FACULTY OF COMPUTING
NSBM Green University

Online Auction Management System

SOFTWARE ARCHITECTURE REPORT

Course Details

Module	SE205.3 - Software Architecture
Lecturer	Mr. Diluka Wijesinghe

Group 08 Members

Name	Role	Student ID
Kodituwakku, VT	Backend Developer	27298
Rajapaksha, TAW	Database Developer	31541
Kulasuriya, DKKS	Fullstack Developer	31883
Wanigarathna, MM	Frontend Developer	31110
Katugampala, KTN	Frontend Developer	31934
Suneth, SWND	Fullstack Developer	31744
Dissanayake, DMDV	QA	33041

October 23, 2025

Contents

Abstract

This report presents the design and implementation of a high-performance, cross-platform Online Vehicle Auction System. The project's primary objective is to modernize traditional vehicle auctions, which suffer from geographical limitations, transactional opacity, and high operational overhead. Our system addresses these challenges by providing a secure, real-time, and globally accessible platform.

The system is engineered using a robust, service-oriented architecture. The backend is built on the latest ASP.NET Core framework (.NET 9), ensuring scalability and high throughput. The core real-time bidding functionality is powered by WebSockets, while a Redis distributed cache ensures low-latency performance. The frontend is a responsive Single Page Application (SPA) developed with React.js, ensuring a seamless user experience across web, mobile, and desktop platforms.

Key features include distinct portals for Sellers, who can list vehicles and set minimum bids, and Buyers, who can participate in live auctions. A significant innovation is the secure escrow-based payment gateway. This system verifies and holds buyer funds, releasing them to the seller only after the transaction is confirmed, which provides critical financial protection and builds trust for all parties.

The solution is deployed on a scalable Amazon Web Services (AWS) cloud infrastructure and is targeted at both individual enthusiasts and professional dealerships. This report details the architectural decisions and security protocols employed to deliver a production-ready application that enhances trust and efficiency in the vehicle resale market.

Keywords: Vehicle Auction, Real-time Bidding, ASP.NET Core 9, .NET 9, WebSockets, Redis, Escrow Payment, FinTech, React.js, Cross-Platform, AWS, Secure Transactions.

Technologies Used: C#, ASP.NET Core 9, PostgreSQL, Redis, React.js, WebSockets, Stripe API (with Escrow), JWT, Docker, AWS (EC2, RDS, ElastiCache).

Chapter 1

Introduction

1.1 Background and Context

The global automotive resale market represents a multi-billion dollar industry, yet it has historically been dominated by physical, localized auctions. These traditional methods, while established, are inherently inefficient. They suffer from high operational costs, geographical limitations that restrict both buyer and seller pools, and a lack of real-time price discovery.

While first-generation online marketplaces brought greater accessibility, many failed to replicate the dynamic, time-sensitive nature of a live auction. Furthermore, for high-value assets like vehicles, establishing transactional trust—ensuring vehicle authenticity and, most critically, payment security—remains the single greatest challenge in the digital space. This project addresses the clear market need for a platform that combines the real-time engagement of a live auction with robust, modern security and trust-building mechanisms.

1.2 Problem Statement

The traditional and early-digital vehicle auction models suffer from several critical flaws that this project aims to solve:

- **Lack of Trust and Transparency:** Buyers face significant risks related to vehicle condition, while sellers face equally high risks of non-payment or payment fraud after a vehicle has been sold.

- **Limited Market Access:** Physical auctions restrict both buyers and sellers to a single geographical location, artificially depressing market reach and potential value.

- **Poor Real-time Experience:** Many online platforms lack true real-time bidding, relying on 'proxy bids' or page refreshes, which fails to create a competitive, transparent, and engaging auction environment.

- **Transactional Insecurity:** The handover of a high-value asset and a large sum of money is a major point of friction, with few platforms offering a secure intermediary (escrow) service to protect both parties.

1.3 Aims and Objectives

The primary aim of this project is to design, develop, and deploy a high-performance, cross-platform Online Vehicle Auction System.

