VIETNAM NATIONAL UNIVERSITY OF HO CHI MINH CITY

INTERNATIONAL UNIVERSITY

SCHOOL OF COMPUTER SCIENCE AND ENGINEERING

CAR E-COMMERCE WEBSITE

Web Application Development

Final Project Report

Members:

Pham Anh Khoi – ITCSIU23018
Nguyen Vu Thanh Tinh – ITCSIU23039
Hoang An Thien – ITCSIU23035

December 21, 2025

Contents

1 Introduction

This section introduces the Apex Car Dealership Platform, a comprehensive web-based e-commerce system engineered to facilitate online vehicle browsing, comparison, and purchasing. The platform implements a modern three-tier architecture with clear separation of concerns between the presentation, business logic, and data persistence layers.

1.1 About Us

The Apex project is developed by an undergraduate team specializing in software engineering and web application development. Our objective is to apply modern software design principles and industry-standard technologies to build a robust, maintainable, and scalable web-based system.

The development emphasizes clean architecture, adherence to design patterns, and practical implementation of e-commerce workflows. The target users include automotive dealerships and small-to-medium enterprises seeking digital platforms for online vehicle sales management.

1.2 Product Overview

Apex is a full-stack e-commerce application designed for automotive dealerships, providing an end-to-end solution for vehicle discovery, comparison, and acquisition. The platform encompasses the following core modules:

- **Vehicle Catalog**: Hierarchical organization of vehicles by brands and models with comprehensive specifications, pricing, and media assets stored in object storage (MinIO).
- **Model Comparison**: Side-by-side comparison engine enabling users to evaluate multiple vehicle models based on configurable attributes.
- **Shopping Cart & Checkout**: Stateful cart management with multi-step checkout workflow integrating payment processing and order creation.
- **Order Management**: State machine-driven order lifecycle with defined transitions (Pending $\rightarrow$ Confirmed $\rightarrow$ Delivering $\rightarrow$ Delivered).
- **Test Drive Booking**: Slot-based scheduling system with availability management and booking confirmation.
- **User Dashboard**: Consolidated interface for profile management, order history, and booking records.
- **Authentication & Authorization**: JWT-based authentication with role-based access control (RBAC) supporting User and Admin roles.

