**TRƯỜNG ĐẠI HỌC AN GIANG  
KHOA CÔNG NGHỆ THÔNG TIN**

****

**Project**

**Development and Deployment of a Multi-Service Website:**

**URBAN FLOW- Shipping, Delivery & Sales**

**Using  
MERN STACK VÀ NEXT.JS**

**STUDENT:**

**DTH225650 - Huỳnh Quốc Huy**

**AN GIANG, August 2025**

**TRƯỜNG ĐẠI HỌC AN GIANG**

**KHOA CÔNG NGHỆ THÔNG TIN**

****

**Project**

**Development and Deployment of a Multi-Service Website:**

**URBAN FLOW- Shipping,Delivery & Sales  
Using  
MERN STACK VÀ NEXT.JS**

**STUDENT:**

**DTH225650 - Huỳnh Quốc Huy**

**AN GIANG, August 2025**

.............................................................................................................

**Giảng viên hướng dẫn**

( Ký và ghi rõ họ tên)

Nội dung nhận xét:

* **Đồng ý** hay **Không đồng ý**  cho sinh viên báo cáo TTCK; Nếu không đồng ý cần ghi rõ lý do.
* Kết quả được so với yêu cầu;
* Ý kiến khác (nếu có)

**MỤC LỤC**

[**1.** **GIỚI THIỆU TỔNG QUAN** 4](#_Toc205832413)

[**1.1.** **GIỚI THIỆU VỀ QUY TRÌNH PHÁT TRIỂN WEBSITE** 4](#_Toc205832414)

[**1.2.** **TỔNG QUAN VỀ MERN STACK** 4](#_Toc205832415)

[**1.3.** **SƠ ĐỒ KIẾN TRÚC TỔNG THỂ CỦA ỨNG DỤNG** 5](#_Toc205832416)

[**1.3.1.** **Lớp Client (Frontend)**: 5](#_Toc205832417)

[**1.3.2.** **Lớp Server (Backend)**: 5](#_Toc205832418)

[**1.3.3.** **Lớp Cơ sở dữ liệu (Database)**: 6](#_Toc205832419)

[**1.3.4.** **Tích hợp Thời gian thực (Real-time)**: 6](#_Toc205832420)

[**2.** **FRONTEND – XÂY DỰNG GIAO DIỆN NGƯỚI DÙNG VỚI REACTJS VÀ NEXT.JS** 6](#_Toc205832421)

[**2.1.** **REACTJS** 6](#_Toc205832422)

[**2.1.1.** **Tổng Quan Về Thư Viện Reactjs** 6](#_Toc205832423)

[**2.1.2.** **Kiến Trúc Component, Virtual Dom (Virtual Docment Object Model)** 7](#_Toc205832424)

[**2.1.3.** **Ưu Và Nhược Điểm Của Reactjs** 7](#_Toc205832425)

[**2.1.4.** **Các Khái Niệm Mở Rộng: Jsx, State, Props, Hooks** 8](#_Toc205832426)

[**2.2.** **NEXT.JS** 10](#_Toc205832427)

[**2.2.1.** **Tổng Quan Về Framework Next.Js** 10](#_Toc205832428)

[**2.2.2.** **SSR (Server-Side Rendering), SSG (Static Site Generation), ISR (Incremental Static Regeneration)** 10](#_Toc205832429)

[**2.2.3.** **Ưu Và Nhược Điểm Của Next.Js So Với Reactjs Thuần** 11](#_Toc205832430)

[**2.2.4.** **Quy Trình Triển Khai: Cấu Trúc Thư Mục, Hệ Thống Routing, Fetching Dữ Liệu Với Getstaticprops Và Getserversideprops** 12](#_Toc205832431)

[**3.** **BACKEND – XÂY DỰNG MÁY CHỦ VỚI NODEJS VÀ EXPRESS.JS** 13](#_Toc205832432)

[**3.1.** **NODEJS** 13](#_Toc205832433)

[**3.1.1.** **Tổng Quan Về Môi Trường Runtime NodeJS** 13](#_Toc205832434)

[**3.1.2.** **Kiến Trúc Bất Đồng Bộ (Non-Blocking I/O) Và Event Loop** 13](#_Toc205832435)

[**3.1.3.** **Ưu Và Nhược Điểm Của Nodejs** 13](#_Toc205832436)

[**3.1.4.** **Các Khái Niệm Mở Rộng: NPM (Node Package Manager), Modules** 14](#_Toc205832437)

[**3.2.** **EXPRESS.JS** 15](#_Toc205832438)

[**3.2.1.** **Tổng Quan Về Framwork Express.js** 15](#_Toc205832439)

[**3.2.2.** **Middleware, Routing** 15](#_Toc205832440)

[**3.2.3.** **Ưu và nhược điểm của Express.js** 16](#_Toc205832441)

[**3.2.4.** **Quy Trình Triển Khai: Cài Đặt Và Cấu Hình Cơ Bản, Xây Dựng Các Api Restful** 16](#_Toc205832442)

[**4.** **DATABASE – Quản Lý Cơ Sở Dữ Liệu (MongoDB)** 17](#_Toc205832443)

[**4.1.** **Tổng Quan Về MongoDB** 17](#_Toc205832444)

[**4.1.1.** **Giới thiệu Về Cơ Sở Dữ Liệu NoSQL** 17](#_Toc205832445)

[**4.1.2.** **Ưu Và Nhược Điểm Của MongoDB So Với SQL** 17](#_Toc205832446)

[**4.1.3.** **Mô Hình Dữ Liệu: Document, Collection** 18](#_Toc205832447)

[**4.2.** **QUY TRÌNH TÍCH HỢP VỚI BACKEND** 19](#_Toc205832448)

[**4.2.1.** **Cài Đặt Và Cấu Hình Mongodb** 19](#_Toc205832449)

[**4.2.2.** **Sử dụng Mongoose (Thư Viện ODM) Để Tương Tác với MongoDB** 19](#_Toc205832450)

[**4.2.3.** **Các Thao Tác Cơ Bản (CRUD: Create, Read, Update, Delete)** 20](#_Toc205832451)

[**5.** **QUY TRÌNH TRIỂN KHAI DỰ ÁN (DEPLOYMENT)** 21](#_Toc205832452)

[**5.1.** **XÂY DỰNG ỨNG DỤNG HOÀN CHỈNH** 21](#_Toc205832453)

[**5.1.1.** **Tích Hợp Frontend (Next.js) và Backend (Node.js + Express.js)** 21](#_Toc205832454)

[**5.1.2.** **Tối Ưu Hóa Hiện Suất Và Bảo Mật** 22](#_Toc205832455)

[**5.2.** **TRIỂN KHAI LÊN MÁY CHỦ** 22](#_Toc205832456)

[**5.2.1.** **Chuẩn Bị Môi Trường Máy Chủ (Ví dụ: Vercel, Heroku, AWS, Google Cloud).** 22](#_Toc205832457)

[**5.2.2.** **Các Bước Triển Khai: Cấu Hình Môi Trường, Chạy Ứng Dụng, Quản Lý Domain** 24](#_Toc205832458)

[**5.3.** **CÁC VẤN ĐỀ SAU TRIỂN KHAI** 25](#_Toc205832459)

[**5.3.1.** **Giám sát hiệu suất ứng dụng** 25](#_Toc205832460)

[**5.3.2.** **Sao lưu và phục hồi dữ liệu.** 25](#_Toc205832461)

1. **GIỚI THIỆU TỔNG QUAN**
   1. **GIỚI THIỆU VỀ QUY TRÌNH PHÁT TRIỂN WEBSITE**

Trong kỷ nguyên số hóa hiện nay, website không chỉ đơn thuần là một công cụ cung cấp thông tin mà còn là một nền tảng dịch vụ phức tạp, đòi hỏi sự kết hợp chặt chẽ của nhiều công nghệ để mang lại trải nghiệm tốt nhất cho người dùng. Quy trình phát triển một ứng dụng web hiện đại thường được phân chia thành hai phần chính, độc lập nhưng luôn tương tác với nhau: Frontend và Backend. Sự phân tách này không chỉ giúp việc quản lý dự án trở nên dễ dàng hơn mà còn cho phép các nhóm phát triển làm việc song song, đẩy nhanh tốc độ hoàn thành. .

**Frontend (Phía người dùng):** Là toàn bộ giao diện và các tương tác mà người dùng nhìn thấy và sử dụng trực tiếp trên trình duyệt. Frontend chịu trách nhiệm về giao diện, bố cục, màu sắc, phông chữ và các yếu tố tương tác. Mục tiêu chính của frontend là tạo ra một trải nghiệm người dùng (UX) trực quan, mượt mà và hấp dẫn. Các công nghệ chủ đạo trong frontend bao gồm HTML, CSS, JavaScript và một vài thư viện và framwork frontend nâng cao như ReactJS, Next.js, Vue.js, AngularJS…

**Backend (Phía máy chủ):** Là "bộ não" của ứng dụng, hoạt động ẩn sau frontend. Backend chịu trách nhiệm xử lý logic nghiệp vụ, tương tác với cơ sở dữ liệu, quản lý xác thực và quyền truy cập của người dùng, và cung cấp dữ liệu cho frontend thông qua các giao diện lập trình ứng dụng (API). Các công nghệ backend phổ biến bao gồm Node.js, Express.js, PHP, Python, Java, v.v.

Dự án này tuân thủ một quy trình phát triển chuyên nghiệp, áp dụng các phương pháp hiện đại để xây dựng một nền tảng dịch vụ đa năng. Chúng tôi tập trung vào việc tạo ra một hệ thống có tính tương tác cao, hiệu suất vượt trội và khả năng mở rộng tốt để đáp ứng các nhu cầu phức tạp của người dùng trong lĩnh vực giao hàng, vận chuyển và thương mại điện tử.

* 1. **TỔNG QUAN VỀ MERN STACK**

Để hiện thực hóa các mục tiêu của dự án, chúng tôi đã lựa chọn sử dụng **MERN Stack**, một bộ công nghệ mã nguồn mở dựa trên JavaScript, và tích hợp thêm framework Next.js để tối ưu hóa hiệu suất. MERN Stack là viết tắt của **MongoDB, Express.js, ReactJS, Node.js**. Sự đồng nhất về ngôn ngữ lập trình (JavaScript) trên toàn bộ stack là một lợi thế lớn, giúp việc phát triển trở nên liền mạch và hiệu quả hơn.

* **MongoDB**: Là một cơ sở dữ liệu NoSQL (Not Only SQL) hàng đầu. Thay vì lưu trữ dữ liệu trong các bảng với cấu trúc cố định như các cơ sở dữ liệu quan hệ (SQL), MongoDB lưu trữ dữ liệu dưới dạng các tài liệu (document) theo định dạng BSON (binary JSON), cho phép cấu trúc dữ liệu linh hoạt và dễ dàng thay đổi. Điều này đặc biệt hữu ích cho các ứng dụng có dữ liệu đa dạng và thường xuyên biến động như thông tin đơn hàng, sản phẩm và vị trí địa lý.
* **Express.js**: Là một framework backend tối giản và linh hoạt, được xây dựng trên nền tảng Node.js. Express.js cung cấp một bộ công cụ mạnh mẽ để xây dựng các API RESTful, quản lý routing (định tuyến các yêu cầu) và middleware (các hàm xử lý trung gian). Nhờ có Express.js, việc xây dựng backend trở nên nhanh chóng và có tổ chức hơn rất nhiều so với việc sử dụng Node.js thuần.
* **ReactJS**: Là một thư viện JavaScript được phát triển bởi Facebook, chuyên dùng để xây dựng giao diện người dùng. ReactJS tập trung vào việc tạo ra các component giao diện có thể tái sử dụng, giúp quá trình phát triển trở nên hiệu quả và dễ bảo trì. Điểm nổi bật của ReactJS là việc sử dụng **Virtual DOM (Document Object Model ảo)**, giúp tối ưu hóa hiệu suất bằng cách chỉ cập nhật những thay đổi cần thiết lên DOM thật.
* **Node.js**: Là môi trường runtime JavaScript mã nguồn mở, cho phép thực thi mã JavaScript ở phía máy chủ. Điểm mạnh lớn nhất của Node.js là kiến trúc **bất đồng bộ và hướng sự kiện (event-driven, non-blocking I/O)**. Điều này giúp Node.js có thể xử lý hàng nghìn yêu cầu đồng thời một cách hiệu quả, rất phù hợp với các ứng dụng có tính năng thời gian thực như theo dõi vị trí tài xế hay thông báo đơn hàng.

**Sự kết hợp giữa MERN Stack và Next.js** tạo nên một hệ sinh thái phát triển mạnh mẽ, đáp ứng được các yêu cầu về hiệu suất, khả năng mở rộng, và trải nghiệm người dùng cao cấp.

* 1. **SƠ ĐỒ KIẾN TRÚC TỔNG THỂ CỦA ỨNG DỤNG**

Kiến trúc của ứng dụng **URBAN FLOW** được thiết kế theo mô hình **client-server**, với sự phân tách rõ ràng giữa các lớp logic để đảm bảo tính module hóa và dễ bảo trì. Kiến trúc này có thể được mô tả qua các thành phần chính sau:

* + 1. **Lớp Client (Frontend)**:
  + Sử dụng **Next.js** và **ReactJS** để xây dựng giao diện người dùng tương tác cao.
  + Các component React chịu trách nhiệm hiển thị thông tin sản phẩm, bản đồ, form đặt hàng và trạng thái đơn hàng.
  + Tận dụng các tính năng của Next.js như **SSR** và **SSG** để tối ưu hóa thời gian tải trang và khả năng SEO.
  + Gửi các yêu cầu HTTP (GET, POST, PUT, DELETE) đến backend thông qua các API để lấy và gửi dữ liệu.
    1. **Lớp Server (Backend)**:
  + Sử dụng **Node.js** làm môi trường thực thi và **Express.js** để tạo các API.
  + Lớp này bao gồm các **API routes** (định tuyến API) để xử lý các yêu cầu từ frontend. Ví dụ: /api/orders để quản lý đơn hàng, /api/products để quản lý sản phẩm.
  + Tích hợp các **middleware** để xử lý các tác vụ như xác thực người dùng (authentication), kiểm tra dữ liệu đầu vào và xử lý lỗi.
  + Lớp server cũng đóng vai trò cầu nối giữa frontend và cơ sở dữ liệu.
    1. **Lớp Cơ sở dữ liệu (Database)**:
  + Sử dụng **MongoDB** để lưu trữ tất cả dữ liệu của ứng dụng.
  + Các collection chính bao gồm users (thông tin người dùng và tài xế), products (thông tin sản phẩm), và orders (chi tiết đơn hàng, trạng thái, vị trí).
  + MongoDB đặc biệt hữu ích với các **truy vấn không gian địa lý (geospatial queries)**, cho phép tìm kiếm tài xế gần nhất với vị trí của người dùng một cách hiệu quả.
    1. **Tích hợp Thời gian thực (Real-time)**:
  + Sử dụng thư viện **Socket.IO** trên nền tảng Node.js để thiết lập các kết nối WebSocket.Các kết nối này cho phép cập nhật vị trí của tài xế trên bản đồ, thông báo trạng thái đơn hàng và tin nhắn giữa người dùng và tài xế một cách tức thời, tạo nên trải nghiệm mượt mà và liền mạch.