The key objectives to achieve this aim are:

1. To engineer a **real-time bidding engine** using WebSockets to provide instantaneous, sub-second bid updates to all connected users.

2. To establish a **high-trust environment** by implementing a secure, escrow-based payment gateway that holds funds until the transaction (e.g., vehicle handover) is confirmed by both parties.

3. To develop a **scalable backend architecture** using the latest ASP.NET Core 9 framework and a Redis cache, capable of handling high-frequency bidding traffic.

4. To create **distinct user portals** for Sellers (to list vehicles and set minimum prices) and Buyers (to participate in auctions) using a responsive React.js frontend.

5. To ensure the platform is **highly secure**, protecting user data, authentication, and all financial transactions.

1.4 Scope of the Project

1.4.1 In Scope

The project's scope is focused on delivering the core auction functionality:

- A complete User Management system with role-based access (Buyer, Seller, Admin).

- A Seller portal for creating, managing, and monitoring vehicle listings, including setting reserve prices.

- A Buyer portal for browsing, searching, and placing bids in real-time.

- The real-time WebSocket-based notification system for bid confirmations and auction alerts.

- The complete escrow payment workflow, from buyer payment-hold to seller fund-release.

1.4.2 Out of Scope

To ensure delivery within the project timeline, the following features are considered out of scope for the current version:

- Native mobile applications (iOS/Android). The system is, however, fully web-responsive for mobile browsers.

- AI-powered vehicle price valuation or recommendation engines.

- Live video streaming for auctions.

- Integration with third-party vehicle history report services (e.g., CarFax).

1.5 Technology and Methodology

The system is developed using a modern, service-oriented architecture. The backend is built with ASP.NET Core 9 on the .NET 9 platform, the frontend with React.js, and the primary database with PostgreSQL. Real-time communication is handled by WebSockets, and performance is accelerated using a Redis distributed cache. The entire solution is designed for cloud-native deployment on Amazon Web Services (AWS).

1.6 Report Organization

This report is structured as follows:

Chapter 2: Literature Review reviews existing auction platforms and the core technologies relevant to this project.

Chapter 3: System Architecture provides a detailed overview of the system's design, including its service-oriented architecture and cloud deployment model on AWS.

Chapter 4: Design Patterns discusses the key design patterns, such as Observer (for real-time) and Repository, that were implemented.

Chapter 5: Implementation Details showcases key code segments and explains the implementation of core features like the bidding engine and escrow payment.

Chapter 6: Testing details the quality assurance strategy, including unit, integration, and performance testing.

Chapter 7: Individual Contributions outlines the specific roles and responsibilities of each team member.

Chapter 8: Conclusion summarizes the project's achievements, limitations, and potential for future work.

Chapter 2

Literature Review

2.1 Overview

This chapter presents a critical review of existing systems and technologies relevant to the Online Vehicle Auction market. We analyze local classified platforms (e.g., `ikman.lk`) and high-frequency bidding platforms (e.g., `1xBet`) to identify a significant "market gap." The chapter concludes by justifying our chosen technology stack, which is designed to fill this gap.

2.2 Market and Technology Analysis

2.2.1 Category 1: Local Classified Platforms

Platforms like `ikman.lk` and `riyasewana.lk` dominate the Sri Lankan vehicle resale market. Our review identifies them as **digital classified ad listings**, not true auction platforms.

Strengths High user traffic and a large inventory of listings.

Weaknesses They are static listings. Negotiations happen offline, which is slow and lacks transparency. There is no competitive price discovery, and critically, **no transactional security**, leaving users exposed to fraud.

2.2.2 Category 2: High-Frequency Bidding Platforms

Platforms such as `1xBet` (sports betting) and `eBay` are masters of high-speed, concurrent, real-time event handling, primarily using `WebSockets` and distributed caching (like `Redis`).

Strengths Their architecture is built to handle thousands of simultaneous interactions at very low latency, which is a desirable model for live auctions.

Weaknesses Their business model lacks the high-trust, escrow-based payment system required for high-value, physical assets like vehicles.

