách các thành phần khác nhau trong dự án tương tác và bổ trợ cho nhau, đồng thời đánh giá hiệu quả tổng thể của hệ thống.

1.4 Cấu trúc báo cáo

Báo cáo gồm 5 chương:

- **Chương 1**: Giới thiệu tổng quan về dự án Việt Food và các công nghệ sử dụng.
- **Chương 2**: Phân tích kiến trúc và thiết kế tổng thể của hệ thống.
- **Chương 3**: Phân tích chi tiết các thành phần frontend và backend.

- **Chương 4**: Đánh giá và nhận xét về hiệu quả của các kỹ thuật và giải pháp áp dụng.
- **Chương 5**: Kết luận và đề xuất hướng phát triển trong tương lai.

Chương 2

Phân tích kiến trúc và thiết kế hệ thống

2.1 Kiến trúc tổng thể

Việt Food áp dụng kiến trúc client-server hiện đại, với sự phân chia rõ ràng giữa frontend (React) và backend (Node.js/Express). Kiến trúc này tạo nên một hệ thống linh hoạt và mạnh mẽ, mang lại nhiều lợi ích:

- **Phát triển độc lập**: Các nhóm phát triển có thể làm việc song song trên frontend và backend.
- **Khả năng mở rộng**: Mỗi phần có thể được mở rộng độc lập dựa trên yêu cầu cụ thể.
- **Tái sử dụng API**: Backend cung cấp API RESTful có thể phục vụ nhiều client khác nhau (web, mobile).
- **Bảo mật tốt hơn**: Sự tách biệt giữa frontend và backend giúp tăng cường bảo mật.
- **Hiệu suất tối ưu**: Cho phép tối ưu hóa riêng biệt cho UI và xử lý dữ liệu.

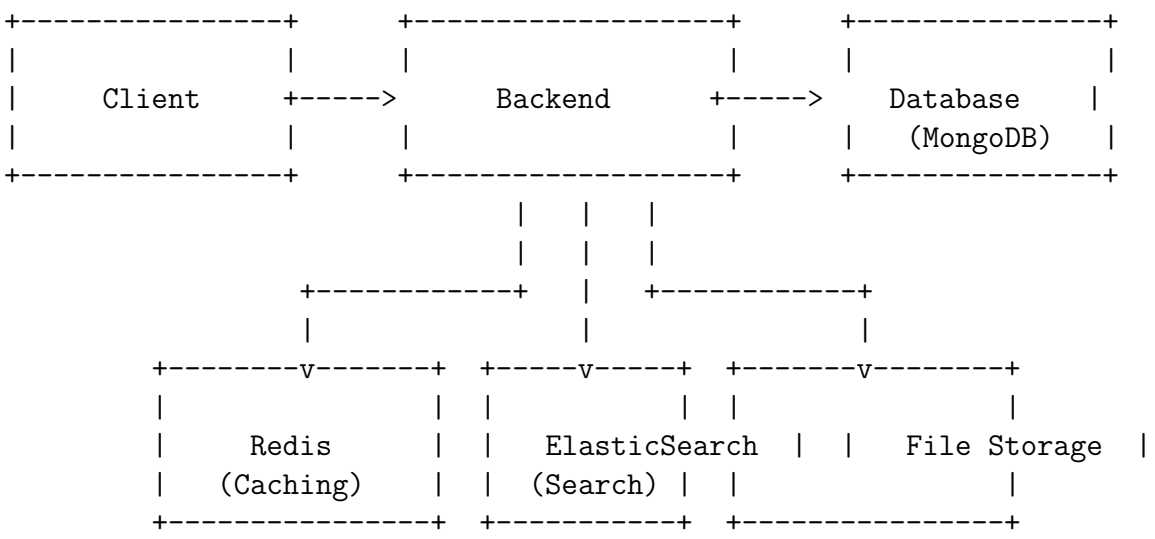
2.1.1 Mô hình kiến trúc

Hệ thống áp dụng kiến trúc MVC (Model-View-Controller) kết hợp với các dịch vụ (Services):

- **Model**: Đại diện cho cấu trúc dữ liệu và xử lý logic liên quan đến dữ liệu.
- **View**: Được thực hiện ở phía client, hiển thị thông tin cho người dùng.
- **Controller**: Xử lý các yêu cầu từ client, tương tác với các service và trả về kết quả.
- **Service**: Chứa logic nghiệp vụ, tương tác với database và các thành phần bên ngoài.

2.1.2 Sơ đồ kiến trúc

Kiến trúc hệ thống được thiết kế với nhiều lớp:



Hình 2.1: Sơ đồ kiến trúc tổng thể của hệ thống

2.2 Thiết kế module/component chính

Cấu trúc dự án được tổ chức thành các module chức năng riêng biệt, gồm cả phần frontend và backend:

2.2.1 Controllers

Xử lý các request từ client và điều phối các thao tác:

- **DishController**: Quản lý thông tin món ăn
- **CategoryController**: Quản lý danh mục
- **AuthController**: Xử lý xác thực người dùng
- **MessageController**: Quản lý tin nhắn

2.2.2 Services

Chứa logic nghiệp vụ chính:

- **DishService**: Xử lý logic liên quan đến món ăn, bao gồm caching với Redis
- **CategoryService**: Quản lý danh mục món ăn
- **AuthService**: Xử lý đăng nhập, đăng ký và xác thực
- **MessageWorkerService**: Xử lý tin nhắn với Redis Stream
- **SearchService**: Tương tác với Elasticsearch để tìm kiếm

2.2.3 Middlewares

Xử lý các tác vụ trung gian:

- **Authentication**: Kiểm tra và xác thực người dùng
- **ImageProcessor**: Xử lý và tối ưu hóa hình ảnh với Sharp
- **Error Handler**: Xử lý lỗi thống nhất

2.2.4 Models

Mô tả cấu trúc dữ liệu trong backend:

- **DishModel**: Mô hình dữ liệu cho món ăn
- **CategoryModel**: Mô hình dữ liệu cho danh mục
- **UserModel**: Mô hình dữ liệu người dùng
- **MessageModel**: Mô hình dữ liệu tin nhắn

2.2.5 Frontend Components

Frontend được xây dựng theo kiến trúc component-based với React, tổ chức thành các thành phần:

- **UI Components**: Các component giao diện cơ bản, được xây dựng trên Radix UI và tùy chỉnh với Tailwind CSS
- **Layout Components**: Quản lý bố cục chung của ứng dụng
- **Page Components**: Tương ứng với các trang trong ứng dụng
- **Feature Components**: Các component chức năng đặc thù như giỏ hàng, tìm kiếm, thanh toán