1.3 Work Breakdown Structure

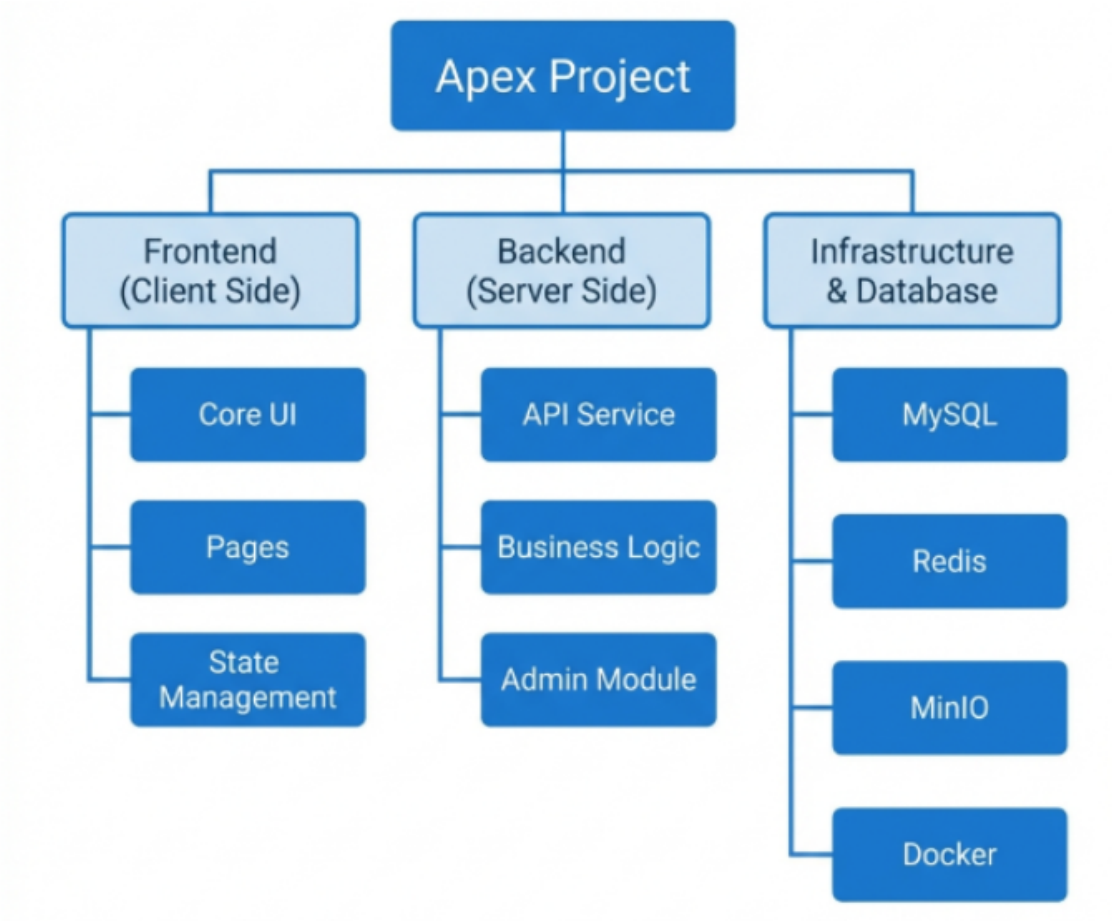

Figure 1: Work Breakdown Structure

The project is decomposed into three primary modules following a modular development approach:

- **Frontend (Presentation Layer)**: React 19-based SPA with TypeScript, implementing reusable UI components, custom hooks for state management, and service modules for API communication.
- **Backend (Application Layer)**: FastAPI-based RESTful API implementing business logic through a layered service architecture with Pydantic schema validation.
- **Infrastructure (Data Layer)**: Containerized services including MySQL for relational data, Redis for caching and OTP storage, and MinIO for object storage.

This decomposition enables parallel development, independent testing, and modular deployment of each subsystem.

1.4 Development Process

Development follows an iterative Agile methodology with the following phases:

1. **Requirements & Design**: Analysis of functional requirements, database schema design using Entity-Relationship modeling, and API contract specification.
2. **Environment Setup**: Docker Compose orchestration of MySQL, Redis, and MinIO services ensuring consistent development environments.

3. **Implementation**: Parallel frontend and backend development with continuous integration of API endpoints and UI components.

4. **Integration & Testing**: End-to-end testing of data flows, authentication mechanisms, and business workflows.

5. **Refinement**: Iterative optimization based on testing feedback and code review.

1.5 Technology Stack

The system employs a modern technology stack organized by architectural layer:

Layer	Technology	Purpose
Presentation	React 19, TypeScript, Vite TailwindCSS, Axios	Single Page Application Styling, HTTP Client
API	FastAPI, Pydantic Uvicorn	REST API, Schema Validation ASGI Server
Business Logic	Python 3.12	Service Layer Implementation
Data Access	SQLAlchemy ORM	Object-Relational Mapping
Infrastructure	MySQL 8 Redis 7 MinIO Docker Compose	Relational Database Caching, Session Storage S3-Compatible Object Storage Container Orchestration
Security	PyJWT	Token-based Authentication