2.2.3 Identifying the Research Gap

> **Identifying the Research Gap**
>
> The literature review reveals a clear gap in the market: **There is no platform that combines the high-frequency, real-time technology of a betting site with the high-trust, secure payment model of a FinTech application.**
> Our project is designed to fill this specific gap by being both technologically advanced

(real-time, fast) and commercially secure (escrow payments).

2.3 Technology Stack Justification

The technology stack was chosen to meet the demands of this identified gap, prioritizing performance, reliability, and security.

Table 2.1: Technology Stack Selection Rationale

Component	Selected Technology	Alternatives	Rationale for Selection
Backend	**ASP.NET 9**	Node.js, Spring Boot	Industry-leading performance, perfect for handling concurrent WebSocket connections.
Real-time	**WebSockets**	Long Polling, SSE	Essential for competitive, real-time bidding. Provides a persistent, instant connection.
Database	**PostgreSQL**	MySQL, MongoDB	**ACID compliance** is crucial for reliable financial transactions and secure bids.
Caching	**Redis**	In-Memory Cache	A **distributed cache** is essential for scaling, allowing servers to share live auction data.
Frontend	**React.js**	Angular, Vue.js	Its efficient Virtual DOM is ideal for handling the rapid state changes from live bids.
Payment	**Escrow (Stripe)**	Direct Payment	**Core of building trust.** Protects both buyer and seller from payment fraud.
Hosting	**AWS (Cloud)**	On-Premise, Azure	High availability and scalability (EC2, RDS) to handle sudden auction traffic spikes.

2.4 Conclusion

The review confirms that existing platforms are inadequate, lacking either real-time technology or transactional security. Our chosen stack is therefore justified, as it directly targets this gap by combining a high-performance backend (`ASP.NET 9`, `WebSockets`, `Redis`) with a high-trust business model (`PostgreSQL`, `Escrow`) to create a novel and efficient vehicle auction platform.

Chapter 3

System Architecture

3.1 Architectural Overview

This chapter presents the complete architectural design of the Online Vehicle Auction System. The system is engineered using a **Service-Oriented Architecture (SOA)** built on a robust three-tier model. This style ensures a clear **separation of concerns**, enhances **scalability**, and maintains **security**. The architecture is cloud-native, leveraging Amazon Web Services (AWS) for high availability. Figure 3.1 provides a high-level overview.

3.2 Architectural Layers

Our system is structured into three distinct logical layers.

3.2.1 Presentation Layer (Frontend)

The cross-platform UI is built with **React.js** as a Single Page Application (SPA), providing a fluid experience on **web, mobile, and desktop**. It renders UI components, handles user interactions, communicates with the backend via RESTful Application Programming Interface (API) calls, and maintains a persistent **WebSocket** connection for real-time updates.

3.2.2 Application Layer (Backend)

Built on **ASP.NET Core 9**, this layer is the system's core ("brain"). It exposes secure API endpoints, handles authentication (JSON Web Token (JWT)), executes business rules (e.g., bid validation), serves as the **WebSocket server**, and integrates with Stripe for **escrow payment** logic.

3.2.3 Data Layer (Persistence)

This layer handles data storage using two technologies:

PostgreSQL (Primary Database) Stores persistent business data (users, vehicles, payments) ensuring **ACID compliance** for transactions.

Redis (Cache) A high-speed, in-memory cache storing volatile data (current bids, auction state) to reduce database load and ensure low latency.

3.3 Database Design

The schema is relational and normalized (3NF) for integrity. Figure 3.2 illustrates the core entities (`Users`, `Vehicles`, `Auctions`, `Bids`, `Payments`) and their relationships.