Kiến trúc này đảm bảo rằng ứng dụng không chỉ mạnh mẽ mà còn có khả năng mở rộng linh hoạt, cho phép tích hợp thêm nhiều tính năng mới trong tương lai một cách dễ dàng.

1. **FRONTEND – XÂY DỰNG GIAO DIỆN NGƯỚI DÙNG VỚI REACTJS VÀ NEXT.JS**
   1. **REACTJS**
      1. **Tổng Quan Về Thư Viện Reactjs**

ReactJS là một thư viện JavaScript mã nguồn mở được phát triển bởi Facebook, đóng vai trò cốt lõi trong việc xây dựng giao diện người dùng (UI) cho các ứng dụng web một trang (Single-Page Applications - SPA) hoặc các ứng dụng web phức tạp. Thay vì là một framework toàn diện, ReactJS tập trung vào "view layer" của mô hình MVC (Model-View-Controller), cho phép các nhà phát triển tạo ra các thành phần giao diện (UI components) có khả năng tái sử dụng. Điều này giúp quá trình phát triển trở nên hiệu quả, có cấu trúc và dễ bảo trì hơn.

Trong dự án URBAN FLOW, ReactJS đóng vai trò chủ đạo trong việc xây dựng toàn bộ giao diện người dùng, từ các trang tĩnh như giới thiệu, liên hệ đến các màn hình phức tạp và tương tác cao như bản đồ theo dõi đơn hàng, form đặt dịch vụ và trang thương mại điện tử.

* + 1. **Kiến Trúc Component, Virtual Dom (Virtual Docment Object Model)**
* **Kiến trúc Component:**

ReactJS được xây dựng trên một kiến trúc dựa trên component. Mỗi thành phần của giao diện người dùng, dù là nhỏ nhất như một nút bấm hay phức tạp như một bản đồ tương tác, đều được xem là một component độc lập. Các component này có thể được kết hợp với nhau để tạo thành các giao diện phức tạp hơn. Ví dụ, một trang sản phẩm có thể bao gồm các components như Header, ProductImage, ProductDetails, ReviewSection và Footer.

* **Ưu điểm của kiến trúc component:**

**Tái sử dụng:** Các component có thể được sử dụng lại ở nhiều nơi khác nhau, giúp giảm thiểu việc viết lại mã.

**Dễ quản lý:** Việc phân chia giao diện thành các component nhỏ giúp việc quản lý, debug và phát triển tính năng mới trở nên dễ dàng hơn. Mỗi component chỉ chịu trách nhiệm cho một phần cụ thể của giao diện, giúp mã nguồn trở nên có cấu trúc và dễ hiểu.

Tính module hóa: Các component có thể được phát triển và kiểm thử độc lập, sau đó mới được tích hợp vào ứng dụng.

* **Virtual DOM (Virtual Document Object Model):**

Virtual DOM là một trong những tính năng cốt lõi giúp ReactJS đạt được hiệu suất cao. DOM (Document Object Model) là một cây cấu trúc dữ liệu của các phần tử trên trang web, và việc thay đổi DOM thật là một thao tác tốn kém về tài nguyên và làm chậm ứng dụng.

**Cơ chế hoạt động:** Khi có bất kỳ thay đổi nào trong dữ liệu, thay vì cập nhật trực tiếp lên DOM thật, React tạo ra một bản sao ảo của DOM trong bộ nhớ. Sau đó, nó so sánh (diffing) bản sao Virtual DOM mới này với bản cũ để xác định những thay đổi tối thiểu. Cuối cùng, React chỉ cập nhật những thay đổi đó (cơ chế reconciliation) lên DOM thật.

**Lợi ích:** Cơ chế này giúp giảm thiểu đáng kể số lần thao tác với DOM thật, từ đó tăng tốc độ hiển thị và mang lại trải nghiệm người dùng mượt mà, đặc biệt quan trọng đối với các ứng dụng có tính năng động, cập nhật dữ liệu liên tục như theo dõi vị trí trên bản đồ hay cập nhật trạng thái đơn hàng.

* + 1. **Ưu Và Nhược Điểm Của Reactjs**
* **Ưu điểm:**

**Tăng tốc độ phát triển:** Nhờ kiến trúc component có thể tái sử dụng, việc xây dựng các giao diện phức tạp trở nên nhanh chóng hơn.

**Hiệu suất cao:** Cơ chế Virtual DOM giúp tối ưu hóa hiệu suất render, mang lại trải nghiệm mượt mà cho người dùng.

**Cộng đồng lớn và mạnh mẽ:** React có một cộng đồng hỗ trợ khổng lồ, với vô số thư viện, tài liệu và các công cụ phát triển, giúp việc học hỏi và giải quyết vấn đề trở nên dễ dàng.

**Hệ sinh thái phong phú:** Có rất nhiều thư viện được xây dựng xung quanh React, từ quản lý trạng thái (Redux, Zustand) đến các thư viện UI (Material UI, Ant Design).

* **Nhược điểm:**

**Chỉ là thư viện:** ReactJS chỉ tập trung vào phần giao diện, nên để xây dựng một ứng dụng hoàn chỉnh, chúng tôi cần tích hợp thêm các thư viện khác (ví dụ: react-router-dom cho routing) hoặc sử dụng một framework như Next.js để có được một hệ thống hoàn chỉnh hơn.

**Tốc độ cập nhật nhanh:** React và hệ sinh thái của nó phát triển rất nhanh, đôi khi đòi hỏi các nhà phát triển phải liên tục cập nhật kiến thức.

* + 1. **Các Khái Niệm Mở Rộng: Jsx, State, Props, Hooks**

Trong quá trình xây dựng ứng dụng web URBAN FLOW bằng ReactJS, việc nắm vững các khái niệm cốt lõi sau là vô cùng quan trọng. Đây là những nền tảng giúp chúng tôi tạo ra các giao diện động, tương tác và dễ bảo trì.

* **JSX (JavaScript XML)**

JSX là một cú pháp mở rộng của JavaScript, cho phép chúng tôi viết mã HTML/XML ngay bên trong file JavaScript. Mặc dù không phải là một ngôn ngữ lập trình độc lập, JSX giúp việc xây dựng giao diện trở nên trực quan và dễ đọc hơn rất nhiều so với việc sử dụng React.createElement(). Về cơ bản, JSX chỉ là một "đường cú pháp" (syntactic sugar) cho phép biểu diễn các đối tượng React. Các trình duyệt không thể hiểu trực tiếp JSX, do đó nó cần được một bộ transpiler (chẳng hạn như Babel) chuyển đổi thành JavaScript thuần trước khi chạy.

* + **Ví dụ:** 
    - Thay vì viết:

**React**.**createElement**(**'h1'**, **null**, **'Hello**, **URBAN** **FLOW!**');

* + - Chúng ta viết trực tiếp trong JSX:

<**h1**>**Hello**, **URBAN** **FLOW**!<**/h1**>

Trong dự án, JSX được sử dụng rộng rãi để tạo các component, từ các thẻ cơ bản như <**div**>, <**p**> đến các component phức tạp như <**ProductCard**> hay <**MapComponent**>.

* **State (Trạng thái)**

State là một đối tượng JavaScript bên trong một component, chứa dữ liệu có thể thay đổi. Khi dữ liệu trong state thay đổi, React sẽ tự động kích hoạt quá trình render lại component để cập nhật giao diện người dùng. State giúp các component có tính tương tác và động.

Trong dự án URBAN FLOW, state được sử dụng để quản lý nhiều loại dữ liệu khác nhau.

**Ví dụ**:

* + Vị trí hiện tại của người dùng trên bản đồ.
  + Danh sách sản phẩm trong giỏ hàng.
  + Trạng thái của một đơn hàng (đang chờ, đang giao, đã hoàn thành).
  + Dữ liệu từ các form nhập liệu (tên, số điện thoại, địa chỉ).
* **Props (Properties)**

Props là một đối tượng chỉ đọc được sử dụng để truyền dữ liệu từ component cha xuống component con. Props giúp các component con có thể hiển thị dữ liệu động mà không cần phải tự quản lý state của nó. Đây là cơ chế chính để các component giao tiếp với nhau trong React. Quy trình truyền dữ liệu một chiều này (unidirectional data flow) giúp việc gỡ lỗi trở nên dễ dàng và dự đoán được.

**Ví dụ:** Một component OrderTracking (theo dõi đơn hàng) có thể truyền các props như orderId, status, driverLocation xuống một component con là MapDisplay để hiển thị thông tin chi tiết trên bản đồ.

<**MapDisplay** **driverLocation**={**driverLocation**} />

* **Hooks**

Hooks là các hàm đặc biệt được giới thiệu trong React 16.8, cho phép các functional component sử dụng các tính năng của class component như quản lý state và xử lý các tác vụ vòng đời. Sự ra đời của Hooks đã làm thay đổi đáng kể cách chúng ta viết React, giúp mã nguồn trở nên gọn gàng, dễ đọc và dễ bảo trì hơn.

Trong dự án URBAN FLOW, chúng tôi sử dụng các Hooks sau:

* + useState(): Hook này cho phép thêm state vào functional component. Ví dụ, chúng tôi sử dụng useState để lưu trữ dữ liệu sản phẩm, thông tin người dùng hoặc trạng thái hiển thị của các modal.
  + useEffect(): Hook này dùng để xử lý các tác vụ phụ (side effects) như gọi API từ backend, đăng ký các sự kiện hoặc thay đổi DOM. Đây là công cụ hữu ích để fetch dữ liệu sản phẩm khi component được mount hoặc cập nhật vị trí của tài xế theo thời gian thực.
  + useContext(): Hook này cho phép các component truy cập vào các giá trị từ context mà không cần phải truyền props qua nhiều cấp. Trong dự án, useContext có thể được dùng để truy cập thông tin người dùng đang đăng nhập trên toàn bộ ứng dụng.
  1. **NEXT.JS**
     1. **Tổng Quan Về Framework Next.Js**

**Next.js** là một **framework React** mã nguồn mở được phát triển bởi Vercel, được thiết kế để giải quyết những hạn chế của ReactJS thuần túy. Trong khi ReactJS là một thư viện tập trung vào việc xây dựng giao diện người dùng, Next.js cung cấp một nền tảng toàn diện để xây dựng các ứng dụng web với đầy đủ các tính năng như định tuyến (routing), tối ưu hóa hiệu suất và SEO (tối ưu hóa công cụ tìm kiếm). Nó cho phép các nhà phát triển tạo ra các ứng dụng web phức tạp một cách dễ dàng và hiệu quả hơn.

**Next.js** hoạt động dựa trên Node.js và tận dụng các tính năng mạnh mẽ của React. Nó cung cấp một cấu trúc mặc định, giúp giảm thiểu thời gian cấu hình ban đầu, cho phép các nhà phát triển tập trung vào việc xây dựng logic nghiệp vụ. Đối với dự án **URBAN FLOW**, việc sử dụng Next.js là một quyết định chiến lược để đảm bảo ứng dụng có hiệu suất cao, thời gian tải trang nhanh và thân thiện với các công cụ tìm kiếm, điều rất quan trọng đối với một nền tảng đa dịch vụ.

* + 1. **SSR (Server-Side Rendering), SSG (Static Site Generation), ISR (Incremental Static Regeneration)**

**Next.js** cung cấp các phương pháp render trang linh hoạt, giúp tối ưu hóa hiệu suất và trải nghiệm người dùng.

* **Server-Side Rendering (SSR):**

SSR là kỹ thuật render trang web ở phía máy chủ cho mỗi yêu cầu của người dùng. Khi người dùng truy cập một URL, máy chủ sẽ xử lý dữ liệu và render toàn bộ trang HTML, sau đó gửi về trình duyệt.

**Cơ chế hoạt động:** Next.js sử dụng hàm getServerSideProps để fetch dữ liệu từ API hoặc database ở phía máy chủ trước khi trang được render.

**Ưu điểm:** Đảm bảo người dùng luôn nhận được nội dung mới nhất. SSR rất quan trọng cho các trang có dữ liệu thay đổi thường xuyên, như trang theo dõi đơn hàng của URBAN FLOW, nơi thông tin cần được cập nhật liên tục.

**Nhược điểm:** Tốc độ phản hồi có thể chậm hơn SSG vì trang phải được render lại cho mỗi yêu cầu.

* **Static Site Generation (SSG):**

SSG là phương pháp render các trang web thành các file HTML tĩnh trong quá trình build ứng dụng. Sau khi build, các file này sẽ được lưu trữ và phục vụ trực tiếp từ CDN (Content Delivery Network).

**Cơ chế hoạt động:** Next.js sử dụng hàm getStaticProps để fetch dữ liệu trong quá trình build.

**Ưu điểm:** Tốc độ tải trang cực nhanh vì không cần render lại cho mỗi yêu cầu. SSG lý tưởng cho các trang có nội dung ít thay đổi, như trang giới thiệu, chính sách, hoặc các trang sản phẩm trong một cửa hàng trực tuyến mà dữ liệu không thay đổi quá thường xuyên.

**Nhược điểm:** Nội dung sẽ không được cập nhật cho đến khi ứng dụng được build lại.

* **Incremental Static Regeneration (ISR):**

ISR là sự kết hợp giữa SSR và SSG. Nó cho phép các trang tĩnh được cập nhật sau khi ứng dụng đã được deploy mà không cần build lại toàn bộ.

**Cơ chế hoạt động:** Khi sử dụng ISR, bạn có thể thiết lập một khoảng thời gian (ví dụ: revalidate: 60) trong getStaticProps. Next.js sẽ kiểm tra lại dữ liệu sau khoảng thời gian đó. Nếu có dữ liệu mới, nó sẽ render lại trang tĩnh và cập nhật cache.

**Ưu điểm:** Cung cấp hiệu suất cao như SSG nhưng vẫn đảm bảo nội dung luôn được cập nhật, rất phù hợp cho các trang blog hoặc danh mục sản phẩm của URBAN FLOW.

* + 1. **Ưu Và Nhược Điểm Của Next.Js So Với Reactjs Thuần**

Việc lựa chọn Next.js thay vì ReactJS thuần cho dự án URBAN FLOW mang lại nhiều lợi thế nhưng cũng đi kèm một số nhược điểm:

* **Ưu điểm:**

Tối ưu SEO: Next.js giải quyết vấn đề SEO lớn của ReactJS thuần (tối ưu SEO cho SPA rất khó) bằng cách render trang ở phía server, giúp các công cụ tìm kiếm dễ dàng crawl và index nội dung.

Hiệu suất vượt trội: Nhờ SSR, SSG và code splitting tự động, ứng dụng Next.js có thời gian tải trang ban đầu nhanh hơn đáng kể.

Hệ thống Routing đơn giản: Next.js có hệ thống routing dựa trên tệp (file-based routing) rất trực quan, không cần phải cài đặt và cấu hình thư viện bên ngoài như React Router.

API Routes: Next.js cho phép bạn xây dựng các API backend ngay trong dự án, giúp việc phát triển toàn diện (full-stack) trở nên dễ dàng hơn.

* **Nhược điểm:**

Hạn chế sự linh hoạt: Next.js có một số quy tắc và cấu trúc mặc định, có thể hạn chế sự linh hoạt trong một số trường hợp.

Đường cong học tập ban đầu: Đối với những người mới chỉ biết ReactJS, việc làm quen với các khái niệm như SSR, SSG, getServerSideProps có thể mất một chút thời gian.