2.2.6 Frontend Services và Hooks

Phần frontend sử dụng các service và custom hooks để quản lý logic:

- **API Services**: Xử lý các tương tác với backend thông qua axios
- **Context API**: Sử dụng React Context để quản lý trạng thái toàn cục như xác thực người dùng (CartContext, AuthContext)
- **Custom Hooks**: Các hooks chuyên biệt như useAuth, useCart, useProfile để đơn giản hóa quản lý trạng thái và data fetching
- **Local Storage**: Lưu trữ tokens xác thực và thông tin session người dùng

2.2.7 Quản lý Routing

Việt Food sử dụng thư viện wouter để quản lý routing trong ứng dụng, với các route chính:

- **Public Routes**: Trang chủ, danh sách món ăn, chi tiết món ăn, tìm kiếm
- **Protected Routes**: Giỏ hàng, quản lý đơn hàng, tài khoản cá nhân
- **Admin Routes**: Quản lý món ăn, danh mục, người dùng và thống kê

2.2.8 State Management

Quản lý trạng thái trong ứng dụng được thực hiện thông qua:

- **React Context**: Quản lý trạng thái toàn cục như thông tin người dùng, giỏ hàng
- **React Query**: Quản lý trạng thái server và cache dữ liệu
- **Local Component State**: Quản lý trạng thái độc lập của từng component

2.3 Thiết kế cơ sở dữ liệu

Hệ thống sử dụng MongoDB làm cơ sở dữ liệu chính với các collection chính:

- **Dishes**: Lưu trữ thông tin món ăn
- **Categories**: Danh mục món ăn
- **Users**: Thông tin người dùng
- **Messages**: Lưu trữ tin nhắn

Bên cạnh đó, hệ thống còn sử dụng:

- **Redis**: Lưu trữ cache, quản lý phiên làm việc và xử lý tin nhắn real-time
- **Elasticsearch**: Đánh chỉ mục và tìm kiếm dữ liệu nhanh chóng

2.3.1 Lý do lựa chọn thiết kế

Việc kết hợp MongoDB, Redis và Elasticsearch cung cấp nhiều lợi thế:

- **MongoDB**: Lưu trữ dữ liệu linh hoạt với schema động, phù hợp với ứng dụng có dữ liệu phức tạp
- **Redis**: Caching hiệu quả, giảm tải cho database chính và xử lý thông tin tạm thời
- **Elasticsearch**: Tìm kiếm mạnh mẽ với full-text search và ranking

Chương 3

Phân tích chi tiết các thành phần dự án

3.1 Phân tích chi tiết các thành phần của dự án

Phần này phân tích chi tiết các thành phần frontend và backend của dự án Việt Food. Chúng tôi sẽ đi sâu vào các tính năng nổi bật, kỹ thuật triển khai, và những giải pháp được áp dụng để đảm bảo hiệu suất và trải nghiệm người dùng tốt.

3.2 Frontend - Giao diện người dùng

3.2.1 Cấu trúc component hiện đại

Frontend của Việt Food được xây dựng bằng React và TypeScript, áp dụng một cấu trúc có tính mô-đun cao và dễ bảo trì. Hệ thống component được tổ chức theo các lớp:

1. **Atomic Design Pattern**: Các component được tổ chức theo mô hình Atoms, Molecules, Organisms, Templates và Pages.
2. **Chia nhỏ component**: Các thành phần UI được tách thành các component nhỏ để tăng khả năng tái sử dụng.
3. **Cô lập logic**: Logic được tách biệt với giao diện thông qua các custom hooks.

Một ví dụ về cấu trúc component trong Việt Food:

```
// Atomic design: Button (Atom)
src/components/ui/button.tsx

// DishCard (Molecule)
src/components/dishes/dish-card.tsx

// DishList (Organism)
src/components/dishes/dish-list.tsx

// HomePage (Page)
src/pages/home.tsx
```

3.2.2 Styling và Thiết kế

Việt Food sử dụng Tailwind CSS kết hợp với các component từ Radix UI để tạo ra một hệ thống giao diện nhất quán và hiện đại:

- **Utility-first CSS**: Sử dụng các lớp của Tailwind giúp phát triển nhanh và nhất quán.
- **Theme Configuration**: Hệ thống màu sắc, typography, và spacing được cấu hình trong tailwind.config.ts.
- **Component Libraries**: Sử dụng Radix UI làm nền tảng, tùy chỉnh với Tailwind CSS.
- **Responsive Design**: Ứng dụng được thiết kế đáp ứng trên các kích thước màn hình khác nhau.

Một ví dụ về cách kết hợp Radix UI và Tailwind CSS:

```
// Button component sử dụng Radix UI Slot với Tailwind CSS
import { Slot } from "@radix-ui/react-slot"
import { cva, type VariantProps } from "class-variance-authority"

const buttonVariants = cva(
  "inline-flex items-center justify-center rounded-md text-sm font-medium",
  {
    variants: {
      variant: {
        default: "bg-primary text-primary-foreground hover:bg-primary/90",
        destructive: "bg-destructive text-destructive-foreground hover:bg-destructive/
        outline: "border border-input bg-background hover:bg-accent hover:text-accent-
        // ... các variants khác
      },
      size: {
        default: "h-10 px-4 py-2",
        sm: "h-9 rounded-md px-3",
        lg: "h-11 rounded-md px-8",
      },
    },
    defaultVariants: {
      variant: "default",
      size: "default",
    },
  }
)
```

3.2.3 Quản lý dữ liệu và State Management

Việt Food sử dụng một hỗn hợp các công cụ để quản lý dữ liệu và trạng thái ứng dụng:

- **React Context API**: Quản lý trạng thái toàn cục của ứng dụng như xác thực người dùng (auth), giỏ hàng (cart).
- **Custom Hooks**: Tạo ra các hooks chuyên biệt như useAuth, useProfile để truy cập và cập nhật trạng thái.
- **Zod**: Validation dữ liệu chặt chẽ với TypeScript.
- **React Hook Form**: Quản lý form với hiệu suất cao.
- **Local Storage**: Lưu trữ access token và refresh token trên client.