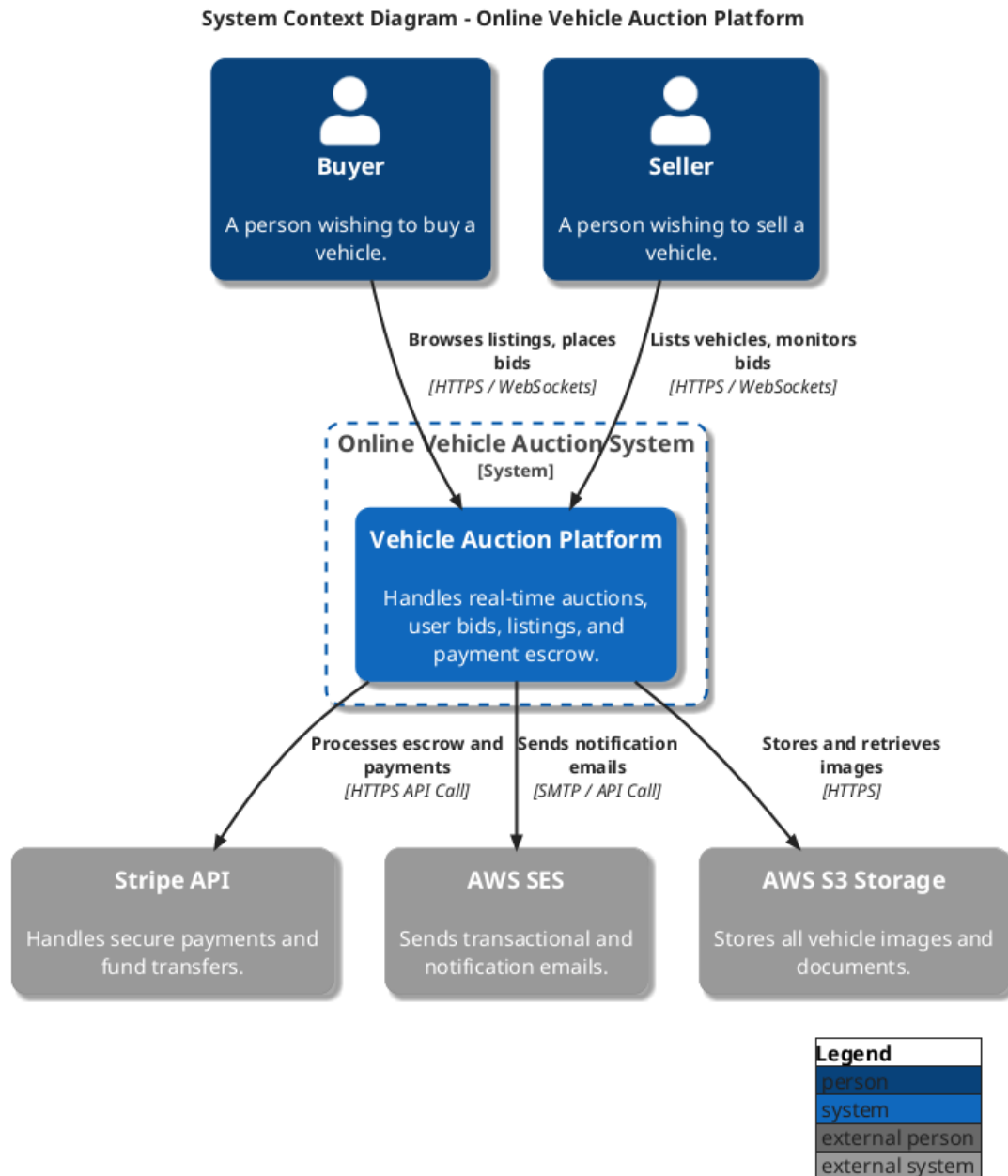**System Context Diagram - Online Vehicle Auction Platform**

Figure 3.1: High-Level System Architecture

This diagram shows the main components (Frontend, Backend API, Database, Cache) and key external systems (Stripe, Email, S3) interacting with the core Vehicle Auction System.