* + 1. **Quy Trình Triển Khai: Cấu Trúc Thư Mục, Hệ Thống Routing, Fetching Dữ Liệu Với Getstaticprops Và Getserversideprops**
* **Cấu trúc thư mục:** Next.js có một cấu trúc thư mục tiêu chuẩn, giúp dự án của chúng tôi có tổ chức ngay từ đầu. Thư mục quan trọng nhất là pages, nơi mỗi file JavaScript/TypeScript sẽ tự động trở thành một route. Ngoài ra, chúng tôi sử dụng các thư mục khác như components, public, styles, lib để quản lý mã nguồn một cách hiệu quả.
  + pages/: Chứa các trang của ứng dụng.
  + pages/api/: Nơi chứa các API routes.
  + components/: Chứa các component React có thể tái sử dụng.
  + public/: Chứa các tài nguyên tĩnh như hình ảnh, font chữ.
* **Hệ thống Routing:** Hệ thống routing của Next.js rất đơn giản.
  + Tạo file pages/index.js sẽ tạo ra route /.
  + Tạo file pages/about.js sẽ tạo ra route /about.
  + Tạo file pages/products/[id].js sẽ tạo ra một route động, ví dụ /products/123.
* **Fetching Dữ liệu:** Next.js cung cấp các hàm đặc biệt để lấy dữ liệu.
  + **getStaticProps:** Được sử dụng để lấy dữ liệu tĩnh trong quá trình build. Trong dự án URBAN FLOW, chúng tôi dùng hàm này cho các trang sản phẩm hoặc blog có nội dung ít thay đổi để đảm bảo tốc độ tải trang nhanh nhất.
  + **getServerSideProps:** Được dùng để lấy dữ liệu động trên mỗi yêu cầu. Đây là hàm lý tưởng cho các trang cần dữ liệu cập nhật theo thời gian thực, như trang theo dõi đơn hàng, để đảm bảo người dùng luôn có thông tin mới nhất.
  + **getStaticPaths:** Đi kèm với **getStaticProps**, hàm này được dùng để định nghĩa các route động nào cần được render tĩnh trước (ví dụ: tạo các trang tĩnh cho tất cả các sản phẩm có sẵn trong cơ sở dữ liệu).

1. **BACKEND – XÂY DỰNG MÁY CHỦ VỚI NODEJS VÀ EXPRESS.JS**
   1. **NODEJS**
      1. **Tổng Quan Về Môi Trường Runtime NodeJS**

**Node.js** không phải là một ngôn ngữ lập trình hay một framework, mà là một môi trường thực thi (runtime environment) JavaScript mã nguồn mở, đa nền tảng. Trước đây, JavaScript chỉ được sử dụng để chạy ở phía trình duyệt (frontend). Tuy nhiên, Node.js đã mở rộng khả năng của JavaScript, cho phép nó được sử dụng để xây dựng các ứng dụng phía máy chủ (backend) mạnh mẽ, có khả năng mở rộng. Node.js sử dụng engine V8 của Google Chrome để biên dịch mã JavaScript thành mã máy, giúp việc thực thi trở nên cực kỳ nhanh chóng. Điều này tạo ra một lợi thế lớn cho các nhà phát triển, vì họ có thể sử dụng cùng một ngôn ngữ (JavaScript) cho cả frontend và backend, giúp quá trình phát triển trở nên liền mạch và hiệu quả hơn.

* + 1. **Kiến Trúc Bất Đồng Bộ (Non-Blocking I/O) Và Event Loop**

Đây là trái tim của Node.js và là yếu tố then chốt giúp nó trở nên mạnh mẽ trong việc xử lý các ứng dụng có nhiều yêu cầu đồng thời.

* **Kiến trúc bất đồng bộ (Non-blocking I/O):**

Không giống như các mô hình lập trình đồng bộ truyền thống, Node.js không xử lý từng yêu cầu một cách tuần tự. Khi có một yêu cầu cần thực hiện một tác vụ tốn thời gian (như đọc file, truy vấn cơ sở dữ liệu, gọi API từ bên ngoài), Node.js sẽ không chờ đợi tác vụ đó hoàn thành. Thay vào đó, nó sẽ gửi yêu cầu và tiếp tục xử lý các tác vụ khác. Khi tác vụ tốn thời gian hoàn thành, nó sẽ kích hoạt một callback function để xử lý kết quả.

* **Event Loop (Vòng lặp sự kiện):**

Event Loop là cơ chế cốt lõi giúp kiến trúc bất đồng bộ của Node.js hoạt động. Nó liên tục kiểm tra hàng đợi sự kiện (event queue) để xem có tác vụ nào đã hoàn thành và sẵn sàng được xử lý chưa. Khi phát hiện một sự kiện đã hoàn thành, Event Loop sẽ lấy callback function tương ứng và đưa nó vào stack để thực thi.

Trong dự án URBAN FLOW, kiến trúc này đặc biệt quan trọng để xử lý hàng loạt các yêu cầu đồng thời từ người dùng, tài xế, và các dịch vụ khác mà không làm ứng dụng bị "đứng". Ví dụ, khi hàng trăm tài xế gửi yêu cầu cập nhật vị trí cùng lúc, Node.js có thể xử lý tất cả một cách hiệu quả mà không bị tắc nghẽn.

* + 1. **Ưu Và Nhược Điểm Của Nodejs**
* **Ưu điểm:**

**Hiệu suất cao:** Nhờ kiến trúc bất đồng bộ và Event Loop, Node.js có khả năng xử lý lượng lớn yêu cầu đồng thời một cách nhanh chóng và hiệu quả.

**Sử dụng JavaScript:** Cho phép phát triển toàn diện (full-stack) chỉ với một ngôn ngữ duy nhất, giúp tiết kiệm thời gian học hỏi và tối ưu hóa quy trình làm việc.

**Cộng đồng lớn và mạnh mẽ:** Node.js có một cộng đồng rộng lớn, với hàng triệu thư viện có sẵn trên NPM, giúp việc tích hợp các tính năng phức tạp trở nên dễ dàng hơn.

**Khả năng mở rộng:** Node.js rất phù hợp để xây dựng các ứng dụng có khả năng mở rộng, đáp ứng được nhu cầu của một nền tảng đa dịch vụ như URBAN FLOW.

* **Nhược điểm:**

Không phù hợp với các tác vụ nặng về CPU: Node.js chỉ sử dụng một luồng (single-threaded), nên không hiệu quả cho các tác vụ tính toán phức tạp. Tuy nhiên, vấn đề này có thể được giải quyết bằng cách sử dụng các Worker Threads.

Sự thay đổi nhanh: Hệ sinh thái của Node.js phát triển rất nhanh, đôi khi đòi hỏi các nhà phát triển phải liên tục cập nhật kiến thức.

* + 1. **Các Khái Niệm Mở Rộng: NPM (Node Package Manager), Modules**
* **NPM (Node Package Manager):**

NPM là trình quản lý gói mặc định của Node.js và là kho lưu trữ phần mềm lớn nhất thế giới. Nó cho phép các nhà phát triển dễ dàng cài đặt, quản lý và chia sẻ các thư viện (packages) cho dự án của mình.

Trong dự án URBAN FLOW, chúng tôi sử dụng NPM để cài đặt các thư viện cần thiết như Express.js, Mongoose (tương tác với MongoDB), Socket.IO (cho tính năng thời gian thực) và nhiều công cụ hỗ trợ khác.

* **Modules:**

Modules là một cách để chia nhỏ mã nguồn thành các file riêng biệt, có thể được tái sử dụng và quản lý dễ dàng. Mỗi file trong Node.js đều được coi là một module.

**Tác dụng:** Giúp tổ chức mã nguồn một cách có cấu trúc, tránh xung đột biến toàn cục và cho phép chia sẻ chức năng giữa các file.

**Cơ chế hoạt động:** Chúng tôi sử dụng các cú pháp require() và exports để nhập và xuất các module, giúp xây dựng một hệ thống backend có tổ chức, dễ bảo trì.

* 1. **EXPRESS.JS**
     1. **Tổng Quan Về Framwork Express.js**

**Express.js** là một framework phía máy chủ tối giản, linh hoạt và mạnh mẽ, được xây dựng trên nền tảng Node.js. Express.js không phải là một nền tảng độc lập, mà là một lớp trừu tượng hóa, cung cấp một bộ công cụ và tính năng phong phú giúp đơn giản hóa việc xây dựng các ứng dụng web và API. Nó giải quyết những khó khăn khi sử dụng Node.js thuần túy, chẳng hạn như việc phải tự viết logic phức tạp để xử lý routing, middleware, và các yêu cầu HTTP. Nhờ Express.js, chúng tôi có thể xây dựng backend cho dự án URBAN FLOW một cách nhanh chóng, có tổ chức và dễ bảo trì.

* + 1. **Middleware, Routing**
* **Middleware (Phần mềm trung gian):**

Middleware là trái tim của Express.js. Nó là các hàm có quyền truy cập vào đối tượng yêu cầu (request), đối tượng phản hồi (response) và hàm next(). Middleware có thể thực hiện nhiều tác vụ khác nhau trong chu kỳ xử lý một yêu cầu HTTP, từ khi yêu cầu được gửi đến máy chủ cho đến khi phản hồi được trả về.

**Các loại Middleware:**

* + Application-level middleware: Áp dụng cho toàn bộ ứng dụng. Ví dụ: app.use(express.json()); để tự động phân tích dữ liệu JSON trong body của yêu cầu. Trong dự án URBAN FLOW, chúng tôi sử dụng middleware này để xử lý dữ liệu từ form đặt hàng hoặc thông tin sản phẩm.
  + Router-level middleware: Áp dụng cho một nhóm các route cụ thể. Chúng tôi có thể sử dụng loại middleware này để xác thực người dùng trước khi cho phép họ truy cập vào các route quản lý đơn hàng.
  + Error-handling middleware: Dùng để xử lý các lỗi xảy ra trong quá trình xử lý yêu cầu.
* **Routing (Định tuyến):**

Routing là quá trình xác định cách một ứng dụng phản hồi lại một yêu cầu của client (chẳng hạn như một yêu cầu GET hoặc POST) đến một endpoint cụ thể. Express.js cung cấp một API rõ ràng để định nghĩa các route, giúp chúng tôi dễ dàng xây dựng các API RESTful. Mỗi route sẽ bao gồm một đường dẫn (path) và một hàm xử lý (handler function) nhận hai tham số là req (request) và res (response).

**Ví dụ:**

* + app.get('/api/products', ...): Trả về danh sách sản phẩm.
  + app.post('/api/orders', ...): Tạo một đơn hàng mới.
  + app.put('/api/orders/:id', ...): Cập nhật thông tin đơn hàng theo ID.

Trong dự án URBAN FLOW, chúng tôi sử dụng routing để xây dựng các API cho tất cả các chức năng, bao gồm quản lý sản phẩm, đơn hàng, tài xế và người dùng.

* + 1. **Ưu và nhược điểm của Express.js**
* **Ưu điểm:**

**Tối giản và linh hoạt:** Express.js cung cấp các tính năng cốt lõi, cho phép các nhà phát triển tự do lựa chọn các thư viện khác để tích hợp, phù hợp với mọi quy mô dự án.

**Hiệu suất cao:** Nhờ được xây dựng trên nền tảng Node.js, Express.js có hiệu suất cao, lý tưởng cho việc xây dựng các API có tính năng thời gian thực.

**Cộng đồng lớn:** Express.js có một cộng đồng hỗ trợ đông đảo và rất nhiều tài liệu, các gói (package) mở rộng sẵn có.

* **Nhược điểm:**

**Tối giản:** Vì là một framework tối giản, Express.js không cung cấp nhiều tính năng "có sẵn" như các framework đầy đủ chức năng khác (ví dụ: Adones.js, NestJS). Điều này đôi khi đòi hỏi phải cài đặt và cấu hình thêm nhiều thư viện bên ngoài.

**Cấu trúc linh hoạt:** Mặc dù sự linh hoạt là một ưu điểm, nó cũng có thể trở thành nhược điểm nếu không có một quy tắc hoặc cấu trúc rõ ràng, đặc biệt trong các dự án lớn.

* + 1. **Quy Trình Triển Khai: Cài Đặt Và Cấu Hình Cơ Bản, Xây Dựng Các Api Restful**
* **Cài đặt và Cấu hình Cơ bản:**
  + **Khởi tạo dự án Node.js:** Sử dụng lệnh npm init -y để tạo file package.json.
  + **Cài đặt Express.js:** Dùng lệnh [**npm install express**].
  + **Tạo file server chính:** Tạo một file server.js (hoặc index.js) để viết code backend.
  + **Cấu hình server:** Cấu hình các middleware cơ bản (như express.json(), cors) và định nghĩa các port để ứng dụng lắng nghe.
* **Xây dựng các API RESTful:**
  + Trong dự án URBAN FLOW, chúng tôi xây dựng các API RESTful để xử lý tất cả các yêu cầu từ frontend. Mỗi API sẽ thực hiện một chức năng cụ thể và tuân thủ các nguyên tắc của REST (Representational State Transfer).
  + **Tài nguyên (Resources):** Các đối tượng trong hệ thống, ví dụ: /api/products (tài nguyên sản phẩm), /api/orders (tài nguyên đơn hàng).
  + **Các phương thức HTTP (HTTP Verbs**): Sử dụng các phương thức chuẩn để thực hiện các thao tác:
    - **GET:** Lấy dữ liệu (ví dụ: lấy danh sách sản phẩm).
    - **POST:** Tạo dữ liệu mới (ví dụ: tạo một đơn hàng mới).
    - **PUT/PATCH:** Cập nhật dữ liệu (ví dụ: cập nhật trạng thái đơn hàng).
    - **DELETE:** Xóa dữ liệu (ví dụ: xóa một sản phẩm).
    - **Phản hồi (Response):** Các API sẽ trả về dữ liệu dưới dạng JSON cùng với các mã trạng thái HTTP thích hợp (ví dụ: 200 OK, 201 Created, 404 Not Found, 500 Server Error).

Việc sử dụng Express.js giúp chúng tôi dễ dàng tổ chức mã nguồn, xây dựng các API mạnh mẽ và hiệu quả, đảm bảo tính ổn định cho toàn bộ hệ thống backend của URBAN FLOW.

1. **DATABASE – Quản Lý Cơ Sở Dữ Liệu (MongoDB)**
   1. **Tổng Quan Về MongoDB**
      1. **Giới thiệu Về Cơ Sở Dữ Liệu NoSQL**

**MongoDB** là một hệ quản trị cơ sở dữ liệu NoSQL (Not only SQL) mã nguồn mở, theo định hướng tài liệu (document-oriented). Khác với các hệ cơ sở dữ liệu quan hệ truyền thống (SQL) như MySQL hay PostgreSQL, MongoDB không lưu trữ dữ liệu trong các bảng với các hàng và cột có cấu trúc cố định. Thay vào đó, nó lưu trữ dữ liệu dưới dạng các tài liệu (document) theo định dạng BSON (Binary JSON), một định dạng nhị phân của JSON. Mỗi tài liệu có thể có cấu trúc khác nhau, mang lại sự linh hoạt đáng kể trong việc thiết kế và quản lý dữ liệu.