Ví dụ về việc triển khai Context API và custom hooks để quản lý giỏ hàng:

```
// CartContext.tsx
import { createContext, useContext, useState, ReactNode } from "react";

interface CartItem {
  id: string;
  name: string;
  price: number;
  imageUrl: string;
  quantity: number;
}

interface CartContextType {
  cartItems: CartItem[];
  isCartOpen: boolean;
  addToCart: (item: CartItem) => void;
  removeFromCart: (id: string) => void;
  // ... các phương thức khác
}

const CartContext = createContext<CartContextType | undefined>(undefined);

export const CartProvider = ({ children }: { children: ReactNode }) => {
  const [cartItems, setCartItems] = useState<CartItem[]>([]);
  const [isCartOpen, setIsCartOpen] = useState(false);

  // Triển khai các phương thức

  return (
    <CartContext.Provider value={{ cartItems, isCartOpen, ... }}>
      {children}
    </CartContext.Provider>
  );
};

// Custom hook để sử dụng context
export const useCart = () => {
```

```
  const context = useContext(CartContext);
  if (context === undefined) {
    throw new Error("useCart must be used within a CartProvider");
  }
  return context;
};

function DishListPage() {
  const [filters, setFilters] = useState({
    category: '',
    search: '',
    page: 1
  })

  const { data, isLoading, error } = useDishes(filters)

  // ...
}
```

3.2.4 Routing và Navigation

Việt Food sử dụng thư viện wouter cho routing nhẹ nhàng và hiệu quả:

- **Lazy Loading**: Các trang được tải lazy để giảm kích thước bundle ban đầu.
- **Route Protection**: Các route được bảo vệ dựa trên vai trò người dùng.
- **URL Params và Query Strings**: Xử lý các tham số đường dẫn và query string cho tìm kiếm và lọc.

Cấu trúc routing trong Việt Food:

```
import { Route, Switch } from 'wouter'
import { lazy, Suspense } from 'react'

// Lazy loaded components
const Home = lazy(() => import('./pages/home'))
const DishList = lazy(() => import('./pages/dish-list'))
const DishDetail = lazy(() => import('./pages/dish-detail'))
const Cart = lazy(() => import('./pages/cart'))

function App() {
  return (
    <Suspense fallback={<LoadingSpinner />}>
      <Switch>
        <Route path="/" component={Home} />
        <Route path="/dishes" component={DishList} />
        <Route path="/dishes/:id" component={DishDetail} />
        <Route path="/cart">
          {/* Protected route */}
```

```
          {isAuthenticated ? <Cart /> : <Redirect to="/login" />}
        </Route>
        {/* ... more routes */}
      </Switch>
    </Suspense>
  )
}
```

3.2.5 Optimization tại Frontend

Frontend của Việt Food áp dụng nhiều kỹ thuật tối ưu hoá để đảm bảo hiệu suất cao:

- **Code Splitting**: Chia nhỏ bundle JS thông qua lazy loading.
- **Memoization**: Sử dụng React.memo, useMemo, và useCallback để giảm re-renders không cần thiết.
- **Virtualization**: Sử dụng windowing cho danh sách dài.
- **Image Optimization**: Sử dụng các kỹ thuật lazy loading, responsive images, và next-gen formats (WebP).
- **Bundle Optimization**: Sử dụng Vite để bundling nhanh và hiệu quả.

Ví dụ về tối ưu hoá component với memoization:

```
import { memo, useMemo, useCallback } from 'react'

// Memoized component để tránh re-renders không cần thiết
const DishCard = memo(function DishCard({ dish, onAddToCart }) {
  // Xử lý component rendering
  return (
    <div className="dish-card">
      <img
        src={dish.imageUrl}
        alt={dish.name}
        loading="lazy" // Lazy load images
      />
      <h3>{dish.name}</h3>
      <p>{dish.price.toLocaleString('vi-VN')} đ</p>
      <button onClick={() => onAddToCart(dish)}>Thêm vào giỏ</button>
    </div>
  )
})

function DishList({ dishes }) {
  // Memoize expensive calculations
  const sortedDishes = useMemo(() => {
    return [...dishes].sort((a, b) => a.price - b.price)
  }, [dishes])
```

```
  // Memoize callback functions
  const handleAddToCart = useCallback((dish) => {
    // Logic để thêm vào giỏ hàng
  }, [])

  return (
    <div className="dish-grid">
      {sortedDishes.map(dish => (
        <DishCard
          key={dish.id}
          dish={dish}
          onAddToCart={handleAddToCart}
        />
      ))}
    </div>
  )
}
```

3.3 Backend - Kỹ thuật tối ưu hóa

3.3.1 Redis Caching

3.3.2 Khái niệm và lợi ích

Redis là hệ thống lưu trữ dữ liệu key-value trong bộ nhớ, có khả năng lưu trữ nhiều kiểu dữ liệu khác nhau. Trong Việt Food, Redis được sử dụng chủ yếu làm hệ thống cache để tối ưu hóa thời gian truy vấn dữ liệu:

- **Tốc độ truy xuất nhanh**: Dữ liệu được lưu trong bộ nhớ, giảm độ trễ so với truy vấn database
- **Giảm tải cho database chính**: Các truy vấn phổ biến được cache, giảm số lượng truy cập vào MongoDB
- **Tăng khả năng phục vụ đồng thời**: Hệ thống có thể xử lý nhiều request hơn trong cùng một thời điểm

3.3.3 Triển khai trong dự án

Trong Việt Food, Redis được cấu hình và khởi tạo trong file `config/redis.ts`:

```
import Redis from "ioredis"
import { REDIS_PORT, REDIS_HOST } from "../config"

const redis = new Redis({
    host: REDIS_HOST,
    port: REDIS_PORT,
})
```

```
redis.on("connect", () => {
    console.log(`[SUCCESS] Redis connected: ${redis.options.host}:${redis.options.port
})

redis.on("error", (err) => {
    console.error(err)
})

export default redis
```

3.3.4 Cache dữ liệu món ăn

Dữ liệu món ăn là loại dữ liệu được truy cập thường xuyên nhưng ít thay đổi, nên việc cache là rất hiệu quả. Trong `DishService`, các phương thức tìm kiếm món ăn đều áp dụng caching:

```
// Lấy món ăn theo ID với cache
static async findById(id: string | Types.ObjectId): Promise<IDishDocument | null> {
    const cacheKey = `dish:${id}`
    const cachedDish = await redis.get(cacheKey)

    if (cachedDish) {
        return JSON.parse(cachedDish)
    }

    const dish = await DishModel.findById(id).populate('category').lean()
    if (dish) {
        await redis.set(cacheKey, JSON.stringify(dish), 'EX', 3600 * 24)
        return dish
    }

    return null
}
```

3.3.5 Prime Cache khi khởi động

Một điểm đáng chú ý là Việt Food sử dụng kỹ thuật "prime cache"khi khởi động server - tức là nạp trước dữ liệu phổ biến vào cache:

```
static async primeAllDishesCache(): Promise<void> {
    try {
        const allDishes = await DishModel.find({ isAvailable: true }).lean()
        if (allDishes && allDishes.length > 0) {
            const pipeline = redis.pipeline()
            for (const dish of allDishes) {
                const cacheKey = `dish:${dish._id}`
                dish.imageUrl = getFullImageUrl(dish.imageUrl)
                pipeline.set(cacheKey, JSON.stringify(dish), 'EX', 3600 * 24)
```

```
            }
            await pipeline.exec()
        }
    } catch (error) {
        console.error('Error priming all dishes cache:', error)
    }
}
```