Table 1: Technology Stack by Architectural Layer

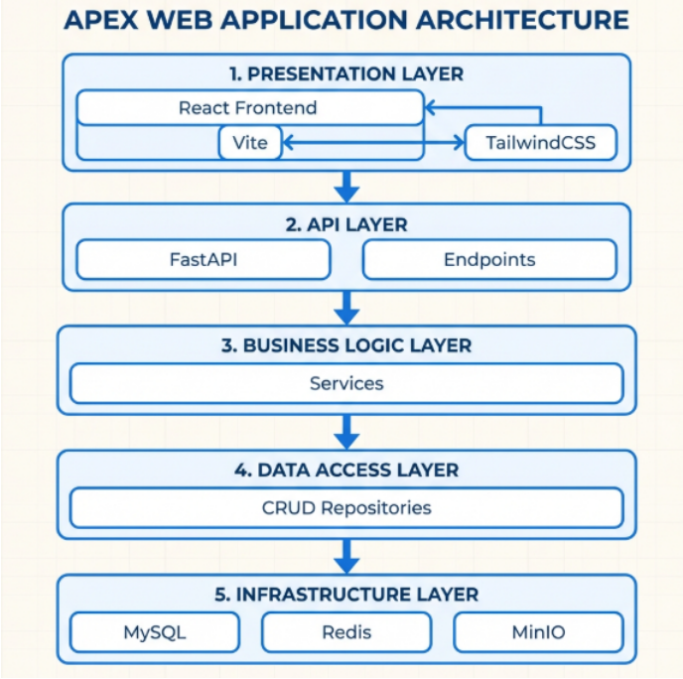

Figure 2: System Architecture Overview

2 Architecture Design

2.1 Use Case Diagram

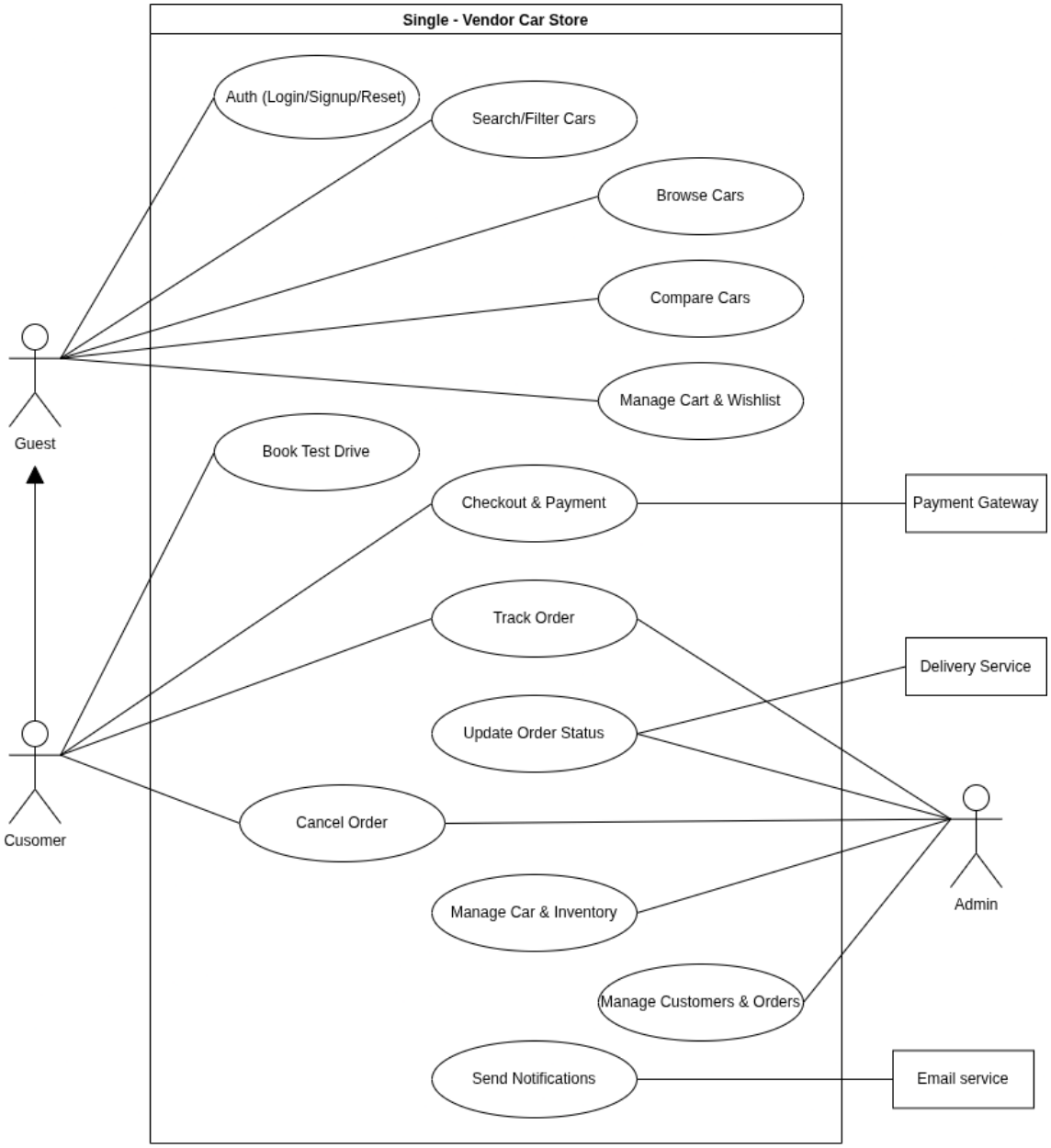

Figure 3: Use Case Diagram

The use case diagram illustrates a role-based access model with four actor types:

- **Guest**: Unauthenticated users with access to public features (browsing, searching, filtering, comparing vehicles, and authentication actions).
- **Customer**: Authenticated users inheriting Guest capabilities plus transactional features (cart management, checkout, order tracking, test drive booking, and profile management).
- **Admin**: System operators with management capabilities including inventory control, order status updates, and user administration.

- **External Systems**: Loosely-coupled integrations including Payment Gateway, Delivery Service, and Email Service.

This design enforces the principle of least privilege, ensuring users only access features appropriate to their role level.

2.2 Backend Architecture

The backend implements a *Layered Architecture* pattern, enforcing unidirectional dependencies where each layer only communicates with adjacent layers. This design promotes separation of concerns, testability, and maintainability.

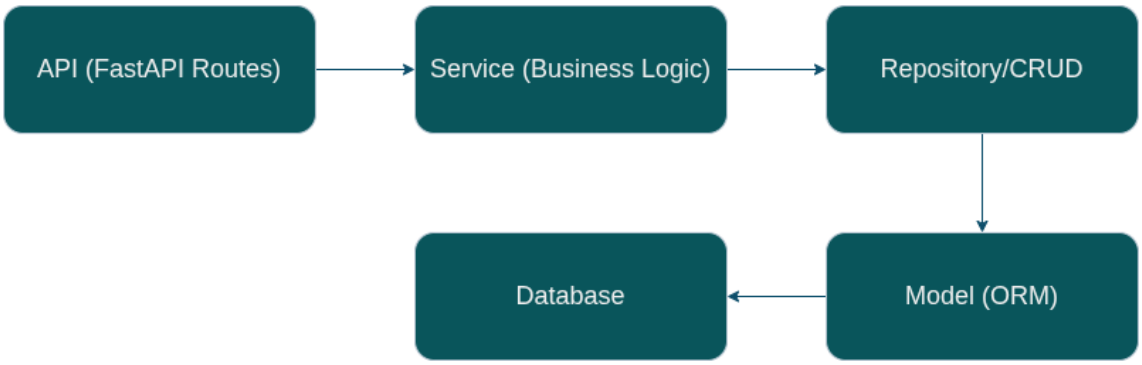

Figure 4: Request Flow Through Architectural Layers

2.2.1 Layer Responsibilities