MongoDB được thiết kế để dễ dàng mở rộng theo chiều ngang (horizontal scaling), cho phép xử lý lượng dữ liệu lớn và các yêu cầu truy vấn phức tạp một cách hiệu quả. Đây là lý do chính khiến MongoDB trở thành lựa chọn hàng đầu cho các ứng dụng web hiện đại, đặc biệt là những ứng dụng có tính năng động và cấu trúc dữ liệu không đồng nhất như dự án URBAN FLOW. ..

* + 1. **Ưu Và Nhược Điểm Của MongoDB So Với SQL**

Việc lựa chọn MongoDB cho dự án URBAN FLOW được cân nhắc dựa trên các ưu và nhược điểm sau:

* **Ưu điểm:**

**Linh hoạt về cấu trúc dữ liệu:** MongoDB cho phép các tài liệu trong cùng một bộ sưu tập (collection) có cấu trúc khác nhau. Điều này rất phù hợp với dự án của chúng tôi, nơi các loại dữ liệu như đơn hàng, sản phẩm và thông tin người dùng có thể thay đổi và bổ sung thuộc tính mới một cách dễ dàng mà không ảnh hưởng đến toàn bộ hệ thống.

**Khả năng mở rộng:** MongoDB hỗ trợ tính năng phân vùng dữ liệu (sharding), cho phép phân tán dữ liệu trên nhiều máy chủ. Điều này giúp hệ thống của chúng tôi dễ dàng mở rộng khi lượng dữ liệu và số lượng người dùng tăng lên.

**Tốc độ phát triển nhanh:** Với mô hình dữ liệu linh hoạt, việc thay đổi cấu trúc dữ liệu (schema) trở nên dễ dàng hơn so với SQL. Điều này giúp đẩy nhanh tốc độ phát triển và thử nghiệm các tính năng mới cho URBAN FLOW.

**Hỗ trợ truy vấn không gian địa lý:** MongoDB có các tính năng tích hợp sẵn để xử lý dữ liệu địa lý (geospatial data), rất quan trọng cho các chức năng bản đồ và định vị trong dự án. Chúng tôi có thể dễ dàng tìm kiếm các tài xế gần nhất hoặc xác định phạm vi giao hàng.

* **Nhược điểm:**

Thiếu tính toàn vẹn dữ liệu (Data integrity): Vì không có các ràng buộc chặt chẽ như khóa ngoại (foreign key), việc đảm bảo tính toàn vẹn dữ liệu có thể khó khăn hơn so với SQL. Chúng tôi cần phải tự quản lý các mối quan hệ này ở phía ứng dụng.

Không phù hợp với các giao dịch phức tạp: Mặc dù MongoDB đã có những cải tiến về giao dịch đa tài liệu, nhưng nó vẫn không phải là lựa chọn tốt nhất cho các ứng dụng đòi hỏi các giao dịch phức tạp, nhiều bước (multi-step transactions) và tính nhất quán tuyệt đối..

* + 1. **Mô Hình Dữ Liệu: Document, Collection**

Mô hình dữ liệu của MongoDB đơn giản và trực quan, được xây dựng dựa trên hai khái niệm chính: Document và Collection.

* **Document (Tài liệu):**

Một document trong MongoDB tương đương với một hàng (row) trong cơ sở dữ liệu SQL. Tuy nhiên, nó có thể chứa các trường (field) lồng nhau và mảng (array), mang lại cấu trúc phức tạp và linh hoạt hơn nhiều.

Mỗi document được đại diện bởi một cặp key-value, được lưu trữ dưới dạng BSON.

Trong dự án URBAN FLOW, mỗi tài liệu có thể là một đơn hàng, một sản phẩm hoặc một tài khoản người dùng. Ví dụ, một tài liệu đơn hàng có thể chứa các thông tin như orderId, userId, products (là một mảng các sản phẩm), totalPrice, status, pickupLocation (chứa dữ liệu địa lý), v.v.

* **Collection (Bộ sưu tập):**

Một collection trong MongoDB tương đương với một bảng (table) trong cơ sở dữ liệu SQL.

Một collection chứa một nhóm các document. Tuy nhiên, không giống như một bảng SQL, các document trong cùng một collection có thể có cấu trúc khác nhau.

Trong dự án URBAN FLOW, chúng tôi sử dụng các collection sau:

* + users: Chứa các document về tài khoản người dùng và tài xế.
  + products: Chứa các document về thông tin sản phẩm và nhà cung cấp.
  + orders: Chứa các document về các đơn hàng, bao gồm thông tin chi tiết về sản phẩm, trạng thái và vị trí.
  + …
  1. **QUY TRÌNH TÍCH HỢP VỚI BACKEND**
     1. **Cài Đặt Và Cấu Hình Mongodb**

Việc tích hợp MongoDB vào dự án được thực hiện theo các bước sau:

* **Cài đặt MongoDB:** Chúng tôi sử dụng các dịch vụ MongoDB Cloud (MongoDB Atlas) để đảm bảo tính ổn định và khả năng mở rộng. MongoDB Atlas cung cấp một cơ sở dữ liệu được quản lý hoàn toàn, giúp chúng tôi không phải lo lắng về việc cài đặt và cấu hình máy chủ.
* **Tạo một cơ sở dữ liệu mới:** Trong MongoDB Atlas, chúng tôi tạo một cơ sở dữ liệu mới cho dự án URBAN FLOW và cấu hình các thông số bảo mật cần thiết, bao gồm tạo người dùng cơ sở dữ liệu và thiết lập danh sách địa chỉ IP được phép truy cập.
* **Lấy chuỗi kết nối (Connection String):** MongoDB Atlas cung cấp một chuỗi kết nối duy nhất. Chuỗi này chứa thông tin cần thiết để ứng dụng Node.js kết nối đến cơ sở dữ liệu.
* **Cấu hình trong dự án Node.js:** Chúng tôi lưu trữ chuỗi kết nối này trong một biến môi trường (.env) để đảm bảo tính bảo mật. Trong file server chính của dự án, chúng tôi sử dụng thư viện Mongoose để thiết lập kết nối đến MongoDB..
  + 1. **Sử dụng Mongoose (Thư Viện ODM) Để Tương Tác với MongoDB**

Mặc dù Node.js có thể kết nối trực tiếp với MongoDB, việc sử dụng thư viện Mongoose mang lại nhiều lợi ích đáng kể. Mongoose là một Object Data Modeling (ODM) library, cung cấp một lớp trừu tượng hóa giúp việc tương tác với MongoDB trở nên dễ dàng và có cấu trúc hơn.

* **Mô hình hóa dữ liệu (Schema):** Mongoose cho phép chúng tôi định nghĩa các lược đồ (schema) cho dữ liệu. Một schema mô tả cấu trúc, kiểu dữ liệu, các ràng buộc và các giá trị mặc định cho một document. Điều này giúp đảm bảo tính nhất quán của dữ liệu, ngay cả khi MongoDB là một cơ sở dữ liệu không ràng buộc về schema.
  + **Ví dụ:** Chúng tôi tạo một UserSchema để định nghĩa các trường như name, email, password, role (người dùng hoặc tài xế).
* **Mô hình (Model):** Sau khi định nghĩa schema, chúng tôi tạo một model từ schema đó. Model là một lớp cung cấp các phương thức để thực hiện các thao tác CRUD (Create, Read, Update, Delete) với collection tương ứng.
* **Các tính năng khác:** Mongoose còn cung cấp các tính năng hữu ích khác như middleware, validation (kiểm tra dữ liệu hợp lệ), và population (cho phép "tham chiếu" giữa các document).
  + 1. **Các Thao Tác Cơ Bản (CRUD: Create, Read, Update, Delete)**

Dựa trên các model được định nghĩa bằng Mongoose, chúng tôi xây dựng các API RESTful trong Express.js để thực hiện các thao tác cơ bản với cơ sở dữ liệu. Dưới đây là các thao tác chính trong dự án URBAN FLOW:

* **Create (Tạo):**

Khi người dùng đăng ký tài khoản hoặc đặt một đơn hàng mới, frontend sẽ gửi yêu cầu POST đến backend.

Backend nhận dữ liệu từ yêu cầu và sử dụng phương thức model.create() của Mongoose để tạo một document mới trong collection tương ứng (users, orders).

* **Read (Đọc):**

Khi người dùng muốn xem danh sách sản phẩm, lịch sử đơn hàng, hoặc thông tin chi tiết một tài khoản, frontend sẽ gửi yêu cầu GET đến backend.

Backend sử dụng các phương thức của Mongoose như model.find() (để lấy nhiều document), model.findById() (để lấy một document theo ID), hoặc model.findOne() (để lấy một document theo điều kiện) để truy xuất dữ liệu từ MongoDB.

* **Update (Cập nhật):**

Khi người dùng cập nhật thông tin cá nhân hoặc khi trạng thái của một đơn hàng thay đổi, frontend sẽ gửi yêu cầu PUT hoặc PATCH đến backend.

Backend sẽ sử dụng phương thức model.findByIdAndUpdate() hoặc model.updateOne() của Mongoose để tìm document và cập nhật dữ liệu.

* **Delete (Xóa):**

Khi cần xóa một sản phẩm khỏi danh mục hoặc hủy một đơn hàng, frontend sẽ gửi yêu cầu DELETE đến backend.

Backend sử dụng phương thức model.findByIdAndDelete() hoặc model.deleteOne() của Mongoose để xóa document khỏi collection.

Quy trình này đảm bảo rằng việc quản lý dữ liệu trong dự án URBAN FLOW được thực hiện một cách có hệ thống, an toàn và hiệu quả, tận dụng tối đa sức mạnh của cả Node.js/Express.js và MongoDB.

1. **QUY TRÌNH TRIỂN KHAI DỰ ÁN (DEPLOYMENT)**

Quy trình triển khai dự án là một giai đoạn quan trọng, nơi các thành phần frontend và backend được kết hợp và đưa lên môi trường sản phẩm để người dùng có thể truy cập. Đối với một ứng dụng phức tạp như **URBAN FLOW**, việc triển khai đòi hỏi sự chuẩn bị kỹ lưỡng về cơ sở hạ tầng, tối ưu hóa hiệu suất và đảm bảo an toàn.

* 1. **XÂY DỰNG ỨNG DỤNG HOÀN CHỈNH**

Trước khi triển khai, chúng tôi phải đảm bảo tất cả các thành phần của ứng dụng được xây dựng, tích hợp và tối ưu hóa một cách hoàn chỉnh.

* + 1. **Tích Hợp Frontend (Next.js) và Backend (Node.js + Express.js)**

Mặc dù frontend và backend của URBAN FLOW được phát triển độc lập, chúng phải được tích hợp chặt chẽ để tạo thành một hệ thống hoạt động liền mạch.

* **Giao tiếp thông qua API RESTful:**

Frontend (Next.js/ReactJS) gửi các yêu cầu HTTP (GET, POST, PUT, DELETE) tới các API endpoint được cung cấp bởi Backend (Node.js/Express.js).

**Ví dụ:** Khi người dùng đặt một đơn hàng mới trên giao diện React, yêu cầu POST sẽ được gửi đến /api/orders trên backend. Backend sẽ xử lý yêu cầu, lưu dữ liệu vào MongoDB và trả về phản hồi cho frontend.

* **Kết nối thời gian thực với WebSockets (Socket.IO):**

Đối với các tính năng cần cập nhật dữ liệu ngay lập tức như theo dõi vị trí tài xế trên bản đồ hoặc thông báo trạng thái đơn hàng, chúng tôi sử dụng Socket.IO.

Frontend sẽ thiết lập kết nối WebSocket với backend. Khi có sự kiện thay đổi dữ liệu (ví dụ: tài xế cập nhật vị trí), backend sẽ gửi thông báo trực tiếp đến các client (người dùng, tài xế) thông qua WebSocket mà không cần frontend phải liên tục gửi yêu cầu. Điều này giúp giảm tải cho server và cải thiện trải nghiệm người dùng.

* **Cấu hình biến môi trường:**

Các thông tin nhạy cảm như URL của API backend, khóa API của các dịch vụ bên ngoài (bản đồ, thanh toán) được lưu trữ dưới dạng biến môi trường.

Điều này đảm bảo rằng các thông tin này không bị lộ trong mã nguồn công khai và dễ dàng thay đổi khi triển khai sang các môi trường khác nhau (phát triển, thử nghiệm, sản phẩm).

* + 1. **Tối Ưu Hóa Hiện Suất Và Bảo Mật**
* **Tối ưu hóa hiệu suất:**

**Giảm thiểu kích thước Bundle:** Sử dụng các công cụ như Webpack (được tích hợp sẵn trong Next.js) để tối ưu hóa kích thước của các file JavaScript, CSS và hình ảnh.

**Lazy Loading:** Sử dụng tính năng code splitting của Next.js để chỉ tải các phần của ứng dụng khi chúng thực sự cần thiết, giúp giảm thời gian tải ban đầu.

**Caching:** Cấu hình caching ở cả phía client (browser cache) và phía server (CDN) để lưu trữ các tài nguyên tĩnh và dữ liệu thường xuyên được truy cập, giảm tải cho máy chủ.

**Nén dữ liệu (Compression):** Kích hoạt Gzip hoặc Brotli trên máy chủ để nén các phản hồi HTTP, giảm băng thông truyền tải.

* **Tối ưu hóa bảo mật:**

**Xác thực và ủy quyền (Authentication & Authorization):** Triển khai các cơ chế xác thực mạnh mẽ (ví dụ: JWT - JSON Web Tokens) để bảo vệ các API. Đảm bảo người dùng chỉ có thể truy cập vào các tài nguyên mà họ được phép.

**Bảo mật dữ liệu:** Sử dụng các phương pháp mã hóa (hashing) cho mật khẩu người dùng trước khi lưu vào MongoDB.

**Ngăn chặn tấn công:**

* + **Cross-Site Scripting (XSS):** Áp dụng các biện pháp bảo vệ để ngăn chặn việc chèn mã độc vào trang web.
  + **Cross-Site Request Forgery (CSRF):** Sử dụng các token CSRF để bảo vệ các form gửi dữ liệu.
  + **SQL Injection (NoSQL Injection):** Mặc dù MongoDB là NoSQL, vẫn cần cẩn trọng với các truy vấn động để tránh các lỗ hổng tương tự. Sử dụng Mongoose giúp giảm thiểu rủi ro này.
* **HTTPS:** Bắt buộc sử dụng giao thức HTTPS để mã hóa tất cả lưu lượng truy cập giữa client và server, đảm bảo dữ liệu không bị nghe lén.
  1. **TRIỂN KHAI LÊN MÁY CHỦ**

Quá trình **triển khai (deployment)** là giai đoạn đưa ứng dụng **URBAN FLOW** từ môi trường phát triển lên môi trường sản phẩm để người dùng có thể truy cập. Việc lựa chọn và cấu hình môi trường máy chủ phù hợp là yếu tố quyết định đến hiệu suất, độ ổn định và khả năng mở rộng của ứng dụng.

* + 1. **Chuẩn Bị Môi Trường Máy Chủ (Ví dụ: Vercel, Heroku, AWS, Google Cloud).**

Việc lựa chọn nền tảng đám mây phù hợp là rất quan trọng để đảm bảo ứng dụng có hiệu suất và khả năng mở rộng tốt. Mỗi nền tảng đều có những ưu điểm riêng, phù hợp với các mục đích sử dụng khác nhau.

* **Vercel (cho Next.js Frontend):**

Vercel là nền tảng tối ưu cho các ứng dụng Next.js. Nó được thiết kế đặc biệt để tận dụng tối đa các tính năng của Next.js như Server-Side Rendering (SSR) và Static Site Generation (SSG).