Phương thức này được gọi khi server khởi động trong `server.ts`:

```
connectDB();
connectElasticsearch();
DishService.primeAllDishesCache()
```

3.3.6 Invalidation cache

Khi dữ liệu thay đổi, cache cần được cập nhật hoặc xóa để tránh dữ liệu không đồng bộ. Việt Food xử lý vấn đề này bằng cách xóa các key cache liên quan khi có thay đổi:

```
// Xóa cache khi cập nhật món ăn
const keys = await redis.keys('dishes:*')
if (keys.length > 0) {
    const pipeline = redis.pipeline()
    keys.forEach(key => pipeline.del(key))
    await pipeline.exec()
}
```

3.4 Redis Stream

3.4.1 Khái niệm và lợi ích

Redis Stream là một cấu trúc dữ liệu mới trong Redis, cho phép lưu trữ và quản lý các dòng sự kiện theo thời gian thực, tương tự như một message broker:

- **Xử lý tin nhắn không đồng bộ**: Các thành phần khác nhau có thể giao tiếp mà không cần chờ đợi nhau
- **Mô hình publish-subscribe**: Phân tách giữa producer và consumer
- **Khả năng mở rộng**: Nhiều consumer có thể xử lý cùng một stream
- **Độ tin cậy**: Tin nhắn được lưu trữ lâu dài và có thể được xử lý lại nếu cần

3.4.2 Triển khai trong dự án

Trong Việt Food, Redis Stream được sử dụng chủ yếu để xử lý tin nhắn. Đầu tiên, stream key được định nghĩa trong file cấu hình Redis:

```
export const MESSAGE_STREAM_KEY = "message_stream";
```

MessageWorkerService xử lý tin nhắn từ stream:

```
export default class MessageWorkerService {
    static async createConsumerGroup() {
        try {
            const streamExists = await redis.exists(MESSAGE_STREAM_KEY);
            if (!streamExists) {
                await redis.xgroup('CREATE', MESSAGE_STREAM_KEY, GROUP_NAME, '0', 'MK
            }
        } catch (error) {
            console.log('error', error);
        }
    }

    static async processStreamMessages() {
        while (true) {
            try {
                const result = await redis.xreadgroup(
                    'GROUP', GROUP_NAME,
                    CONSUMER_NAME,
                    'COUNT', 1,
                    'BLOCK', 0,
                    'STREAMS', MESSAGE_STREAM_KEY,
                    '>'
                ) as XReadGroupResult;

                const message = this.convertStreamMessageToObject(result?.[0]?.[1]?.[
                if (message) {
                    await MessageService.createMessage(message.userId, message.content
                }
            } catch (error) {
                console.error('Error processing stream messages:', error);
            }
        }
    }
}
```

Stream consumer được khởi động riêng biệt từ file `consumer.ts`:

```
import MessageWorkerService from './services/message-worker.service';

(async () => {
    await MessageWorkerService.createConsumerGroup();
    await MessageWorkerService.processStreamMessages();
})();
```

3.5 HTTP Compression

3.5.1 Khái niệm và lợi ích

HTTP Compression là kỹ thuật nén dữ liệu trước khi gửi từ server đến client, giúp giảm kích thước dữ liệu truyền tải:

- **Giảm băng thông**: Tiết kiệm băng thông và chi phí mạng
- **Tăng tốc độ tải trang**: Người dùng nhận được phản hồi nhanh hơn
- **Cải thiện trải nghiệm**: Đặc biệt hiệu quả với người dùng có kết nối mạng chậm

3.5.2 Triển khai trong dự án

Việt Food sử dụng middleware `compression` của Express để nén dữ liệu:

```
import compression from "compression";

const compressOptions = {
    threshold: 1024,  // Chỉ nén dữ liệu lớn hơn 1KB
    level: 6,        // Mức độ nén từ 1-9, cân bằng giữa tốc độ và hiệu quả nén
}

app.use(compression(compressOptions));
```

Cấu hình này được thiết lập trong `server.ts`, áp dụng cho tất cả các request. Điểm đáng chú ý:

- **threshold 1024 bytes**: Chỉ nén các phản hồi có kích thước lớn hơn 1KB, tránh lãng phí tài nguyên CPU cho các phản hồi nhỏ
- **level 6**: Mức nén cân bằng giữa tốc độ và hiệu quả nén

3.6 Tối ưu hóa hình ảnh với Sharp

3.6.1 Khái niệm và lợi ích

Sharp là thư viện xử lý hình ảnh cho Node.js, được xây dựng trên libvips - thư viện xử lý hình ảnh hiệu suất cao:

- **Tốc độ xử lý cao**: Nhanh hơn nhiều so với các thư viện xử lý ảnh thông thường
- **Tiết kiệm bộ nhớ**: Sử dụng bộ nhớ hiệu quả khi xử lý ảnh lớn
- **Tối ưu hóa định dạng**: Chuyển đổi sang các định dạng hiệu quả như WebP
- **Thay đổi kích thước**: Giảm kích thước ảnh nhưng vẫn giữ chất lượng tốt

3.6.2 Triển khai trong dự án

Việt Food sử dụng Sharp thông qua middleware `image-processor.middleware.ts` để xử lý ảnh khi người dùng tải lên:

```
import sharp from 'sharp';
import fs from 'fs';
import path from 'path';

const processOneImage = async (file: Express.Multer.File): Promise<void> => {
    const originalPath = file.path;
    const fileDir = path.dirname(originalPath);
    const filename = path.basename(originalPath, path.extname(originalPath));
    const webpFilename = `${filename}.webp`;
    const outputPath = path.join(fileDir, webpFilename);

    await sharp(originalPath)
        .resize(1000)
        .webp({
            quality: 80,
            effort: 6
        })
        .toFile(outputPath);

    fs.unlinkSync(originalPath);

    file.path = outputPath;
    file.filename = webpFilename;
    file.mimetype = 'image/webp';
};
```

Middleware này thực hiện các tối ưu:

- **Resize**: Giảm kích thước ảnh xuống tối đa 1000px để tiết kiệm không gian lưu trữ
- **Chuyển đổi sang WebP**: Định dạng hiện đại có kích thước nhỏ hơn nhưng chất lượng tương đương hoặc tốt hơn JPEG/PNG
- **Điều chỉnh chất lượng**: Cân bằng giữa kích thước file và chất lượng hình ảnh (quality: 80)
- **Mức độ nén tối ưu**: Cấu hình effort: 6 đảm bảo cân bằng giữa tốc độ xử lý và hiệu quả nén

Middleware này được áp dụng cho các route xử lý upload hình ảnh, đảm bảo tất cả hình ảnh đều được tối ưu trước khi lưu trữ.