API Layer (`api/`) Defines FastAPI route handlers that receive HTTP requests, extract and validate input using Pydantic schemas, and delegate processing to the service layer. This layer implements:

- Dependency injection for database sessions via `get_db()`
- Authentication dependencies via `get_current_user()` and role checkers
- Request/response serialization through Pydantic models

Service Layer (`services/`) Contains business logic orchestration, workflow coordination, and transaction management. Services are organized by domain:

- `auth/`: Authentication, token management, OTP handling
- `car/`: Model filtering, comparison logic
- `cart/`: Shopping cart operations
- `checkout/`: Order placement workflow
- `order/`: Order lifecycle and state transitions

Repository Layer (`crud/`) Implements the Repository Pattern, encapsulating all database operations. The `CRUDBase` generic class provides reusable CRUD operations:

```
class CRUDBase(Generic[ModelType, CreateSchemaType, UpdateSchemaType]):
    def get(self, db: Session, id: Any) -> Optional[ModelType]
    def get_multi(self, db: Session, *, skip: int, limit: int) -> List[ModelType]
    def create(self, db: Session, input_object: CreateSchemaType) -> ModelType
    def update(self, db: Session, *, db_object, input_object) -> ModelType
    def remove(self, db: Session, *, id: int) -> ModelType
```

Model Layer (`models/`) Defines SQLAlchemy ORM models representing database entities with relationships, constraints, and cascade behaviors.