**Ưu điểm:**

* + Triển khai tự động: Tự động build và deploy ứng dụng mỗi khi có thay đổi trên repository Git (GitHub, GitLab, Bitbucket).
  + Mạng lưới phân phối nội dung (CDN) toàn cầu: Giúp các tài nguyên tĩnh được phân phối gần người dùng cuối, giảm độ trễ và tăng tốc độ tải trang.
  + Hỗ trợ SSR và SSG: Xử lý hiệu quả các phương thức render của Next.js, đảm bảo hiệu suất cao và SEO tốt.

Trong dự án URBAN FLOW, chúng tôi sử dụng Vercel để host toàn bộ phần frontend. Điều này giúp trang web có tốc độ tải nhanh, mượt mà và dễ dàng quản lý việc cập nhật.

* **Heroku (cho Node.js Backend):**

Heroku là một nền tảng PaaS (Platform as a Service) phổ biến, dễ sử dụng cho các ứng dụng Node.js. Heroku cho phép các nhà phát triển triển khai ứng dụng mà không cần quản lý cơ sở hạ tầng.

**Ưu điểm:**

* + Dễ dàng triển khai: Chỉ cần đẩy mã nguồn lên Heroku Git remote, Heroku sẽ tự động phát hiện, build và chạy ứng dụng.
  + Quản lý biến môi trường: Cung cấp giao diện dễ dàng để quản lý các biến môi trường quan trọng, đảm bảo tính bảo mật.
  + Khả năng mở rộng: Có thể dễ dàng tăng số lượng "dynos" (các instance của ứng dụng) để mở rộng quy mô khi cần.

Heroku là một lựa chọn tốt để bắt đầu cho backend của URBAN FLOW, giúp việc triển khai nhanh chóng và đơn giản.

* **AWS (Amazon Web Services) / Google Cloud:**

Đây là những nền tảng đám mây mạnh mẽ và linh hoạt hơn, phù hợp cho các dự án lớn, đòi hỏi kiểm soát cao hơn.

**AWS EC2/Google Cloud Compute Engine:** Chúng tôi có thể sử dụng các dịch vụ máy ảo này để tự host backend Node.js. Điều này mang lại sự kiểm soát tuyệt đối về cấu hình máy chủ nhưng cũng đòi hỏi nhiều công sức quản lý hơn.

**AWS DocumentDB/Google Cloud Firestore:** Các dịch vụ database tương thích với MongoDB được quản lý hoàn toàn, có khả năng mở rộng và bảo mật cao.

**Ưu điểm:** Khả năng mở rộng vô hạn, đa dạng dịch vụ, tùy chỉnh tối đa.

**Nhược điểm:** Phức tạp hơn trong việc cấu hình và quản lý.

Sự kết hợp giữa Vercel cho frontend và một dịch vụ như Heroku hoặc AWS cho backend, cùng với MongoDB Atlas cho database, mang lại sự cân bằng giữa hiệu suất, dễ quản lý và khả năng mở rộng cho dự án URBAN FLOW.

* + 1. **Các Bước Triển Khai: Cấu Hình Môi Trường, Chạy Ứng Dụng, Quản Lý Domain**
* **Cấu hình biến môi trường:**

Trước khi triển khai, chúng tôi thiết lập các biến môi trường quan trọng trên nền tảng máy chủ. Các biến này bao gồm:

* + MONGODB\_URI: Chuỗi kết nối đến cơ sở dữ liệu MongoDB Atlas.
  + API\_URL: URL của backend API.
  + JWT\_SECRET: Khóa bí mật để tạo và xác thực token.
  + NODE\_ENV=production: Thiết lập môi trường sản phẩm.

Việc sử dụng biến môi trường đảm bảo rằng các thông tin nhạy cảm không bị lộ trong mã nguồn và dễ dàng thay đổi khi cần.

* **Triển khai Frontend (Vercel):**

Kết nối kho mã nguồn (Git repository) của dự án Next.js với Vercel.

Vercel sẽ tự động phát hiện dự án Next.js và thực hiện các lệnh build (next build) để tạo các tệp tĩnh và start (next start) để khởi chạy server.

Vercel cũng tự động tối ưu hóa code splitting, image optimization và hosting trên CDN.

* **Triển khai Backend (Heroku/AWS):**

Với Heroku: Chúng tôi chỉ cần cấu hình Heroku Git remote và đẩy mã nguồn lên. Heroku sẽ tự động cài đặt các dependencies (dựa trên package.json) và khởi động ứng dụng Node.js.

Với AWS/Google Cloud: Chúng tôi tạo một máy ảo, cài đặt Node.js và các dependencies. Sau đó, sử dụng các công cụ như PM2 để quản lý tiến trình của ứng dụng, đảm bảo ứng dụng luôn chạy ổn định và tự khởi động lại khi có lỗi.

* **Quản lý Domain:**

Sau khi ứng dụng được triển khai, chúng tôi sẽ có các URL mặc định từ các nền tảng (ví dụ: urban-flow.vercel.app, urban-flow-backend.herokuapp.com).

Chúng tôi mua một tên miền (ví dụ: urbanflow.com) và cấu hình bản ghi DNS để trỏ tên miền về các dịch vụ đã triển khai.

Cuối cùng, chúng tôi kích hoạt HTTPS bằng cách sử dụng chứng chỉ SSL/TLS. Cả Vercel và Heroku đều cung cấp tính năng này một cách tự động, đảm bảo tất cả lưu lượng truy cập được mã hóa và bảo mật.

* 1. **CÁC VẤN ĐỀ SAU TRIỂN KHAI**

Sau khi ứng dụng đã đi vào hoạt động, việc giám sát và bảo trì là rất quan trọng để đảm bảo hệ thống luôn ổn định và hiệu quả.

* + 1. **Giám sát hiệu suất ứng dụng**
* **Logs (Nhật ký**): Chúng tôi sử dụng các dịch vụ logs tích hợp (như của Vercel và Heroku) để thu thập và phân tích các nhật ký từ cả frontend và backend. Điều này giúp chúng tôi theo dõi các lỗi, cảnh báo và hành vi bất thường của hệ thống.
* **Metrics (Chỉ số):** Giám sát các chỉ số hiệu suất quan trọng như:
  + **Thời gian phản hồi API:** Theo dõi độ trễ của các API để xác định các điểm nghẽn.
  + **Tải CPU và RAM:** Đảm bảo máy chủ có đủ tài nguyên để xử lý yêu cầu.
  + Số lượng yêu cầu đồng thời: Đánh giá khả năng chịu tải của hệ thống.
  + **Core Web Vitals:** Giám sát hiệu suất frontend như Largest Contentful Paint (LCP) để đảm bảo trải nghiệm người dùng tốt.
* **Công cụ giám sát:** Chúng tôi có thể sử dụng các công cụ chuyên dụng như New Relic hoặc Datadog để có cái nhìn sâu sắc hơn về hiệu suất ứng dụng và nhận các cảnh báo kịp thời.
  + 1. **Sao lưu và phục hồi dữ liệu.**
* **Sao lưu định kỳ:**

Thiết lập các kế hoạch sao lưu tự động cho cơ sở dữ liệu MongoDB. MongoDB Atlas cung cấp các công cụ tích hợp sẵn để sao lưu dữ liệu theo lịch trình và lưu trữ ở nhiều địa điểm an toàn.

Chúng tôi đảm bảo dữ liệu được sao lưu đều đặn để phòng tránh mất mát do lỗi phần mềm, lỗi phần cứng hoặc tấn công mạng.

* **Kế hoạch phục hồi (Disaster Recovery Plan):**

Xây dựng và kiểm tra quy trình phục hồi dữ liệu từ các bản sao lưu.

Đảm bảo có thể khôi phục hệ thống về trạng thái hoạt động bình thường trong thời gian ngắn nhất nếu xảy ra sự cố nghiêm trọng, chẳng hạn như lỗi cơ sở dữ liệu.

Việc tuân thủ một quy trình triển khai và bảo trì chuyên nghiệp sẽ giúp dự án URBAN FLOW hoạt động ổn định, hiệu quả và đáng tin cậy cho người dùng.

Chúng ta đã có một nền tảng vững chắc với các model dữ liệu, cấu trúc thư mục, và middleware xác thực cơ bản. Bây giờ, hãy đi sâu vào việc xây dựng các **API RESTful** cho từng module chức năng của UrbanFlows.

Mỗi API sẽ tuân theo các nguyên tắc REST:

* **Resource-based URLs:** Sử dụng các URL rõ ràng, tập trung vào tài nguyên (ví dụ: /api/items, /api/orders).
* **HTTP Methods:** Sử dụng đúng các phương thức HTTP (GET, POST, PUT, DELETE) cho các thao tác CRUD tương ứng.
* **Stateless:** Mỗi yêu cầu từ client tới server phải chứa tất cả thông tin cần thiết để server hiểu yêu cầu đó.
* **JSON Format:** Dữ liệu sẽ được gửi và nhận dưới dạng JSON.

Chúng ta sẽ xây dựng các API này theo cặp **Controller** và **Route**.

### 1. Cấu hình chung cho API

Đảm bảo rằng index.js của bạn đã được cấu hình để sử dụng express.json() và kết nối với database như đã hướng dẫn trước đó.

JavaScript

// urbanflows-backend/index.js

// ... (các import và cấu hình dotenv, connectDB, app, PORT)

// Middleware để xử lý JSON trong body của request

app.use(express.json());

// Các routes API sẽ được thêm vào đây

// Ví dụ:

// app.use('/api/accounts', require('./src/routes/accountRoutes'));

// app.use('/api/items', require('./src/routes/itemRoutes'));

// ...

// ... (route test và app.listen)

### 2. Xây dựng API RESTful cho từng module

Chúng ta sẽ xây dựng các API theo từng cặp Controller và Route.

#### 2.1. API Account (Đã làm)

Bạn đã có các API cơ bản cho Account (đăng ký, đăng nhập, profile). Chúng ta sẽ bổ sung thêm các API quản lý tài khoản (chỉ dành cho admin/manager).

**src/controllers/accountController.js (Bổ sung)**

JavaScript

// src/controllers/accountController.js (tiếp tục)

// ... (registerAccount, loginAccount, getAccountProfile)

// @desc Lấy tất cả tài khoản (chỉ admin)

// @route GET /api/accounts

// @access Private/Admin

const getAllAccounts = async (req, res) => {

try {

const accounts = await Account.find({}).select('-password'); // Không trả về mật khẩu

res.json(accounts);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Cập nhật thông tin tài khoản (chỉ admin/chủ tài khoản)

// @route PUT /api/accounts/:id

// @access Private/Admin or Account owner

const updateAccount = async (req, res) => {

const { id } = req.params;

const { fullName, email, username, roles, isActive } = req.body;

try {

const account = await Account.findById(id);

if (!account) {

return res.status(404).json({ message: 'Account not found.' });

}

// Chỉ admin hoặc chính chủ tài khoản mới được cập nhật

if (req.user.roles.includes('admin') || req.user.\_id.toString() === account.\_id.toString()) {

account.fullName = fullName || account.fullName;

account.email = email || account.email;

account.username = username || account.username;

// Chỉ admin mới được thay đổi vai trò hoặc trạng thái kích hoạt

if (req.user.roles.includes('admin')) {

account.roles = roles || account.roles;

account.isActive = typeof isActive === 'boolean' ? isActive : account.isActive;

}

const updatedAccount = await account.save();

res.json({

\_id: updatedAccount.\_id,

fullName: updatedAccount.fullName,

email: updatedAccount.email,

username: updatedAccount.username,

roles: updatedAccount.roles,

isActive: updatedAccount.isActive,

});

} else {

res.status(403).json({ message: 'Not authorized to update this account.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Xóa tài khoản (chỉ admin)

// @route DELETE /api/accounts/:id

// @access Private/Admin

const deleteAccount = async (req, res) => {

const { id } = req.params;

try {

const account = await Account.findById(id);

if (!account) {

return res.status(404).json({ message: 'Account not found.' });

}

await account.deleteOne(); // Sử dụng deleteOne() thay vì remove() cho các phiên bản Mongoose mới hơn

res.json({ message: 'Account removed successfully.' });

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

module.exports = { registerAccount, loginAccount, getAccountProfile, getAllAccounts, updateAccount, deleteAccount };

**src/routes/accountRoutes.js (Bổ sung)**

JavaScript

// src/routes/accountRoutes.js (tiếp tục)

const { protect, isAdmin } = require('../middleware/authMiddleware');

const { getAllAccounts, updateAccount, deleteAccount } = require('../controllers/accountController');

// Route công khai

router.post('/register', registerAccount);

router.post('/login', loginAccount);

// Route cần bảo vệ

router.get('/profile', protect, getAccountProfile);

// Route chỉ dành cho admin

router.get('/', protect, isAdmin, getAllAccounts); // Lấy tất cả tài khoản

router.put('/:id', protect, isAdmin, updateAccount); // Cập nhật tài khoản (admin có thể thay đổi vai trò)

router.delete('/:id', protect, isAdmin, deleteAccount); // Xóa tài khoản

module.exports = router;

#### 2.2. API Category

* **Model:** src/models/Category.js (đã có)

**src/controllers/categoryController.js**

JavaScript

// src/controllers/categoryController.js

const Category = require('../models/Category');

// @desc Lấy tất cả danh mục

// @route GET /api/categories

// @access Public

const getCategories = async (req, res) => {

try {

const categories = await Category.find({});

res.json(categories);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Lấy chi tiết một danh mục

// @route GET /api/categories/:id

// @access Public

const getCategoryById = async (req, res) => {

try {

const category = await Category.findById(req.params.id);

if (category) {

res.json(category);

} else {

res.status(404).json({ message: 'Category not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Tạo danh mục mới (chỉ admin)

// @route POST /api/categories

// @access Private/Admin

const createCategory = async (req, res) => {

const { name, description, imageUrl } = req.body;

try {

const categoryExists = await Category.findOne({ name });

if (categoryExists) {

return res.status(400).json({ message: 'Category with this name already exists.' });

}

const category = await Category.create({ name, description, imageUrl });

res.status(201).json(category);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Cập nhật danh mục (chỉ admin)

// @route PUT /api/categories/:id

// @access Private/Admin

const updateCategory = async (req, res) => {

const { name, description, imageUrl } = req.body;

try {

const category = await Category.findById(req.params.id);

if (category) {

category.name = name || category.name;

category.description = description || category.description;

category.imageUrl = imageUrl || category.imageUrl;

const updatedCategory = await category.save();

res.json(updatedCategory);

} else {

res.status(404).json({ message: 'Category not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Xóa danh mục (chỉ admin)

// @route DELETE /api/categories/:id

// @access Private/Admin

const deleteCategory = async (req, res) => {

try {

const category = await Category.findById(req.params.id);

if (category) {

await category.deleteOne();

res.json({ message: 'Category removed.' });

} else {

res.status(404).json({ message: 'Category not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

module.exports = {

getCategories,

getCategoryById,

createCategory,

updateCategory,

deleteCategory,

};

**src/routes/categoryRoutes.js**

JavaScript

// src/routes/categoryRoutes.js

const express = require('express');

const router = express.Router();

const { getCategories, getCategoryById, createCategory, updateCategory, deleteCategory } = require('../controllers/categoryController');

const { protect, isAdmin } = require('../middleware/authMiddleware');