USERS

INT	UserId	PK	Primary Key
VARCHAR	Name		
VARCHAR	Email		Unique
VARCHAR	PasswordHash		
ENUM	Role		Buyer, Seller, Admin
TIMESTAMP	CreatedAt		

receivers

owns / lists

NOTIFICATIONS

INT	NotificationId	PK
INT	UserId	FK
TEXT	Message	
BOOLEAN	IsRead	
TIMESTAMP	CreatedAt	

VEHICLES

INT	VehicleId	PK
VARCHAR	Make	
VARCHAR	Model	
INT	Year	
TEXT	Description	
JSON	ImageList	
TIMESTAMP	CreatedAt	
INT	UserId	FK

writes

creates as Seller

is auctioned in

places as Buyer

receives reviews

makes as Buyer

REVIEWS

INT	ReviewId	PK	
INT	VehicleId	FK	
INT	UserId	FK	
INT	Rating		1-5
TEXT	Comment		
TIMESTAMP	CreatedAt		

AUCTIONS

INT	AuctionId	PK	
TIMESTAMP	StartTime		
TIMESTAMP	EndTime		
DECIMAL	StartPrice		
DECIMAL	CurrentBid		
ENUM	Status		Pending, Active, Closed, Cancelled
TIMESTAMP	CreatedAt		
INT	VehicleId	FK	Unique
INT	SellerId	FK	
INT	WinningBidId	FK	Nullable

receives bids

winning bid (nullable)

has payment

BIDS

INT	BidId	PK
DECIMAL	Amount	
TIMESTAMP	Timestamp	
INT	AuctionId	FK
INT	BidderId	FK

PAYMENTS

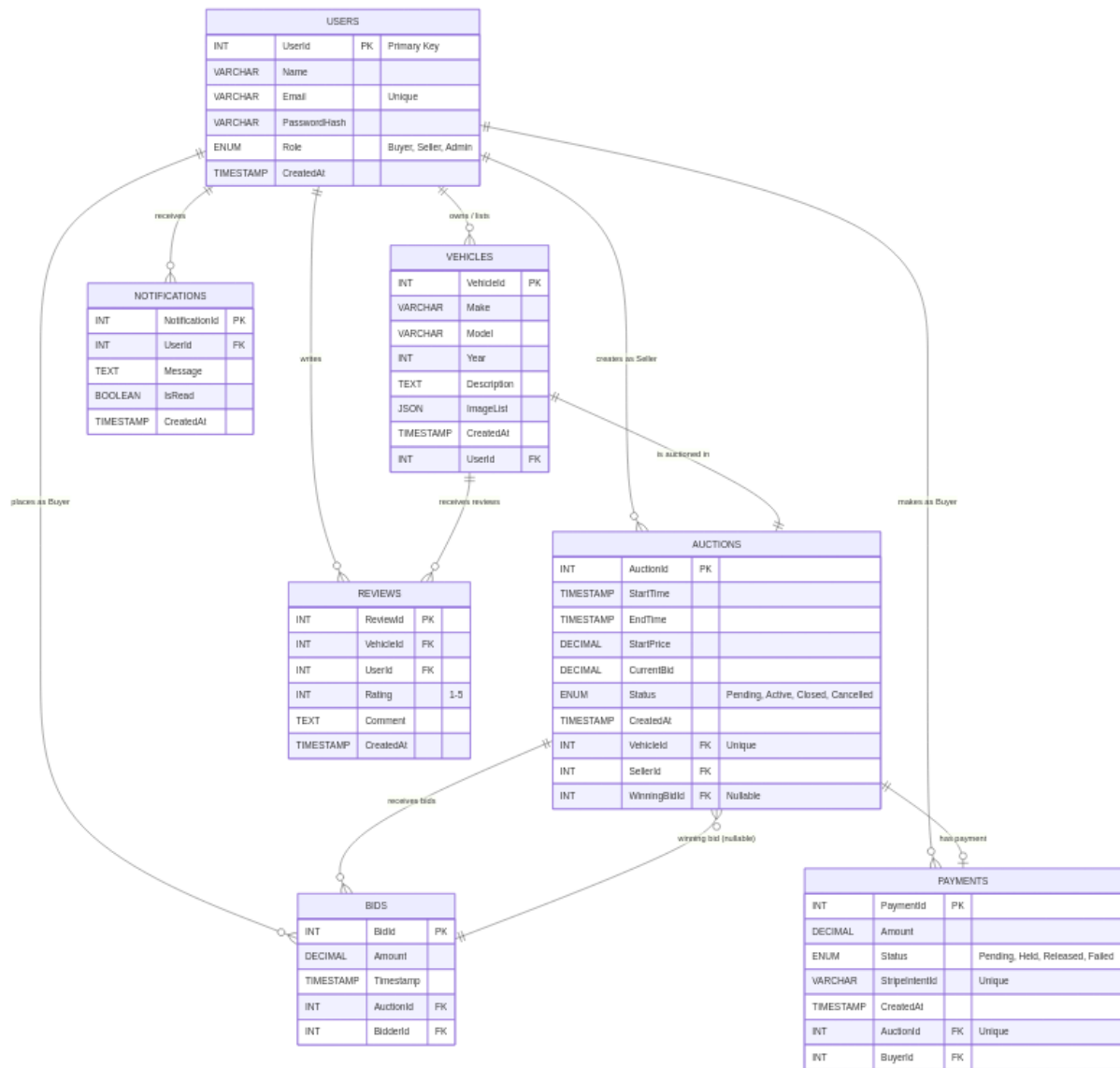| INT	PaymentId	PK	
DECIMAL	Amount		
ENUM	Status		Pending, Held, Released, Failed
VARCHAR	StripeIntentId		Unique
TIMESTAMP	CreatedAt		
INT	AuctionId	FK	Unique
INT	BuyerId	FK	