Schema Layer (`schemas/`) Contains Pydantic models serving as Data Transfer Objects (DTOs) for request validation, response serialization, and API contracts.

Core Layer (`core/`) Provides cross-cutting concerns including application configuration, enumeration definitions, CORS setup, and security utilities.

2.3 Design Patterns

The system implements several established design patterns to promote code quality and maintainability:

2.3.1 Repository Pattern

All database operations are abstracted through repository classes in the `crud/` module. This pattern:

- Decouples business logic from persistence implementation
- Enables unit testing with mock repositories
- Facilitates potential database technology changes

2.3.2 State Machine Pattern

Order lifecycle management implements a finite state machine with explicit transition rules:

```
TRANSITION_RULES = {
    (OrderStatus.PENDING, OrderAction.ADMIN_CONFIRM): OrderStatus.CONFIRMED,
    (OrderStatus.PENDING, OrderAction.CANCEL): OrderStatus.CANCELLED,
    (OrderStatus.CONFIRMED, OrderAction.ADMIN_SHIP): OrderStatus.DELIVERING,
    (OrderStatus.CONFIRMED, OrderAction.CANCEL): OrderStatus.CANCELLED,
    (OrderStatus.DELIVERING, OrderAction.USER_CONFIRM_DELIVERY): OrderStatus.DELIVERED,
}
```

This pattern ensures only valid state transitions occur and provides clear documentation of the order workflow.

2.3.3 Strategy Pattern

The car filtering system implements the Strategy Pattern through `CarFilterStrategy`, separating filter validation and query construction from the service logic:

- `validate()`: Validates filter parameter consistency
- `build_spec()`: Constructs SQLAlchemy query based on filter criteria

2.3.4 Singleton Pattern

Service instances are created as module-level singletons (e.g., `model_service = ModelService()`), ensuring consistent state and resource efficiency across the application.

2.3.5 Dependency Injection

FastAPI's `Depends()` mechanism enables dependency injection for:

- Database session management
- Authentication and authorization
- Request parameter parsing

2.4 Authentication & Authorization

The system implements a JWT-based authentication mechanism with role-based access control:

2.4.1 Token Types

- **Access Token**: Short-lived token for API authentication
- **Refresh Token**: Long-lived token for obtaining new access tokens
- **Pending Auth Token**: Temporary token for two-factor authentication flow

2.4.2 Role-Based Access Control

The `RoleChecker` class validates user roles against required permissions:

```
require_admin = RoleChecker([UserRole.ADMIN])
require_user = RoleChecker([UserRole.USER])
require_any_logged_in_user = RoleChecker([UserRole.USER, UserRole.ADMIN])
```

2.4.3 Two-Factor Authentication

Optional 2FA support using Redis-backed OTP storage with configurable expiration.