// Public routes

router.get('/', getCategories);

router.get('/:id', getCategoryById);

// Admin routes

router.post('/', protect, isAdmin, createCategory);

router.put('/:id', protect, isAdmin, updateCategory);

router.delete('/:id', protect, isAdmin, deleteCategory);

module.exports = router;

**Thêm vào index.js:** app.use('/api/categories', require('./src/routes/categoryRoutes'));

#### 2.3. API Manufacturer

* **Model:** src/models/Manufacturer.js (đã có)

**src/controllers/manufacturerController.js**

JavaScript

// src/controllers/manufacturerController.js

const Manufacturer = require('../models/Manufacturer');

// @desc Lấy tất cả nhà sản xuất

// @route GET /api/manufacturers

// @access Public

const getManufacturers = async (req, res) => {

try {

const manufacturers = await Manufacturer.find({});

res.json(manufacturers);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Lấy chi tiết một nhà sản xuất

// @route GET /api/manufacturers/:id

// @access Public

const getManufacturerById = async (req, res) => {

try {

const manufacturer = await Manufacturer.findById(req.params.id);

if (manufacturer) {

res.json(manufacturer);

} else {

res.status(404).json({ message: 'Manufacturer not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Tạo nhà sản xuất mới (chỉ admin)

// @route POST /api/manufacturers

// @access Private/Admin

const createManufacturer = async (req, res) => {

const { name, email, address, contactPhone, website } = req.body;

try {

const manufacturerExists = await Manufacturer.findOne({ name });

if (manufacturerExists) {

return res.status(400).json({ message: 'Manufacturer with this name already exists.' });

}

const manufacturer = await Manufacturer.create({ name, email, address, contactPhone, website });

res.status(201).json(manufacturer);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Cập nhật nhà sản xuất (chỉ admin)

// @route PUT /api/manufacturers/:id

// @access Private/Admin

const updateManufacturer = async (req, res) => {

const { name, email, address, contactPhone, website } = req.body;

try {

const manufacturer = await Manufacturer.findById(req.params.id);

if (manufacturer) {

manufacturer.name = name || manufacturer.name;

manufacturer.email = email || manufacturer.email;

manufacturer.address = address || manufacturer.address;

manufacturer.contactPhone = contactPhone || manufacturer.contactPhone;

manufacturer.website = website || manufacturer.website;

const updatedManufacturer = await manufacturer.save();

res.json(updatedManufacturer);

} else {

res.status(404).json({ message: 'Manufacturer not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Xóa nhà sản xuất (chỉ admin)

// @route DELETE /api/manufacturers/:id

// @access Private/Admin

const deleteManufacturer = async (req, res) => {

try {

const manufacturer = await Manufacturer.findById(req.params.id);

if (manufacturer) {

await manufacturer.deleteOne();

res.json({ message: 'Manufacturer removed.' });

} else {

res.status(404).json({ message: 'Manufacturer not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

module.exports = {

getManufacturers,

getManufacturerById,

createManufacturer,

updateManufacturer,

deleteManufacturer,

};

**src/routes/manufacturerRoutes.js**

JavaScript

// src/routes/manufacturerRoutes.js

const express = require('express');

const router = express.Router();

const { getManufacturers, getManufacturerById, createManufacturer, updateManufacturer, deleteManufacturer } = require('../controllers/manufacturerController');

const { protect, isAdmin } = require('../middleware/authMiddleware');

// Public routes

router.get('/', getManufacturers);

router.get('/:id', getManufacturerById);

// Admin routes

router.post('/', protect, isAdmin, createManufacturer);

router.put('/:id', protect, isAdmin, updateManufacturer);

router.delete('/:id', protect, isAdmin, deleteManufacturer);

module.exports = router;

**Thêm vào index.js:** app.use('/api/manufacturers', require('./src/routes/manufacturerRoutes'));

#### 2.4. API Item

* **Model:** src/models/Item.js (đã có)

**src/controllers/itemController.js**

JavaScript

// src/controllers/itemController.js

const Item = require('../models/Item');

const Account = require('../models/Account'); // Để populate addedBy

const Category = require('../models/Category'); // Để populate categoryId

const Manufacturer = require('../models/Manufacturer'); // Để populate manufacturerId

// @desc Lấy tất cả sản phẩm

// @route GET /api/items

// @access Public

const getItems = async (req, res) => {

try {

// Có thể thêm tính năng tìm kiếm, phân trang, lọc ở đây

const items = await Item.find({})

.populate('categoryId', 'name') // Lấy tên danh mục

.populate('manufacturerId', 'name') // Lấy tên nhà sản xuất

.populate('addedBy', 'username fullName'); // Lấy thông tin người thêm

res.json(items);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Lấy chi tiết một sản phẩm

// @route GET /api/items/:id

// @access Public

const getItemById = async (req, res) => {

try {

const item = await Item.findById(req.params.id)

.populate('categoryId', 'name')

.populate('manufacturerId', 'name')

.populate('addedBy', 'username fullName');

if (item) {

res.json(item);

} else {

res.status(404).json({ message: 'Item not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Tạo sản phẩm mới (chỉ admin/manager)

// @route POST /api/items

// @access Private/Admin/Manager

const createItem = async (req, res) => {

const { name, description, price, imageUrl, quantity, categoryId, manufacturerId } = req.body;

try {

const itemExists = await Item.findOne({ name });

if (itemExists) {

return res.status(400).json({ message: 'Item with this name already exists.' });

}

// Kiểm tra xem categoryId và manufacturerId có tồn tại không

const category = await Category.findById(categoryId);

if (!category) {

return res.status(404).json({ message: 'Category not found.' });

}

const manufacturer = await Manufacturer.findById(manufacturerId);

if (!manufacturer) {

return res.status(404).json({ message: 'Manufacturer not found.' });

}

const item = await Item.create({

name,

description,

price,

imageUrl,

quantity,

categoryId,

manufacturerId,

addedBy: req.user.\_id, // Người tạo là người đang đăng nhập

});

res.status(201).json(item);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Cập nhật sản phẩm (chỉ admin/manager)

// @route PUT /api/items/:id

// @access Private/Admin/Manager

const updateItem = async (req, res) => {

const { name, description, price, imageUrl, quantity, categoryId, manufacturerId } = req.body;

try {

const item = await Item.findById(req.params.id);

if (item) {

item.name = name || item.name;

item.description = description || item.description;

item.price = price !== undefined ? price : item.price;

item.imageUrl = imageUrl || item.imageUrl;

item.quantity = quantity !== undefined ? quantity : item.quantity;

item.categoryId = categoryId || item.categoryId;

item.manufacturerId = manufacturerId || item.manufacturerId;

const updatedItem = await item.save();

res.json(updatedItem);

} else {

res.status(404).json({ message: 'Item not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Xóa sản phẩm (chỉ admin/manager)

// @route DELETE /api/items/:id

// @access Private/Admin/Manager

const deleteItem = async (req, res) => {

try {

const item = await Item.findById(req.params.id);

if (item) {

await item.deleteOne();

res.json({ message: 'Item removed.' });

} else {

res.status(404).json({ message: 'Item not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

module.exports = {

getItems,

getItemById,

createItem,

updateItem,

deleteItem,

};

**src/routes/itemRoutes.js**

JavaScript

// src/routes/itemRoutes.js

const express = require('express');

const router = express.Router();

const { getItems, getItemById, createItem, updateItem, deleteItem } = require('../controllers/itemController');

const { protect, isAdmin } = require('../middleware/authMiddleware'); // Cần một middleware isManager hoặc kết hợp isAdmin

// Middleware để kiểm tra admin hoặc manager

const isManagerOrAdmin = (req, res, next) => {

if (req.user && (req.user.roles.includes('admin') || req.user.roles.includes('manager'))) {

next();

} else {

res.status(403).json({ message: 'Not authorized as an admin or manager.' });

}

};

// Public routes

router.get('/', getItems);

router.get('/:id', getItemById);

// Admin/Manager routes

router.post('/', protect, isManagerOrAdmin, createItem);

router.put('/:id', protect, isManagerOrAdmin, updateItem);

router.delete('/:id', protect, isManagerOrAdmin, deleteItem);

module.exports = router;

**Thêm vào index.js:** app.use('/api/items', require('./src/routes/itemRoutes'));

#### 2.5. API Order

* **Model:** src/models/Order.js (đã có)
* **Lưu ý:** API này sẽ xử lý các đơn hàng chung. Các chi tiết về sản phẩm (OrderItems), yêu cầu vận chuyển (ShippingRequest) hoặc đặt xe (RideBooking) sẽ được xử lý trong các API riêng biệt nhưng có liên kết với Order.

**src/controllers/orderController.js**

JavaScript

// src/controllers/orderController.js

const Order = require('../models/Order');

const Account = require('../models/Account');

const OrderItem = require('../models/OrderItem'); // Để tạo/liên kết OrderItems

const Item = require('../models/Item'); // Để giảm số lượng tồn kho

// @desc Tạo đơn hàng mới (cho khách hàng)

// @route POST /api/orders

// @access Private

const createOrder = async (req, res) => {

const { orderType, items, shippingAddress, paymentMethod, notes, pickupLocation, deliveryLocation, requestedVehicleType, numberOfPassengers, itemDescription, itemWeight, itemDimensions } = req.body;

const accountId = req.user.\_id;

try {

let totalPrice = 0;

let orderItems = [];

// Tính toán tổng giá và tạo OrderItems nếu là 'product\_order'

if (orderType === 'product\_order' && items && items.length > 0) {

for (const orderItemData of items) {

const item = await Item.findById(orderItemData.itemId);

if (!item) {

return res.status(404).json({ message: `Item with ID ${orderItemData.itemId} not found.` });

}

if (item.quantity < orderItemData.quantity) {

return res.status(400).json({ message: `Not enough stock for item: ${item.name}. Available: ${item.quantity}` });

}

const subtotal = item.price \* orderItemData.quantity;

totalPrice += subtotal;

// Lưu thông tin chi tiết của OrderItem

orderItems.push({

itemId: item.\_id,

quantity: orderItemData.quantity,

priceAtPurchase: item.price,

subtotal: subtotal,

});

// Giảm số lượng tồn kho của sản phẩm

item.quantity -= orderItemData.quantity;

await item.save();

}

} else if (orderType === 'shipping\_request' || orderType === 'ride\_booking') {

// Đối với dịch vụ vận chuyển/đặt xe, totalPrice có thể được tính toán ở frontend hoặc sau này bởi backend

// Hiện tại có thể đặt 0 hoặc một giá trị ước tính

totalPrice = req.body.fare || 0; // Fare sẽ được tính toán chi tiết trong ShippingRequest/RideBooking

} else {

return res.status(400).json({ message: 'Invalid order type or missing items for product order.' });

}

const order = await Order.create({

accountId,

orderType,

totalPrice,

shippingAddress: orderType === 'product\_order' ? shippingAddress : undefined,

paymentMethod,

notes,

// Các trường khác cho ShippingRequest/RideBooking sẽ được tạo trong các controllers riêng

});

// Tạo OrderItems sau khi Order được tạo

if (orderType === 'product\_order' && orderItems.length > 0) {

const createdOrderItems = await Promise.all(

orderItems.map(async (item) => {

return await OrderItem.create({ ...item, orderId: order.\_id });

})

);

// Có thể thêm ID của OrderItems vào Order nếu cần

}

// Nếu là yêu cầu vận chuyển hoặc đặt xe, tạo bản ghi chi tiết

if (orderType === 'shipping\_request') {

const ShippingRequest = require('../models/ShippingRequest'); // Import tại đây để tránh lỗi circular dependency

await ShippingRequest.create({

orderId: order.\_id,

pickupLocation,

deliveryLocation,

itemDescription,

itemWeight,

itemDimensions,

fare: totalPrice, // Gán tạm fare đã tính

requestStatus: 'pending' // Trạng thái ban đầu

});

} else if (orderType === 'ride\_booking') {

const RideBooking = require('../models/RideBooking'); // Import tại đây

await RideBooking.create({

orderId: order.\_id,

pickupLocation,

destinationLocation,

requestedVehicleType,

numberOfPassengers,

fare: totalPrice, // Gán tạm fare đã tính

requestStatus: 'pending'

});

}

res.status(201).json(order);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Lấy tất cả đơn hàng của người dùng (khách hàng/admin)

// @route GET /api/orders/my

// @access Private

const getMyOrders = async (req, res) => {

try {

const orders = await Order.find({ accountId: req.user.\_id })

.populate('accountId', 'fullName email') // Lấy thông tin người đặt

.sort({ createdAt: -1 }); // Sắp xếp theo thời gian mới nhất

res.json(orders);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Lấy chi tiết một đơn hàng (khách hàng/admin)

// @route GET /api/orders/:id

// @access Private

const getOrderById = async (req, res) => {

try {

const order = await Order.findById(req.params.id)

.populate('accountId', 'fullName email')

.populate({

path: 'orderItems', // Nếu bạn quyết định thêm orderItems vào Order schema

populate: {

path: 'itemId',

select: 'name price imageUrl'

}

});

if (order) {

// Đảm bảo chỉ admin hoặc chủ đơn hàng mới có thể xem

if (req.user.roles.includes('admin') || order.accountId.\_id.toString() === req.user.\_id.toString()) {

res.json(order);

} else {

res.status(403).json({ message: 'Not authorized to view this order.' });

}

} else {

res.status(404).json({ message: 'Order not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Lấy tất cả đơn hàng (chỉ admin/manager)

// @route GET /api/orders

// @access Private/Admin/Manager

const getAllOrders = async (req, res) => {

try {

const orders = await Order.find({})

.populate('accountId', 'fullName email')

.sort({ createdAt: -1 });

res.json(orders);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Cập nhật trạng thái đơn hàng (chỉ admin/manager/shipper - tùy loại đơn)

// @route PUT /api/orders/:id/status

// @access Private/Admin/Manager/Shipper

const updateOrderStatus = async (req, res) => {

const { status } = req.body;

try {

const order = await Order.findById(req.params.id);

if (order) {

// Logic kiểm tra quyền và trạng thái chuyển đổi hợp lệ

if (req.user.roles.includes('admin') || req.user.roles.includes('manager')) {

order.status = status;

} else if (req.user.roles.includes('shipper') && (order.orderType === 'shipping\_request' || order.orderType === 'ride\_booking')) {

// Cho phép shipper cập nhật các trạng thái liên quan đến giao hàng/chuyến đi

// Cần kiểm tra shipperId của đơn hàng có khớp với req.user.\_id hay không

// Và chỉ cho phép chuyển đổi trạng thái hợp lệ (ví dụ: pending -> accepted -> picked\_up -> delivered)

order.status = status; // Cần thêm logic kiểm tra sâu hơn

} else {

return res.status(403).json({ message: 'Not authorized to update this order status.' });

}

const updatedOrder = await order.save();

res.json(updatedOrder);

} else {

res.status(404).json({ message: 'Order not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

module.exports = {

createOrder,

getMyOrders,

getOrderById,

getAllOrders,

updateOrderStatus,

};

**src/routes/orderRoutes.js**

JavaScript

// src/routes/orderRoutes.js

const express = require('express');

const router = express.Router();

const { createOrder, getMyOrders, getOrderById, getAllOrders, updateOrderStatus } = require('../controllers/orderController');

const { protect, isAdmin } = require('../middleware/authMiddleware');

const isManagerOrAdmin = (req, res, next) => {

if (req.user && (req.user.roles.includes('admin') || req.user.roles.includes('manager'))) {

next();

} else {

res.status(403).json({ message: 'Not authorized as an admin or manager.' });

}

};

// Customer/User routes

router.post('/', protect, createOrder); // Tạo đơn hàng

router.get('/my', protect, getMyOrders); // Lấy đơn hàng của tôi

router.get('/:id', protect, getOrderById); // Lấy chi tiết đơn hàng (của mình hoặc nếu là admin)

// Admin/Manager routes

router.get('/', protect, isManagerOrAdmin, getAllOrders); // Lấy tất cả đơn hàng

router.put('/:id/status', protect, updateOrderStatus); // Cập nhật trạng thái (cần kiểm tra quyền trong controller)

module.exports = router;

**Thêm vào index.js:** app.use('/api/orders', require('./src/routes/orderRoutes'));

#### 2.6. API OrderItem

* **Model:** src/models/OrderItem.js (đã có)
* **Lưu ý:** OrderItems thường được tạo/quản lý cùng với Order. Các API riêng cho OrderItem chủ yếu phục vụ việc xem chi tiết hoặc quản lý nội bộ.