3.7 Tích hợp và tương tác giữa các kỹ thuật

Các kỹ thuật tối ưu trong Việt Food không hoạt động độc lập mà bổ trợ cho nhau:

- **Sharp + Redis**: Hình ảnh được tối ưu với Sharp, sau đó URL được lưu trong Redis cache
- **Redis Cache + Compression**: Dữ liệu được cache và nén khi truyền tải, giảm thời gian tải trang
- **Redis Stream + Services**: Xử lý tin nhắn không đồng bộ, giải phóng server chính để xử lý các request khác

Việc kết hợp các kỹ thuật này tạo nên một pipeline tối ưu, từ khi dữ liệu được tạo ra, lưu trữ đến khi được truyền tải đến người dùng cuối.

3.8 Hệ thống Agent thông minh

3.8.1 Giới thiệu về Agent

Việt Food triển khai một hệ thống Agent thông minh hỗ trợ khách hàng trong quá trình đặt món và tương tác với ứng dụng:

- **Trợ lý ảo**: Agent hoạt động như một trợ lý ảo thông minh, tương tác với người dùng qua giao diện chat
- **Xử lý ngôn ngữ tự nhiên**: Sử dụng mô hình NLP để hiểu và phản hồi câu hỏi của người dùng
- **Tư vấn món ăn**: Gợi ý món ăn dựa trên sở thích và lịch sử đặt hàng của người dùng
- **Xử lý quy trình**: Hướng dẫn người dùng hoàn thành quy trình đặt hàng

3.8.2 Kiến trúc Agent

Hệ thống Agent được xây dựng với kiến trúc microservice, tách biệt với core service chính:

```
// agent-service.ts
export class AgentService {
    private nlpProcessor: NLPProcessor;
    private recommendationEngine: RecommendationEngine;

    constructor() {
        this.nlpProcessor = new NLPProcessor();
        this.recommendationEngine = new RecommendationEngine();
    }

    async processUserMessage(userId: string, message: string): Promise<AgentResponse>
        // Xử lý tin nhắn từ người dùng
        const intent = await this.nlpProcessor.detectIntent(message);

        // Lưu tin nhắn vào cơ sở dữ liệu
        await this.storeMessage(userId, message, 'user');
```

```
        // Tạo phản hồi dựa trên intent
        const response = await this.generateResponse(userId, intent, message);

        // Lưu phản hồi của Agent
        await this.storeMessage(userId, response.content, 'agent');

        return response;
    }

    private async generateResponse(userId: string, intent: Intent, message: string): P
        switch (intent.type) {
            case 'DISH_RECOMMENDATION':
                return await this.recommendationEngine.getRecommendations(userId, inte
            case 'ORDER_STATUS':
                return await this.getOrderStatus(userId);
            // Xử lý các intent khác
            default:
                return { content: 'Xin lỗi, tôi không hiểu yêu cầu của bạn.' };
        }
    }
}
```

3.8.3 Lưu trữ message và quản lý context

Một trong những thách thức lớn nhất của hệ thống Agent là việc lưu trữ tin nhắn và duy trì context hội thoại. Việt Food sử dụng MongoDB để lưu trữ tin nhắn và Redis để quản lý context session:

```
// message.model.ts
import { Schema, model, Types, Document } from "mongoose";

export enum MessageRole {
    USER = "user",
    ASSISTANT = "assistant",
}

export interface IMessage {
    userId: Types.ObjectId;
    content: string;
    role: MessageRole;
}

const messageSchema = new Schema({
    userId: { type: Schema.Types.ObjectId, ref: "User", required: true },
    content: { type: String, required: true },
    role: { type: String, enum: MessageRole, required: true },
}, { timestamps: true });
```

```
const Message = model("Message", messageSchema);
```

Dịch vụ MessageService cung cấp các phương thức để lưu trữ và truy xuất tin nhắn, kết hợp với Redis Stream để xử lý bất đồng bộ:

```
// message.service.ts
import redis from "@/config/redis";
import Message, { MessageRole } from "@/model/message.model";
import { Types } from "mongoose";

export const MESSAGE_STREAM_KEY = "message_stream";
export const MESSAGE_GROUP_NAME = "message_group";

export default class MessageService {
    static async createMessage(userId: Types.ObjectId, content: string, role: MessageR
        const newMessage = await Message.create({ userId, content, role });
        return newMessage;
    }

    static async getMessagesByUserId(userId: Types.ObjectId) {
        const messages = await Message.find({ userId }).sort({ createdAt: -1 }).lean()
        return messages;
    }

    static addMessageToStream = async (streamKey: string, message: Record<string, any>
        const result = await redis.xadd(streamKey, "*", ...Object.entries(message).fla
        return result;
    }
}
```

3.8.4 Quản lý context hội thoại

Duy trì context hội thoại là yếu tố quan trọng để Agent có thể hiểu được tiến trình đối thoại đang diễn ra. Trong hệ thống Việt Food, context hội thoại được xây dựng bằng cách kết hợp lịch sử tin nhắn với trạng thái hội thoại hiện tại. Mỗi trạng thái hội thoại có thể bao gồm các thông tin quan trọng như:

- **Trạng thái hội thoại**: Đang tìm kiếm món ăn, đang xác nhận đơn hàng, v.v.
- **Thông tin người dùng**: Sở thích, ràng buộc ăn uống, địa chỉ giao hàng
- **Giỏ hàng hiện tại**: Các món ăn đã được thêm vào giỏ hàng trong cuộc hội thoại
- **Intent trước đó**: Mục đích giao tiếp trước đó của người dùng