2.5 Database Design

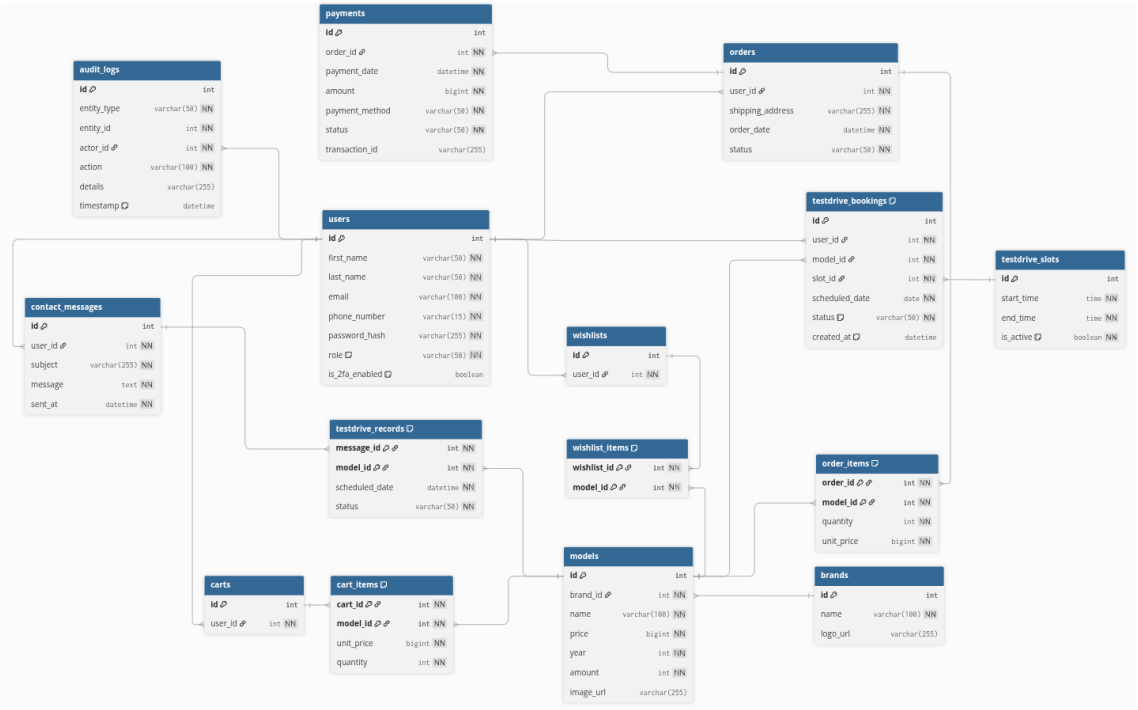

Figure 5: Entity-Relationship Diagram

The relational schema is organized into the following entity groups:

2.5.1 User Management

- `users`: Account information with role enumeration (USER, ADMIN), authentication credentials, and 2FA settings.

2.5.2 Product Catalog

- `brands`: Automotive brand entities
- `models`: Vehicle models with foreign key to brands, including pricing, inventory (amount), and image URLs

2.5.3 Commerce

- `carts` / `cart_items`: User shopping carts with quantity and unit price tracking
- `orders` / `order_items`: Purchase orders with status enumeration and itemized line items
- `payments`: Payment records linked to orders with method and status tracking
- `wishlists` / `wishlist_items`: User wish lists for saved vehicles

2.5.4 Engagement

- `testdrive_slots`: Available time slots for test drive scheduling
- `testdrive_bookings`: User bookings with unique constraint preventing double-booking
- `contact_messages`: Customer communication records

2.5.5 Audit

- `audit_logs`: System-wide audit trail recording entity changes, actor identification, action type, and timestamps

3 Workflow Analysis

3.1 Request-Response Flow

The system follows a standardized request-response pattern across all features:

1. **User Interaction**: UI components capture user actions and form data
2. **Hook Processing**: Custom React hooks manage state and invoke API services
3. **HTTP Request**: Axios-based service modules send requests to backend endpoints
4. **API Validation**: FastAPI endpoints validate requests using Pydantic schemas
5. **Business Logic**: Service layer processes business rules and orchestrates operations
6. **Data Persistence**: CRUD repositories execute database operations via SQLAlchemy
7. **Response**: Structured response propagates back through layers to update UI

3.2 Frontend Architecture

The frontend implements a component-based architecture with clear separation of concerns:

- **Components** (`src/components/`): Reusable UI elements organized by feature domain
- **Hooks** (`src/hooks/`): Custom hooks encapsulating state management, API calls, and side effects
- **Services** (`src/services/`): API abstraction layer with Axios HTTP client configuration
- **Types** (`src/types/`): TypeScript interface definitions ensuring type safety

3.2.1 HTTP Client Configuration

The `http.ts` module configures Axios with:

- Automatic JWT token attachment via request interceptors
- Transparent token refresh handling via response interceptors
- Request queuing during token refresh to prevent race conditions