**src/controllers/orderItemController.js**

JavaScript

// src/controllers/orderItemController.js

const OrderItem = require('../models/OrderItem');

const Order = require('../models/Order');

const Item = require('../models/Item');

// @desc Lấy tất cả OrderItems của một đơn hàng cụ thể

// @route GET /api/orders/:orderId/items

// @access Private (Chủ đơn hàng, Admin, Manager)

const getOrderItemsByOrderId = async (req, res) => {

try {

const order = await Order.findById(req.params.orderId);

if (!order) {

return res.status(404).json({ message: 'Order not found.' });

}

// Kiểm tra quyền: chỉ chủ đơn hàng, admin hoặc manager mới được xem

if (req.user.roles.includes('admin') || req.user.roles.includes('manager') || order.accountId.toString() === req.user.\_id.toString()) {

const orderItems = await OrderItem.find({ orderId: req.params.orderId })

.populate('itemId', 'name imageUrl price'); // Populate thông tin sản phẩm

res.json(orderItems);

} else {

res.status(403).json({ message: 'Not authorized to view these order items.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Lấy chi tiết một OrderItem (cho mục đích quản lý/debug)

// @route GET /api/orderitems/:id

// @access Private (Admin, Manager)

const getOrderItemById = async (req, res) => {

try {

const orderItem = await OrderItem.findById(req.params.id)

.populate('orderId', 'accountId orderDate')

.populate('itemId', 'name imageUrl price');

if (orderItem) {

// Có thể thêm kiểm tra quyền dựa trên orderId.accountId

res.json(orderItem);

} else {

res.status(404).json({ message: 'OrderItem not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Cập nhật OrderItem (chỉ admin/manager - trong trường hợp sửa đơn hàng)

// @route PUT /api/orderitems/:id

// @access Private/Admin/Manager

const updateOrderItem = async (req, res) => {

const { quantity } = req.body;

try {

const orderItem = await OrderItem.findById(req.params.id);

if (orderItem) {

const oldQuantity = orderItem.quantity;

orderItem.quantity = quantity !== undefined ? quantity : oldQuantity;

const item = await Item.findById(orderItem.itemId);

if (!item) return res.status(404).json({ message: 'Linked item not found.' });

// Điều chỉnh số lượng tồn kho

const quantityDifference = oldQuantity - orderItem.quantity; // Nếu giảm quantity thì stock tăng

item.quantity += quantityDifference;

await item.save();

orderItem.subtotal = item.price \* orderItem.quantity; // Cập nhật subtotal

const updatedOrderItem = await orderItem.save();

res.json(updatedOrderItem);

} else {

res.status(404).json({ message: 'OrderItem not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Xóa OrderItem (chỉ admin/manager)

// @route DELETE /api/orderitems/:id

// @access Private/Admin/Manager

const deleteOrderItem = async (req, res) => {

try {

const orderItem = await OrderItem.findById(req.params.id);

if (orderItem) {

// Tăng lại số lượng tồn kho của sản phẩm khi xóa OrderItem

const item = await Item.findById(orderItem.itemId);

if (item) {

item.quantity += orderItem.quantity;

await item.save();

}

await orderItem.deleteOne();

res.json({ message: 'OrderItem removed.' });

} else {

res.status(404).json({ message: 'OrderItem not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

module.exports = {

getOrderItemsByOrderId,

getOrderItemById,

updateOrderItem,

deleteOrderItem

};

**src/routes/orderItemRoutes.js**

JavaScript

// src/routes/orderItemRoutes.js

const express = require('express');

const router = express.Router();

const { getOrderItemsByOrderId, getOrderItemById, updateOrderItem, deleteOrderItem } = require('../controllers/orderItemController');

const { protect, isAdmin } = require('../middleware/authMiddleware');

const isManagerOrAdmin = (req, res, next) => {

if (req.user && (req.user.roles.includes('admin') || req.user.roles.includes('manager'))) {

next();

} else {

res.status(403).json({ message: 'Not authorized as an admin or manager.' });

}

};

// Lấy các item của một đơn hàng cụ thể

router.get('/order/:orderId', protect, getOrderItemsByOrderId);

// Các thao tác quản lý chi tiết OrderItem (chỉ admin/manager)

router.get('/:id', protect, isManagerOrAdmin, getOrderItemById);

router.put('/:id', protect, isManagerOrAdmin, updateOrderItem);

router.delete('/:id', protect, isManagerOrAdmin, deleteOrderItem);

module.exports = router;

**Thêm vào index.js:** app.use('/api/orderitems', require('./src/routes/orderItemRoutes'));

#### 2.7. API Topic

* **Model:** src/models/Topic.js (đã có)

**src/controllers/topicController.js**

JavaScript

// src/controllers/topicController.js

const Topic = require('../models/Topic');

// @desc Lấy tất cả chủ đề

// @route GET /api/topics

// @access Public

const getTopics = async (req, res) => {

try {

const topics = await Topic.find({});

res.json(topics);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Lấy chi tiết một chủ đề

// @route GET /api/topics/:id

// @access Public

const getTopicById = async (req, res) => {

try {

const topic = await Topic.findById(req.params.id);

if (topic) {

res.json(topic);

} else {

res.status(404).json({ message: 'Topic not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Tạo chủ đề mới (chỉ admin/manager)

// @route POST /api/topics

// @access Private/Admin/Manager

const createTopic = async (req, res) => {

const { name, description } = req.body;

try {

const topicExists = await Topic.findOne({ name });

if (topicExists) {

return res.status(400).json({ message: 'Topic with this name already exists.' });

}

const topic = await Topic.create({ name, description });

res.status(201).json(topic);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Cập nhật chủ đề (chỉ admin/manager)

// @route PUT /api/topics/:id

// @access Private/Admin/Manager

const updateTopic = async (req, res) => {

const { name, description } = req.body;

try {

const topic = await Topic.findById(req.params.id);

if (topic) {

topic.name = name || topic.name;

topic.description = description || topic.description;

const updatedTopic = await topic.save();

res.json(updatedTopic);

} else {

res.status(404).json({ message: 'Topic not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Xóa chủ đề (chỉ admin/manager)

// @route DELETE /api/topics/:id

// @access Private/Admin/Manager

const deleteTopic = async (req, res) => {

try {

const topic = await Topic.findById(req.params.id);

if (topic) {

// Có thể thêm logic kiểm tra xem có bài viết nào đang dùng topic này không

await topic.deleteOne();

res.json({ message: 'Topic removed.' });

} else {

res.status(404).json({ message: 'Topic not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

module.exports = {

getTopics,

getTopicById,

createTopic,

updateTopic,

deleteTopic,

};

**src/routes/topicRoutes.js**

JavaScript

// src/routes/topicRoutes.js

const express = require('express');

const router = express.Router();

const { getTopics, getTopicById, createTopic, updateTopic, deleteTopic } = require('../controllers/topicController');

const { protect, isAdmin } = require('../middleware/authMiddleware');

const isManagerOrAdmin = (req, res, next) => {

if (req.user && (req.user.roles.includes('admin') || req.user.roles.includes('manager'))) {

next();

} else {

res.status(403).json({ message: 'Not authorized as an admin or manager.' });

}

};

// Public routes

router.get('/', getTopics);

router.get('/:id', getTopicById);

// Admin/Manager routes

router.post('/', protect, isManagerOrAdmin, createTopic);

router.put('/:id', protect, isManagerOrAdmin, updateTopic);

router.delete('/:id', protect, isManagerOrAdmin, deleteTopic);

module.exports = router;

**Thêm vào index.js:** app.use('/api/topics', require('./src/routes/topicRoutes'));

#### 2.8. API Article

* **Model:** src/models/Article.js (đã có)

**src/controllers/articleController.js**

JavaScript

// src/controllers/articleController.js

const Article = require('../models/Article');

const Topic = require('../models/Topic');

const Account = require('../models/Account');

// @desc Lấy tất cả bài viết (có thể lọc theo censorStatus cho public)

// @route GET /api/articles

// @access Public

const getArticles = async (req, res) => {

try {

// Chỉ lấy các bài viết đã được duyệt cho người dùng thông thường

const query = req.user && (req.user.roles.includes('admin') || req.user.roles.includes('manager')) ? {} : { censorStatus: 'approved' };

const articles = await Article.find(query)

.populate('topicId', 'name')

.populate('authorId', 'fullName username')

.sort({ publishDate: -1 });

res.json(articles);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Lấy chi tiết một bài viết và tăng lượt xem

// @route GET /api/articles/:id

// @access Public

const getArticleById = async (req, res) => {

try {

const article = await Article.findById(req.params.id)

.populate('topicId', 'name')

.populate('authorId', 'fullName username');

if (article) {

// Tăng lượt xem (chỉ khi truy cập bởi người dùng thông thường hoặc không phải admin/manager chỉnh sửa)

if (!(req.user && (req.user.roles.includes('admin') || req.user.roles.includes('manager')))) {

article.viewCount += 1;

await article.save();

}

res.json(article);

} else {

res.status(404).json({ message: 'Article not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Tạo bài viết mới (chỉ admin/manager/có thể là user với quyền đặc biệt)

// @route POST /api/articles

// @access Private/Admin/Manager

const createArticle = async (req, res) => {

const { title, summary, content, topicId } = req.body;

try {

const articleExists = await Article.findOne({ title });

if (articleExists) {

return res.status(400).json({ message: 'Article with this title already exists.' });

}

const topic = await Topic.findById(topicId);

if (!topic) {

return res.status(404).json({ message: 'Topic not found.' });

}

const article = await Article.create({

title,

summary,

content,

topicId,

authorId: req.user.\_id,

censorStatus: req.user.roles.includes('admin') ? 'approved' : 'pending', // Admin tự duyệt, người khác cần duyệt

});

res.status(201).json(article);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Cập nhật bài viết (chỉ admin/manager hoặc chính tác giả)

// @route PUT /api/articles/:id

// @access Private/Admin/Manager/Author

const updateArticle = async (req, res) => {

const { title, summary, content, topicId, censorStatus, publishDate } = req.body;

try {

const article = await Article.findById(req.params.id);

if (article) {

// Kiểm tra quyền: admin/manager hoặc chính tác giả

if (req.user.roles.includes('admin') || req.user.roles.includes('manager') || article.authorId.toString() === req.user.\_id.toString()) {

article.title = title || article.title;

article.summary = summary || article.summary;

article.content = content || article.content;

article.topicId = topicId || article.topicId;

article.lastEditedDate = Date.now();

// Chỉ admin/manager mới được thay đổi censorStatus và publishDate

if (req.user.roles.includes('admin') || req.user.roles.includes('manager')) {

article.censorStatus = censorStatus || article.censorStatus;

article.publishDate = publishDate || article.publishDate;

}

const updatedArticle = await article.save();

res.json(updatedArticle);

} else {

res.status(403).json({ message: 'Not authorized to update this article.' });

}

} else {

res.status(404).json({ message: 'Article not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Xóa bài viết (chỉ admin/manager)

// @route DELETE /api/articles/:id

// @access Private/Admin/Manager

const deleteArticle = async (req, res) => {

try {

const article = await Article.findById(req.params.id);

if (article) {

await article.deleteOne();

res.json({ message: 'Article removed.' });

} else {

res.status(404).json({ message: 'Article not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

module.exports = {

getArticles,

getArticleById,

createArticle,

updateArticle,

deleteArticle,

};

**src/routes/articleRoutes.js**

JavaScript

// src/routes/articleRoutes.js

const express = require('express');

const router = express.Router();

const { getArticles, getArticleById, createArticle, updateArticle, deleteArticle } = require('../controllers/articleController');

const { protect, isAdmin } = require('../middleware/authMiddleware');

const isManagerOrAdmin = (req, res, next) => {

if (req.user && (req.user.roles.includes('admin') || req.user.roles.includes('manager'))) {

next();

} else {

res.status(403).json({ message: 'Not authorized as an admin or manager.' });

}

};

// Public routes (có thể hiển thị bài viết đã duyệt)

router.get('/', getArticles);

router.get('/:id', getArticleById);

// Protected routes (cho người tạo, admin, manager)

router.post('/', protect, isManagerOrAdmin, createArticle); // Có thể cho phép user thông thường tạo, sau đó cần duyệt

router.put('/:id', protect, updateArticle); // Admin/Manager hoặc tác giả

router.delete('/:id', protect, isManagerOrAdmin, deleteArticle); // Chỉ admin/manager

module.exports = router;

**Thêm vào index.js:** app.use('/api/articles', require('./src/routes/articleRoutes'));

#### 2.9. API ShipperProfile

* **Model:** src/models/ShipperProfile.js (đã có)

**src/controllers/shipperProfileController.js**

JavaScript

// src/controllers/shipperProfileController.js

const ShipperProfile = require('../models/ShipperProfile');

const Account = require('../models/Account');

// @desc Đăng ký hồ sơ tài xế (dành cho user muốn trở thành shipper)

// @route POST /api/shippers/profile

// @access Private

const createShipperProfile = async (req, res) => {

const { vehicleType, licensePlate, currentLocation } = req.body;

const accountId = req.user.\_id;

try {

// Kiểm tra xem user đã có hồ sơ shipper chưa

const existingProfile = await ShipperProfile.findOne({ accountId });

if (existingProfile) {

return res.status(400).json({ message: 'User already has a shipper profile.' });

}

// Cần đảm bảo tài khoản này có thể trở thành shipper (ví dụ: vai trò 'customer' được nâng cấp)

const account = await Account.findById(accountId);

if (!account.roles.includes('shipper')) {

account.roles.push('shipper'); // Thêm vai trò shipper

await account.save();

}

const shipperProfile = await ShipperProfile.create({

accountId,

vehicleType,

licensePlate,

currentLocation,

isAvailable: true, // Mặc định sẵn sàng

});

res.status(201).json(shipperProfile);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Lấy hồ sơ tài xế của người dùng hiện tại

// @route GET /api/shippers/profile/me

// @access Private

const getMyShipperProfile = async (req, res) => {

try {

const shipperProfile = await ShipperProfile.findOne({ accountId: req.user.\_id }).populate('accountId', 'fullName email username');

if (shipperProfile) {

res.json(shipperProfile);