Một phương pháp tối ưu để quản lý context hội thoại có thể được triển khai bằng cách sử dụng Google GenAI và in-memory cache, với định nghĩa cơ bản như sau:

```
// agent.service.ts
import { Chat, FunctionCall, GoogleGenAI } from "@google/genai";
```

```
import { GEMINI_API_KEY, MODEL } from "@/config";

interface AgentSessionEntry {
    chat: Chat;
    lastAccessed: number;
}

export class Agent {
    static gemini = new GoogleGenAI({
        apiKey: GEMINI_API_KEY,
    });

    static agentSessionMap = new Map<string, AgentSessionEntry>();

    static readonly SESSION_TTL = 1 * 60 * 60 * 1000; // 1 giờ

    static getOrCreateAgentSession = (user: IUserDocument): Chat => {
        const userId = user?.id;
        let sessionEntry = this.agentSessionMap.get(userId);

        if (sessionEntry) {
            const currentTime = Date.now();
            if (currentTime - sessionEntry.lastAccessed > this.SESSION_TTL) {
                this.agentSessionMap.delete(userId);
                sessionEntry = undefined;
            } else {
                sessionEntry.lastAccessed = currentTime;
                return sessionEntry.chat;
            }
        }

        const newAgentSession: Chat = this.gemini.chats.create({
            model: MODEL,
            config: {
                tools,
                responseMimeType: 'text/plain',
                systemInstruction: `Bạn là nhân viên bán món ăn của cửa hàng Việt Food
                maxOutputTokens: 500,
                topP: 0
            }
        });

        this.agentSessionMap.set(userId, {
            chat: newAgentSession,
            lastAccessed: Date.now()
        });

        return newAgentSession;
```

```
    }
}
```

Phương pháp này mang lại nhiều ưu điểm quan trọng:

- **Context tự động**: Google Gemini tự động duy trì context hội thoại, giảm thiểu việc sử dụng
- **In-memory cache**: Các phiên hội thoại (Chat Session) được lưu trữ trong Map với thời gian sống là 1 giờ
- **Tích hợp tools**: Hệ thống sử dụng các function tools (Function Calling) để Agent có thể thực hiện các tác vụ như tìm kiếm món ăn, thêm vào giỏ hàng

Các function tool quan trọng được triển khai cho hệ thống bao gồm:

- **searchDishes**: Tìm kiếm món ăn theo các tiêu chí
- **addToCart**: Thêm món ăn vào giỏ hàng
- **removeFromCart**: Xóa món ăn khỏi giỏ hàng
- **getDishesInUserCart**: Lấy danh sách món ăn trong giỏ hàng
- **getAllDishes**: Lấy toàn bộ danh sách món ăn

Việc kết hợp Google GenAI, in-memory cache, và các function tool tạo nên một hệ thống Agent thông minh có khả năng duy trì cuộc hội thoại mạch lạc với người dùng. Context được duy trì xuyên suốt cuộc hội thoại giúp Agent hiểu được ý định và mong muốn của người dùng, đồng thời thực hiện các tác vụ cần thiết như tìm kiếm món ăn và đặt hàng.

Chương 4

Đánh giá hiệu quả và đề xuất cải tiến

4.1 Đánh giá tổng thể dự án

Sau khi phân tích chi tiết các thành phần của dự án Việt Food, phần này đánh giá hiệu quả tổng thể của dự án, xem xét cả frontend và backend, đồng thời chỉ ra ưu điểm và hạn chế của cách triển khai hiện tại.

4.2 Ưu điểm của mã nguồn

4.2.1 Frontend

- **Kiến trúc component mạch lạc**: Sử dụng Atomic Design Pattern giúp code dễ bảo trì, mở rộng và tái sử dụng.
- **Tách biệt logic và UI**: Sử dụng custom hooks để tách biệt logic nghiệp vụ khỏi giao diện.
- **React Query**: Sử dụng hiệu quả để quản lý dữ liệu server-side, bao gồm cache và invalidation.
- **Tối ưu hóa hiệu suất**: Áp dụng các kỹ thuật như code splitting, lazy loading, và memoization.
- **Sử dụng Tailwind và Radix UI**: Kết hợp hiệu quả giữa utility-first CSS và headless UI components.
- **Responsive design**: Giao diện đáp ứng tốt trên các kích thước màn hình khác nhau.
- **Strong typing**: Sử dụng TypeScript và Zod giúp phát hiện lỗi sớm và đảm bảo type safety.

4.2.2 Redis Caching

- **Áp dụng hợp lý**: Cache được áp dụng cho các dữ liệu thường xuyên được truy cập nhưng ít thay đổi như thông tin món ăn và danh mục.

- **Chiến lược priming thông minh**: Việc nạp trước dữ liệu phổ biến vào cache khi khởi động server giúp tăng hiệu suất ngay từ đầu.
- **Thời gian hết hạn phù hợp**: Các key cache có thời gian hết hạn khác nhau tùy theo loại dữ liệu, ví dụ:

```
// Dữ liệu chi tiết món ăn - 24 giờ
await redis.set(cacheKey, JSON.stringify(dish), 'EX', 3600 * 24)

// Danh sách món ăn theo query - 10 phút
await redis.set(cacheKey, JSON.stringify(result), 'EX', 600)

// Món ăn phổ biến - 6 giờ
await redis.set(cacheKey, JSON.stringify(popularDishes), 'EX', 3600 * 6)
```

- **Sử dụng pipeline**: Việc sử dụng Redis pipeline khi thực hiện nhiều thao tác liên tiếp giúp giảm độ trễ mạng, tăng hiệu suất.

4.2.3 Redis Stream

- **Kiến trúc phân tách**: Tách biệt xử lý tin nhắn khỏi luồng chính của ứng dụng, giúp giảm tải cho server chính.
- **Khả năng phục hồi**: Cơ chế consumer group đảm bảo tin nhắn vẫn được xử lý ngay cả khi có lỗi.
- **Mở rộng dễ dàng**: Có thể thêm nhiều consumer để xử lý song song nếu cần.

4.2.4 HTTP Compression

- **Cấu hình cân bằng**: Cấu hình compression với ngưỡng 1KB và mức nén 6 là cân bằng hợp lý giữa hiệu quả nén và tài nguyên CPU.
- **Áp dụng toàn cục**: Compression được áp dụng như middleware cho tất cả các response, đảm bảo tính nhất quán.

4.2.5 Tối ưu hình ảnh với Sharp

- **Chuyển đổi định dạng thông minh**: Chuyển đổi sang WebP giúp giảm đáng kể kích thước file (trung bình 30-50% so với JPEG và 80% so với PNG) mà vẫn duy trì chất lượng.
- **Resize tự động**: Giới hạn kích thước hình ảnh ở mức 1000px là đủ cho hầu hết trường hợp sử dụng trên web.
- **Triển khai dưới dạng middleware**: Cách triển khai này đảm bảo tất cả hình ảnh tải lên đều được xử lý đồng nhất.

4.3 Nhược điểm và hạn chế

4.3.1 Frontend

- **Chưa có SSR/SSG**: Hiện tại Việt Food sử dụng Client-Side Rendering (CSR) với Vite, chưa áp dụng Server-Side Rendering hoặc Static Site Generation, ảnh hưởng đến SEO và thời gian tải trang ban đầu.
- **Bundle size lớn**: Mặc dù đã có code-splitting, nhưng có nhiều thư viện lớn từ Radix UI, Chart.js, và Framer Motion làm tăng kích thước bundle.
- **Quá phụ thuộc vào React Query**: Sử dụng React Query trong hầu hết các component làm tăng sự phụ thuộc vào một thư viện bên ngoài.
- **Chưa có test coverage đầy đủ**: Thiếu các unit test và integration test cho các component và custom hooks.
- **Tài nguyên hình ảnh chưa được tối ưu hoá đầy đủ**: Chưa có cơ chế lazy loading hình ảnh tự động và next-gen format (WebP, AVIF) ở phía client.