3.3 Case Study: Model Management

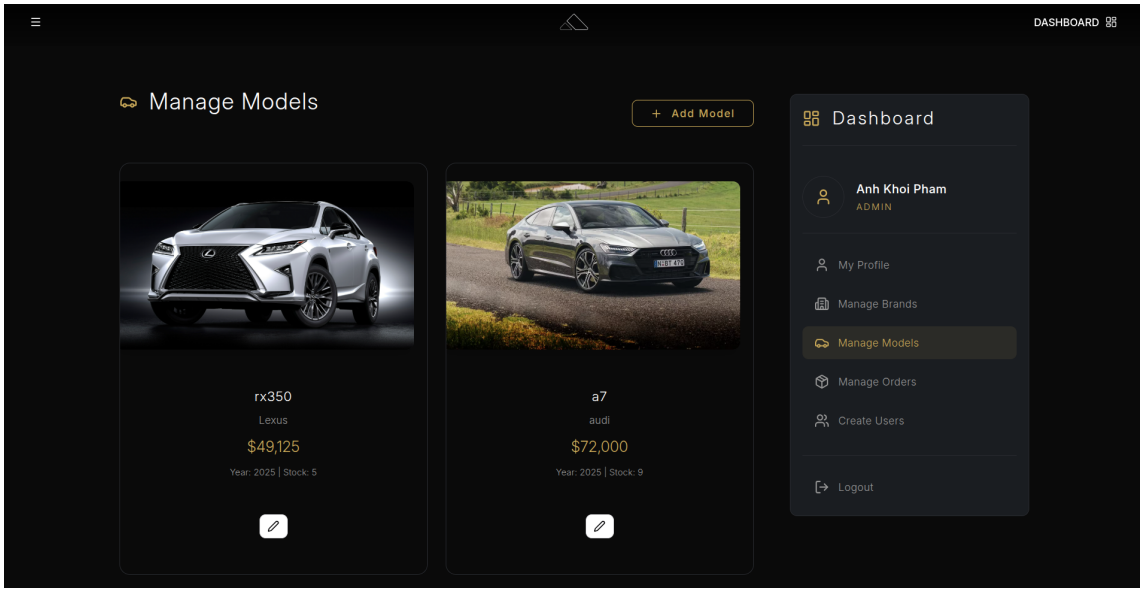

Figure 6: Model Management Interface

3.3.1 Frontend Flow

1. `AdminModels.tsx` renders the model management interface with CRUD operations
2. User interactions trigger the `useModel` hook methods
3. `model.api.ts` constructs FormData and sends multipart requests for image upload support
4. Responses update component state, triggering UI re-render

3.3.2 Backend Flow

1. `endpoints/model.py` receives the request with `require_admin` dependency
2. Request data is validated against `ModelCreate` or `ModelUpdate` schemas
3. `ModelService.create_model()` orchestrates:
   - Image upload to MinIO via `StorageService`
   - Model creation via `crud_model.create()`
   - Audit log creation for traceability
   - Transaction commit
4. Response is serialized using `ModelOut` schema