Figure 3.2: Entity-Relationship (ER) Diagram

Illustrates the main database tables and how they are linked via primary (PK) and foreign (FK) keys, showing relationships like one-to-many (e.g., one User has many Bids).

3.4 Key Process Flows (Sequence Diagrams)

Sequence diagrams illustrate component interactions for critical operations.

3.4.1 User Authentication Flow

Figure 3.3 shows the user login process and JWT issuance for secure API access.
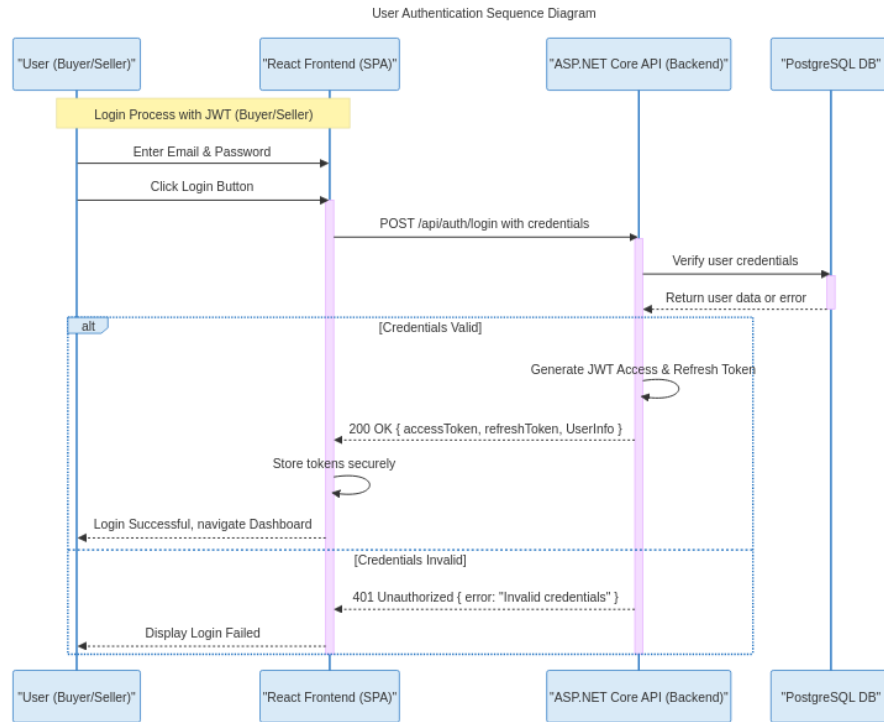

Figure 3.3: User Authentication Sequence Diagram

Details the steps involved when a user submits credentials, the API validates them against the database, and returns JWT tokens upon success.