} else {

res.status(404).json({ message: 'Shipper profile not found for this user.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Cập nhật hồ sơ tài xế (chỉ chủ hồ sơ hoặc admin)

// @route PUT /api/shippers/profile/:id

// @access Private/Admin/Shipper

const updateShipperProfile = async (req, res) => {

const { vehicleType, licensePlate, currentLocation, isAvailable } = req.body;

try {

const shipperProfile = await ShipperProfile.findById(req.params.id);

if (shipperProfile) {

// Chỉ chủ hồ sơ hoặc admin mới được cập nhật

if (req.user.roles.includes('admin') || shipperProfile.accountId.toString() === req.user.\_id.toString()) {

shipperProfile.vehicleType = vehicleType || shipperProfile.vehicleType;

shipperProfile.licensePlate = licensePlate || shipperProfile.licensePlate;

shipperProfile.currentLocation = currentLocation || shipperProfile.currentLocation;

shipperProfile.isAvailable = typeof isAvailable === 'boolean' ? isAvailable : shipperProfile.isAvailable;

const updatedProfile = await shipperProfile.save();

res.json(updatedProfile);

} else {

res.status(403).json({ message: 'Not authorized to update this shipper profile.' });

}

} else {

res.status(404).json({ message: 'Shipper profile not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Lấy tất cả hồ sơ tài xế (chỉ admin/manager)

// @route GET /api/shippers/profile

// @access Private/Admin/Manager

const getAllShipperProfiles = async (req, res) => {

try {

const shipperProfiles = await ShipperProfile.find({}).populate('accountId', 'fullName email username roles');

res.json(shipperProfiles);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Xóa hồ sơ tài xế (chỉ admin)

// @route DELETE /api/shippers/profile/:id

// @access Private/Admin

const deleteShipperProfile = async (req, res) => {

try {

const shipperProfile = await ShipperProfile.findById(req.params.id);

if (shipperProfile) {

// Có thể thêm logic gỡ vai trò 'shipper' khỏi Account

await shipperProfile.deleteOne();

res.json({ message: 'Shipper profile removed.' });

} else {

res.status(404).json({ message: 'Shipper profile not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

module.exports = {

createShipperProfile,

getMyShipperProfile,

updateShipperProfile,

getAllShipperProfiles,

deleteShipperProfile,

};

**src/routes/shipperProfileRoutes.js**

JavaScript

// src/routes/shipperProfileRoutes.js

const express = require('express');

const router = express.Router();

const { createShipperProfile, getMyShipperProfile, updateShipperProfile, getAllShipperProfiles, deleteShipperProfile } = require('../controllers/shipperProfileController');

const { protect, isAdmin } = require('../middleware/authMiddleware');

const isManagerOrAdmin = (req, res, next) => {

if (req.user && (req.user.roles.includes('admin') || req.user.roles.includes('manager'))) {

next();

} else {

res.status(403).json({ message: 'Not authorized as an admin or manager.' });

}

};

// Shipper/User routes

router.post('/', protect, createShipperProfile); // Đăng ký profile shipper

router.get('/me', protect, getMyShipperProfile); // Lấy profile của mình

router.put('/:id', protect, updateShipperProfile); // Cập nhật profile của mình (hoặc admin cập nhật)

// Admin/Manager routes

router.get('/', protect, isManagerOrAdmin, getAllShipperProfiles); // Lấy tất cả profile shipper

router.delete('/:id', protect, isAdmin, deleteShipperProfile); // Xóa profile shipper

module.exports = router;

**Thêm vào index.js:** app.use('/api/shippers/profile', require('./src/routes/shipperProfileRoutes'));

#### 2.10. API ShippingRequest

* **Model:** src/models/ShippingRequest.js (đã có)
* **Lưu ý:** API này sẽ tạo các yêu cầu vận chuyển hàng hóa, liên kết với một Order.

**src/controllers/shippingRequestController.js**

JavaScript

// src/controllers/shippingRequestController.js

const ShippingRequest = require('../models/ShippingRequest');

const Order = require('../models/Order');

const ShipperProfile = require('../models/ShipperProfile');

// @desc Lấy tất cả yêu cầu vận chuyển (có thể lọc theo trạng thái, vị trí)

// @route GET /api/shipping-requests

// @access Public (để hiển thị cho tài xế tìm đơn) / Private (admin/manager)

const getShippingRequests = async (req, res) => {

try {

// Có thể thêm query params để lọc theo status, location, shipperId, v.v.

const requests = await ShippingRequest.find({})

.populate('orderId', 'accountId orderDate status')

.populate('shipperId', 'accountId vehicleType currentLocation');

res.json(requests);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Lấy chi tiết yêu cầu vận chuyển

// @route GET /api/shipping-requests/:id

// @access Private (Chủ yêu cầu, Shipper được gán, Admin, Manager)

const getShippingRequestById = async (req, res) => {

try {

const request = await ShippingRequest.findById(req.params.id)

.populate('orderId', 'accountId orderDate status')

.populate('shipperId', 'accountId vehicleType currentLocation');

if (request) {

// Kiểm tra quyền

const order = await Order.findById(request.orderId.\_id);

if (req.user.roles.includes('admin') || req.user.roles.includes('manager') ||

(order && order.accountId.toString() === req.user.\_id.toString()) ||

(request.shipperId && request.shipperId.accountId.toString() === req.user.\_id.toString())

) {

res.json(request);

} else {

res.status(403).json({ message: 'Not authorized to view this shipping request.' });

}

} else {

res.status(404).json({ message: 'Shipping request not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Cập nhật yêu cầu vận chuyển (admin/manager, hoặc shipper cập nhật trạng thái)

// @route PUT /api/shipping-requests/:id

// @access Private/Admin/Manager/Shipper

const updateShippingRequest = async (req, res) => {

const { requestStatus, shipperId, fare, estimatedTime } = req.body;

try {

const request = await ShippingRequest.findById(req.params.id);

if (request) {

// Chỉ admin/manager hoặc shipper được gán mới có thể cập nhật

if (req.user.roles.includes('admin') || req.user.roles.includes('manager') ||

(request.shipperId && request.shipperId.toString() === req.user.shipperProfile.\_id.toString())

) {

request.requestStatus = requestStatus || request.requestStatus;

request.shipperId = shipperId || request.shipperId;

request.fare = fare || request.fare;

request.estimatedTime = estimatedTime || request.estimatedTime;

// Cập nhật trạng thái của Order liên quan

const order = await Order.findById(request.orderId);

if (order) {

order.status = requestStatus; // Đồng bộ trạng thái

await order.save();

}

const updatedRequest = await request.save();

res.json(updatedRequest);

} else {

res.status(403).json({ message: 'Not authorized to update this shipping request.' });

}

} else {

res.status(404).json({ message: 'Shipping request not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Xóa yêu cầu vận chuyển (chỉ admin/manager)

// @route DELETE /api/shipping-requests/:id

// @access Private/Admin/Manager

const deleteShippingRequest = async (req, res) => {

try {

const request = await ShippingRequest.findById(req.params.id);

if (request) {

// Cũng cần xóa Order liên quan nếu Order đó chỉ phục vụ cho request này

await request.deleteOne();

res.json({ message: 'Shipping request removed.' });

} else {

res.status(404).json({ message: 'Shipping request not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

module.exports = {

getShippingRequests,

getShippingRequestById,

updateShippingRequest,

deleteShippingRequest

};

**src/routes/shippingRequestRoutes.js**

JavaScript

// src/routes/shippingRequestRoutes.js

const express = require('express');

const router = express.Router();

const { getShippingRequests, getShippingRequestById, updateShippingRequest, deleteShippingRequest } = require('../controllers/shippingRequestController');

const { protect, isAdmin } = require('../middleware/authMiddleware');

const isManagerOrAdmin = (req, res, next) => {

if (req.user && (req.user.roles.includes('admin') || req.user.roles.includes('manager'))) {

next();

} else {

res.status(403).json({ message: 'Not authorized as an admin or manager.' });

}

};

// Public/Shipper routes (tùy thuộc vào chính sách hiển thị đơn cho shipper)

router.get('/', protect, getShippingRequests); // Cần protect nếu chỉ shipper/admin được xem tất cả

router.get('/:id', protect, getShippingRequestById);

// Admin/Manager/Shipper routes

router.put('/:id', protect, updateShippingRequest); // Logic kiểm tra quyền trong controller

router.delete('/:id', protect, isAdmin, deleteShippingRequest); // Chỉ admin

module.exports = router;

**Thêm vào index.js:** app.use('/api/shipping-requests', require('./src/routes/shippingRequestRoutes'));

#### 2.11. API RideBooking

* **Model:** src/models/RideBooking.js (đã có)
* **Lưu ý:** Tương tự ShippingRequest, API này tạo các yêu cầu đặt xe cho người, liên kết với một Order.

**src/controllers/rideBookingController.js**

JavaScript

// src/controllers/rideBookingController.js

const RideBooking = require('../models/RideBooking');

const Order = require('../models/Order');

const ShipperProfile = require('../models/ShipperProfile');

// @desc Lấy tất cả yêu cầu đặt xe

// @route GET /api/ride-bookings

// @access Public (cho tài xế) / Private (admin/manager)

const getRideBookings = async (req, res) => {

try {

const bookings = await RideBooking.find({})

.populate('orderId', 'accountId orderDate status')

.populate('shipperId', 'accountId vehicleType currentLocation');

res.json(bookings);

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Lấy chi tiết yêu cầu đặt xe

// @route GET /api/ride-bookings/:id

// @access Private (Chủ yêu cầu, Shipper được gán, Admin, Manager)

const getRideBookingById = async (req, res) => {

try {

const booking = await RideBooking.findById(req.params.id)

.populate('orderId', 'accountId orderDate status')

.populate('shipperId', 'accountId vehicleType currentLocation');

if (booking) {

// Kiểm tra quyền tương tự ShippingRequest

const order = await Order.findById(booking.orderId.\_id);

if (req.user.roles.includes('admin') || req.user.roles.includes('manager') ||

(order && order.accountId.toString() === req.user.\_id.toString()) ||

(booking.shipperId && booking.shipperId.accountId.toString() === req.user.\_id.toString())

) {

res.json(booking);

} else {

res.status(403).json({ message: 'Not authorized to view this ride booking.' });

}

} else {

res.status(404).json({ message: 'Ride booking not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Cập nhật yêu cầu đặt xe

// @route PUT /api/ride-bookings/:id

// @access Private/Admin/Manager/Shipper

const updateRideBooking = async (req, res) => {

const { requestStatus, shipperId, fare, estimatedArrivalTime, estimatedTravelTime } = req.body;

try {

const booking = await RideBooking.findById(req.params.id);

if (booking) {

// Kiểm tra quyền tương tự ShippingRequest

if (req.user.roles.includes('admin') || req.user.roles.includes('manager') ||

(booking.shipperId && booking.shipperId.toString() === req.user.shipperProfile.\_id.toString())

) {

booking.requestStatus = requestStatus || booking.requestStatus;

booking.shipperId = shipperId || booking.shipperId;

booking.fare = fare || booking.fare;

booking.estimatedArrivalTime = estimatedArrivalTime || booking.estimatedArrivalTime;

booking.estimatedTravelTime = estimatedTravelTime || booking.estimatedTravelTime;

// Cập nhật trạng thái của Order liên quan

const order = await Order.findById(booking.orderId);

if (order) {

order.status = requestStatus; // Đồng bộ trạng thái

await order.save();

}

const updatedBooking = await booking.save();

res.json(updatedBooking);

} else {

res.status(403).json({ message: 'Not authorized to update this ride booking.' });

}

} else {

res.status(404).json({ message: 'Ride booking not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

// @desc Xóa yêu cầu đặt xe (chỉ admin/manager)

// @route DELETE /api/ride-bookings/:id

// @access Private/Admin/Manager

const deleteRideBooking = async (req, res) => {

try {

const booking = await RideBooking.findById(req.params.id);

if (booking) {

// Cũng cần xóa Order liên quan nếu Order đó chỉ phục vụ cho booking này

await booking.deleteOne();

res.json({ message: 'Ride booking removed.' });

} else {

res.status(404).json({ message: 'Ride booking not found.' });

}

} catch (error) {

res.status(500).json({ message: 'Server error.', error: error.message });

}

};

module.exports = {

getRideBookings,

getRideBookingById,

updateRideBooking,

deleteRideBooking

};

**src/routes/rideBookingRoutes.js**

JavaScript

// src/routes/rideBookingRoutes.js

const express = require('express');

const router = express.Router();

const { getRideBookings, getRideBookingById, updateRideBooking, deleteRideBooking } = require('../controllers/rideBookingController');

const { protect, isAdmin } = require('../middleware/authMiddleware');

const isManagerOrAdmin = (req, res, next) => {

if (req.user && (req.user.roles.includes('admin') || req.user.roles.includes('manager'))) {

next();

} else {

res.status(403).json({ message: 'Not authorized as an admin or manager.' });

}

};

// Public/Shipper routes

router.get('/', protect, getRideBookings);

router.get('/:id', protect, getRideBookingById);

// Admin/Manager/Shipper routes

router.put('/:id', protect, updateRideBooking);

router.delete('/:id', protect, isAdmin, deleteRideBooking);

module.exports = router;

**Thêm vào index.js:** app.use('/api/ride-bookings', require('./src/routes/rideBookingRoutes'));

### 3. Bảo mật (Đã tích hợp)

Các bước bảo mật đã được tích hợp trực tiếp vào các API bạn vừa thấy:

* **Mã hóa mật khẩu:** Được xử lý trong model Account.js bằng bcryptjs khi lưu (AccountSchema.pre('save', ...)).
* **Xác thực (Authentication) với JWT:**
  + **Tạo Token:** Khi người dùng đăng nhập thành công (loginAccount controller), một JWT token sẽ được tạo bằng jsonwebtoken.
  + **Xác thực Middleware:** Middleware protect (src/middleware/authMiddleware.js) được sử dụng trên các route cần bảo vệ. Nó giải mã token từ header của yêu cầu và đính kèm thông tin người dùng vào req.user.
  + **Phân quyền (Authorization):** Middleware isAdmin và isManagerOrAdmin được dùng để hạn chế quyền truy cập vào các route cụ thể (ví dụ: tạo, cập nhật, xóa sản phẩm/danh mục/người dùng chỉ dành cho admin/manager). Logic phân quyền chi tiết cũng được tích hợp ngay trong các controller.

### Các bước tiếp theo sau khi hoàn thành API:

1. **Tạo các file và thư mục:** Đảm bảo bạn đã tạo tất cả các file .js mới trong các thư mục src/controllers và src/routes như đã hướng dẫn.
2. **Cập nhật index.js:** Đảm bảo tất cả các dòng app.use('/api/...', require('./src/routes/...Routes')); đã được thêm vào file index.js của bạn.
3. **Khởi động lại Server:** Chạy npm run dev để khởi động lại server của bạn với tất cả các API mới.
4. **Kiểm thử với Postman/Insomnia:** Sử dụng các công cụ này để gửi các yêu cầu HTTP (GET, POST, PUT, DELETE) tới các endpoint API mới để đảm bảo chúng hoạt động đúng như mong đợi.

Bạn đã có một backend mạnh mẽ và có tổ chức cho UrbanFlows! Bước tiếp theo sẽ là xây dựng phần Frontend với Next.js và tích hợp với các API này.