3.4 Case Study: Checkout Workflow

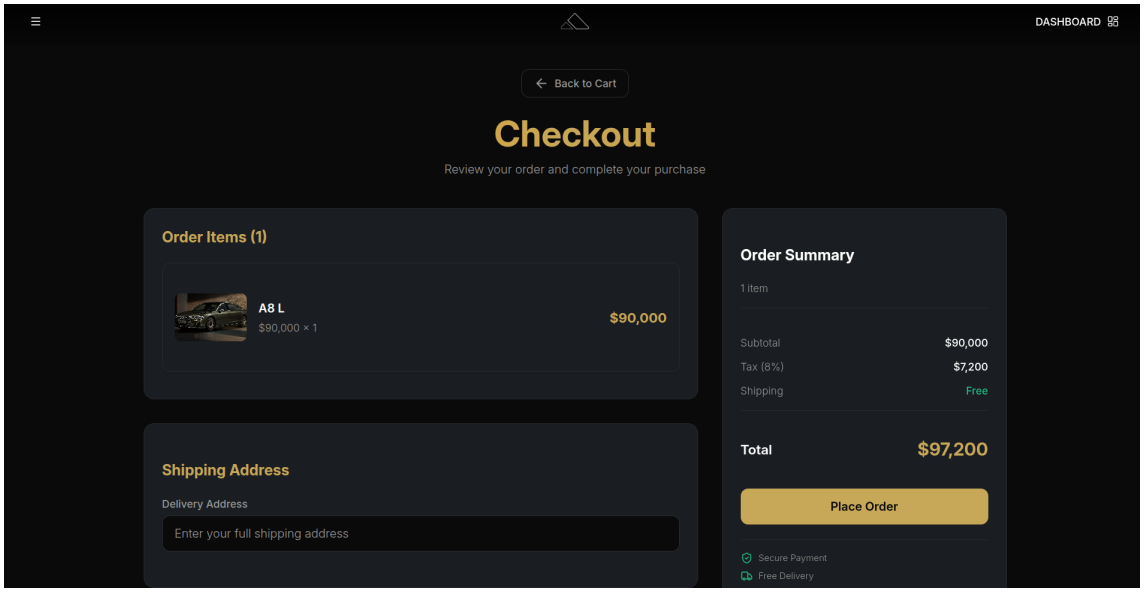

Figure 7: Checkout Page

The checkout process demonstrates multi-service orchestration:

1. `CheckoutService.place_order()` coordinates the workflow
2. `CartService.get_summary()` retrieves cart contents and calculates subtotal
3. `ShippingService.quote_shipping_fee()` calculates delivery cost
4. `OrderService.create_order()` creates order with items and decrements inventory
5. `PaymentGatewayService.charge()` processes payment (simulated)
6. Payment result triggers appropriate order status update
7. `CartService.clear_cart()` empties user cart on success

This workflow demonstrates transaction management with rollback on failure, ensuring data consistency.

4 Evaluation

4.1 Strengths

4.1.1 Clean Architecture

The layered architecture with clear boundaries between API, Service, and Repository layers promotes:

- **Maintainability**: Changes are localized to specific layers
- **Testability**: Layers can be tested in isolation with mock dependencies
- **Scalability**: Independent scaling of presentation and business logic tiers

4.1.2 Design Pattern Implementation

Consistent application of established patterns (Repository, State Machine, Strategy, Dependency Injection) provides:

- Predictable code organization
- Reduced coupling between components
- Reusable abstractions (e.g., `CRUDBase` generic class)

4.1.3 Type Safety

Full-stack type safety through TypeScript on frontend and Pydantic on backend ensures:

- Compile-time error detection
- Self-documenting API contracts
- Reduced runtime errors from type mismatches

4.1.4 Infrastructure as Code

Docker Compose configuration provides:

- Reproducible development environments
- Simplified onboarding for new developers
- Production-like local testing capabilities

4.2 Weaknesses and Recommendations

4.2.1 Error Handling Standardization

Issue: Error handling varies across services, with inconsistent exception types and messages.

Recommendation: Implement a centralized exception hierarchy with standardized error codes and response format.

4.2.2 Validation Duplication

Issue: Some validation logic exists in both frontend hooks and backend schemas.

Recommendation: Establish validation as a backend responsibility with frontend validation serving only for UX enhancement.

4.2.3 Configuration Management

Issue: Some configuration values (e.g., MinIO credentials) are hardcoded in service files.

Recommendation: Centralize all configuration in the `Settings` class with environment variable loading.

4.2.4 Testing Coverage

Issue: Limited automated testing infrastructure.
Recommendation: Implement:

- Unit tests for service layer business logic
- Integration tests for API endpoints
- Repository tests with test database fixtures

4.2.5 API Documentation

Issue: While FastAPI generates OpenAPI documentation, domain-specific documentation is limited.

Recommendation: Add comprehensive docstrings and API description metadata for improved developer experience.

5 Conclusion

The Apex Car Dealership Platform demonstrates a well-architected full-stack web application implementing established software engineering principles. The layered backend architecture, consistent design pattern application, and type-safe development approach create a maintainable and extensible codebase.

Key architectural decisions—including the Repository Pattern for data access abstraction, State Machine Pattern for order lifecycle management, and JWT-based authentication—reflect industry best practices for modern web application development.

Areas for future enhancement include comprehensive automated testing, standardized error handling, and expanded documentation. These improvements would further strengthen the system's reliability and maintainability while facilitating team scalability and knowledge transfer.

The project successfully demonstrates the practical application of software architecture principles to a real-world e-commerce domain, providing a solid foundation for continued development and feature expansion.