3.4.2 Real-time Bidding Flow

Figure 3.4 depicts the real-time bid validation, saving, and broadcasting via WebSockets.

3.4.3 Escrow Payment Flow

Figure 3.5 outlines the high-trust payment model, showing fund holding and release.

3.5 Cross-Cutting Concerns

These elements apply across all architectural layers.
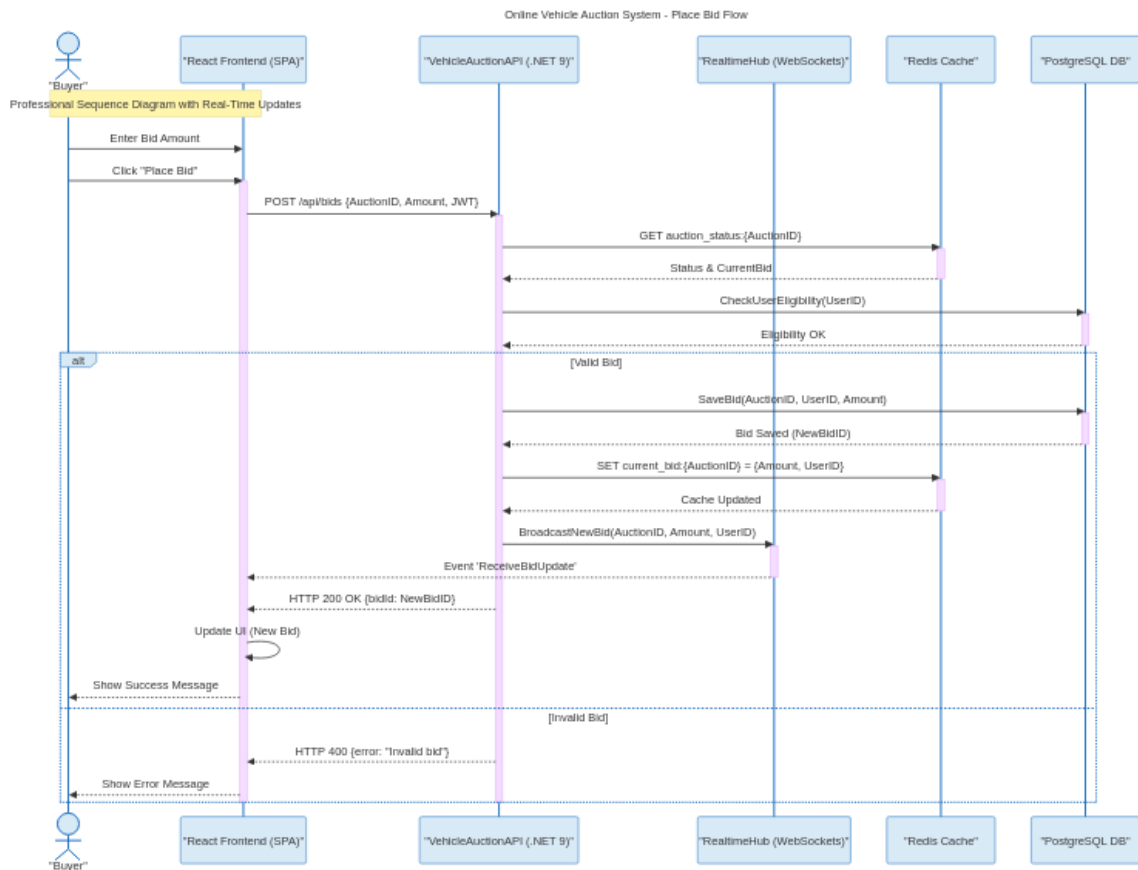
Figure 3.4: Real-time Bidding Sequence Diagram

Shows how a bid placed on the Frontend triggers API validation, database/cache updates, and finally a WebSocket broadcast to update all connected clients instantly.