4.3.2 Redis Caching

- **Chiến lược invalidation chưa tinh tế**: Hiện tại, khi có cập nhật, dự án xóa tất cả các key cache liên quan đến danh sách món ăn. Cách tiếp cận này đơn giản nhưng có thể gây lãng phí khi một số cache vẫn có thể tái sử dụng.

```
// Xóa tất cả cache liên quan đến danh sách món ăn
const keys = await redis.keys('dishes:*')
if (keys.length > 0) {
    const pipeline = redis.pipeline()
    keys.forEach(key => pipeline.del(key))
    await pipeline.exec()
}
```

- **Thiếu cơ chế phát hiện lỗi cache**: Chưa có cơ chế fallback hoặc circuit breaker nếu Redis gặp sự cố.
- **Chưa tận dụng tối đa tính năng của Redis**: Dự án chưa sử dụng các cấu trúc dữ liệu phức tạp của Redis như sorted sets, lists để tối ưu hơn nữa.

4.3.3 Redis Stream

- **Xử lý lỗi còn đơn giản**: Khi gặp lỗi xử lý tin nhắn, hệ thống chỉ ghi log lỗi mà không có cơ chế retry phức tạp hoặc dead letter queue.
- **Thiếu cơ chế giám sát**: Chưa có metrics hoặc monitoring cho việc xử lý stream.

4.3.4 HTTP Compression

- **Áp dụng đơn điệu**: Compression được áp dụng đồng đều cho tất cả các route mà không phân biệt loại dữ liệu.
- **Thiếu header điều khiển cache**: Chưa kết hợp tốt giữa compression và HTTP caching headers.

4.3.5 Tối ưu hình ảnh với Sharp

- **Kích thước cố định**: Resize cố định ở mức 1000px có thể không phù hợp cho tất cả trường hợp sử dụng.
- **Thiếu responsive images**: Chưa tạo nhiều phiên bản hình ảnh với kích thước khác nhau cho các thiết bị khác nhau.

4.4 Đề xuất cải tiến

4.4.1 Frontend

- **Áp dụng Next.js hoặc Astro**: Chuyển sang các framework hỗ trợ SSR/SSG để cải thiện SEO và thời gian tải trang ban đầu.
- **Tree-shaking các thư viện**: Tối ưu hóa bundle size bằng cách import có chọn lọc các component từ Radix UI và các thư viện khác.
- **Tách biệt logic data-fetching**: Giảm sự phụ thuộc vào React Query bằng cách tạo lớp trung gian giữa API calls và React Query.
- **Tăng test coverage**: Thêm unit tests và integration tests cho các component và hooks chính sử dụng Jest và React Testing Library.
- **Tối ưu hóa hình ảnh**: Sử dụng next-gen formats (WebP/AVIF), responsive images và image CDN.
- **Web Vitals monitoring**: Thiết lập hệ thống theo dõi các Core Web Vitals để đánh giá hiệu suất thực tế.
- **Client-side caching**: Triển khai Service Worker và PWA để cải thiện trải nghiệm offline và giảm tải cho server.

4.4.2 Redis Caching

- **Invalidation có chọn lọc**: Thay vì xóa tất cả cache, có thể áp dụng cơ chế cache tagging để chỉ xóa những cache bị ảnh hưởng bởi thay đổi.
- **Cơ chế fallback**: Thêm try-catch và circuit breaker pattern khi tương tác với Redis.
- **Tận dụng Redis Pub/Sub**: Kết hợp với Redis Pub/Sub để thông báo invalidation cache giữa nhiều instance của ứng dụng.
- **Áp dụng caching ở nhiều lớp**: Kết hợp với HTTP caching để tối ưu hơn nữa.

4.4.3 Redis Stream

- **Consumer groups nâng cao**: Thêm nhiều consumer trong cùng một group để xử lý song song.
- **Cơ chế retry có kiểm soát**: Thêm logic để thử lại xử lý tin nhắn với giới hạn số lần và khoảng thời gian tăng dần.
- **Dead letter queue**: Chuyển tin nhắn không thể xử lý sau nhiều lần thử vào một stream riêng để kiểm tra sau.

4.4.4 HTTP Compression

- **Brotli compression**: Xem xét sử dụng Brotli thay vì gzip cho hiệu suất nén tốt hơn trên các trình duyệt hiện đại.
- **Compression có điều kiện**: Áp dụng các mức nén khác nhau cho các loại dữ liệu và endpoint khác nhau.
- **Pre-compressed assets**: Tạo sẵn phiên bản nén của các tài nguyên tĩnh.

4.4.5 Tối ưu hình ảnh với Sharp

- **Tạo nhiều kích thước**: Tạo và lưu trữ nhiều phiên bản của mỗi hình ảnh với các kích thước khác nhau.

```
// Ví dụ: tạo 3 phiên bản hình ảnh
await Promise.all([
    sharp(originalPath).resize(1200).webp({ quality: 80 }).toFile(`${outputPa
    sharp(originalPath).resize(600).webp({ quality: 80 }).toFile(`${outputPat
    sharp(originalPath).resize(300).webp({ quality: 70 }).toFile(`${outputPat
]);
```

- **Áp dụng lazy loading**: Kết hợp với frontend để áp dụng lazy loading và responsive images.
- **CDN integration**: Xem xét sử dụng CDN chuyên biệt cho hình ảnh với khả năng xử lý hình ảnh theo yêu cầu.

Chương 5

Kết luận và hướng phát triển

5.1 Tóm tắt kết quả phân tích

Qua quá trình phân tích dự án Việt Food, chúng tôi đã khảo sát chi tiết các kỹ thuật tối ưu backend được áp dụng trong dự án. Những kỹ thuật này đóng vai trò quan trọng trong việc nâng cao hiệu suất, đảm bảo khả năng phục vụ và tạo trải nghiệm người dùng tốt hơn. Tóm tắt các phát hiện chính:

- **Redis Caching**: Dự án đã triển khai một hệ thống cache toàn diện cho dữ liệu món ăn và danh mục, với chiến lược priming thông minh khi khởi động server. Cách tiếp cận này giúp giảm đáng kể thời gian phản hồi và tải trên database, đặc biệt hữu ích với dữ liệu món ăn - loại dữ liệu được truy cập thường xuyên nhưng ít thay đổi.
- **Redis Stream**: Việc sử dụng Redis Stream để xử lý tin nhắn theo mô hình producer-consumer đã tách biệt được các tác vụ xử lý nặng khỏi luồng chính của ứng dụng. Cách thiết kế này giúp hệ thống có khả năng mở rộng và phục hồi tốt hơn.
- **HTTP Compression**: Compression middleware được áp dụng toàn cục với cấu hình cân bằng giữa hiệu suất nén và tài nguyên CPU. Kỹ thuật này giúp giảm đáng kể kích thước dữ liệu truyền tải, cải thiện thời gian tải trang.
- **Tối ưu hình ảnh với Sharp**: Dự án sử dụng Sharp để xử lý và tối ưu hóa hình ảnh tải lên, chuyển đổi sang định dạng WebP hiệu quả và resize phù hợp. Cách tiếp cận này giúp tiết kiệm không gian lưu trữ và băng thông.

Nhìn chung, Việt Food đã áp dụng các kỹ thuật tối ưu backend một cách hợp lý và hiệu quả. Các kỹ thuật này không chỉ hoạt động độc lập mà còn bổ trợ cho nhau, tạo nên một hệ thống có hiệu suất cao và khả năng phục vụ tốt.

5.2 Bài học kinh nghiệm

Quá trình phân tích dự án Việt Food đã mang lại nhiều bài học quý báu về tối ưu hóa backend trong phát triển ứng dụng web:

5.2.1 Thiết kế hướng hiệu suất ngay từ đầu

Việt Food đã áp dụng các kỹ thuật tối ưu ngay từ giai đoạn thiết kế ban đầu, chứ không phải như một giải pháp bổ sung sau này. Điều này cho thấy tầm quan trọng của việc xem xét các vấn đề hiệu suất ngay từ khi bắt đầu dự án.

5.2.2 Caching là chìa khóa

Redis caching được triển khai một cách toàn diện trong dự án, minh họa rằng chiến lược caching tốt có thể mang lại lợi ích hiệu suất đáng kể. Đặc biệt, việc xác định đúng những dữ liệu nào nên được cache (như dữ liệu món ăn) và cách quản lý vòng đời cache là rất quan trọng.

5.2.3 Phân tách xử lý không đồng bộ

Sử dụng Redis Stream để tách biệt xử lý tin nhắn là một ví dụ tốt về cách phân tách các tác vụ nặng và không đồng bộ khỏi luồng chính của ứng dụng. Cách tiếp cận này giúp ứng dụng vẫn có thể phản hồi nhanh chóng ngay cả khi xử lý các tác vụ phức tạp.

5.2.4 Tối ưu tài nguyên tĩnh

Việc sử dụng Sharp để tối ưu hình ảnh và compression để giảm kích thước dữ liệu truyền tải cho thấy tầm quan trọng của việc tối ưu tài nguyên tĩnh. Những kỹ thuật này có thể mang lại cải thiện hiệu suất đáng kể với chi phí triển khai tương đối thấp.

5.2.5 Cân bằng giữa hiệu suất và phức tạp

Dự án Việt Food đã chọn cách triển khai hợp lý, cân bằng giữa hiệu suất và độ phức tạp của mã nguồn. Một số kỹ thuật tối ưu có thể được cải tiến thêm, nhưng cách tiếp cận hiện tại đã mang lại lợi ích đáng kể mà không làm tăng quá mức độ phức tạp của hệ thống.

5.3 Hướng phát triển tiềm năng

Dựa trên phân tích của chúng tôi, có một số hướng phát triển tiềm năng để nâng cao hơn nữa hiệu suất và khả năng mở rộng của Việt Food:

5.3.1 Kiến trúc Microservices

Với quy mô ngày càng tăng, Việt Food có thể xem xét chuyển từ kiến trúc monolithic hiện tại sang kiến trúc microservices. Điều này sẽ cho phép:

- Phát triển và triển khai độc lập các thành phần
- Mở rộng có chọn lọc các dịch vụ cần thiết
- Cô lập lỗi tốt hơn

Ví dụ, có thể tách thành các service riêng biệt cho quản lý món ăn, xử lý đơn hàng, hệ thống tin nhắn, v.v.

5.3.2 Caching đa tầng

Phát triển hệ thống cache thành nhiều tầng:

- **Client-side caching**: Áp dụng HTTP caching với ETag và Cache-Control headers
- **CDN caching**: Triển khai CDN cho tài nguyên tĩnh
- **API Gateway caching**: Cache ở tầng API Gateway
- **Application caching**: Tiếp tục sử dụng Redis, nhưng với chiến lược cache invalidation tinh tế hơn
- **Database caching**: Query caching ở tầng database

5.3.3 Xử lý ảnh nâng cao

Cải tiến hệ thống xử lý ảnh:

- **Responsive images**: Tạo nhiều phiên bản của mỗi hình ảnh cho các thiết bị khác nhau
- **Tích hợp CDN chuyên biệt**: Sử dụng CDN có khả năng xử lý hình ảnh theo yêu cầu
- **AVIF format**: Áp dụng định dạng AVIF mới, hiệu quả hơn cả WebP
- **Lazy loading và progressive loading**: Tối ưu tải hình ảnh

5.3.4 Tích hợp GraphQL

GraphQL có thể là một bổ sung hữu ích cho API RESTful hiện tại:

- Cho phép client chỉ định chính xác dữ liệu cần thiết
- Giảm over-fetching và under-fetching
- Hỗ trợ tốt cho các ứng dụng mobile với kết nối mạng không ổn định

5.3.5 Giám sát và phân tích hiệu suất

Cải thiện hệ thống giám sát:

- **APM (Application Performance Monitoring)**: Triển khai công cụ giám sát hiệu suất ứng dụng
- **Metrics collection**: Thu thập metrics về cache hit/miss ratio, thời gian phản hồi, v.v.
- **Distributed tracing**: Theo dõi các request qua nhiều service
- **Tự động điều chỉnh**: Điều chỉnh cấu hình cache, số lượng worker dựa trên metrics

5.3.6 Serverless Computing

Xem xét chuyển một số thành phần sang mô hình serverless:

- **Image processing**: Chuyển xử lý hình ảnh sang serverless functions
- **Periodic tasks**: Các tác vụ định kỳ như cập nhật cache
- **Event-driven processing**: Xử lý sự kiện như đơn hàng mới, đánh giá mới

Việt Food đã có nền tảng tốt với các kỹ thuật tối ưu hiện tại. Việc áp dụng các hướng phát triển này sẽ giúp dự án tiếp tục mở rộng quy mô và duy trì hiệu suất cao trong tương lai.