Figure 3.5: Escrow Payment Sequence Diagram

Illustrates the secure process where buyer funds are held via Stripe after payment, and only released to the seller upon confirmation of the transaction (e.g., vehicle handover).

3.5.1 Security

Security is multi-layered: **Authentication** via stateless JWT tokens, **Authorization** via Role-Based Access Control (RBAC), and **Transactional Security** through the escrow system and enforced HTTPS/SSL.

3.5.2 Performance and Caching

The **Redis cache** is critical for handling high bid frequencies, serving current bid reads from memory to significantly reduce **PostgreSQL** load and ensure sub-second responses.

3.5.3 Error Handling and Logging

Global exception handling middleware in ASP.NET Core catches unhandled errors, logs them, and returns standardized error messages to the frontend, ensuring system stability.

3.6 Summary

The architecture successfully fulfills the project's core objectives. The SOA model provides separation of concerns. **WebSockets** and **Redis** deliver a high-performance real-time experience, while the **escrow payment** system and **PostgreSQL** provide the security and integrity needed for a high-trust vehicle auction platform.

Chapter 4

Design Patterns and Architectural Decisions

4.1 Overview

This chapter briefly discusses key design patterns employed and the rationale behind major architectural decisions, focusing on maintainability, scalability, and testability **gamma1994design**.

4.2 Core Architectural Pattern: Model-View-Controller (MVC)

The MVC pattern separates the application into three interconnected components: Model (data/business logic - ASP.NET Core Services/Repositories), View (UI - React.js SPA), and Controller (intermediary - ASP.NET Core API Controllers). It handles user requests, interacts with the model, and returns data to the view.
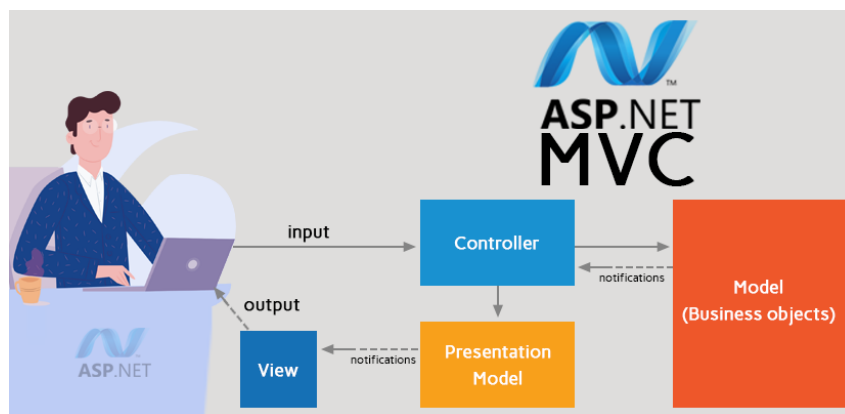

Figure 4.1: Model-View-Controller Interaction Flow

Rationale for MVC: This pattern provides a clear separation of concerns, facilitating parallel development and enhancing testability of the backend business logic independent of the frontend UI.

4.3 Key Design Patterns Implemented

4.3.1 Repository Pattern

Mediates between domain logic and data mapping, abstracting data access **fowler2002patterns**. Provides **separation of concerns** and improves **testability**.

4.3.2 Dependency Injection Pattern

Implements Inversion of Control (IoC) using ASP.NET Core's built-in container. Manages object lifetimes (**Scoped**, **Singleton**) and promotes **loose coupling**.

4.3.3 Unit of Work Pattern

Manages atomic database transactions across multiple repositories **fowler2002patterns**. Ensures **data consistency** during complex operations (e.g., bid placement).

4.3.4 Observer Pattern (via WebSockets)

Defines a one-to-many dependency for real-time updates **gamma1994design**. Implemented using **WebSockets** to broadcast state changes (e.g., new bids) instantly.

4.3.5 Strategy Pattern

Defines interchangeable algorithms, applied here for payment processing (`IPaymentStrategy`). Allows easy addition of **multiple payment providers**.

4.3.6 Singleton Pattern

Ensures a single instance for application-wide services like **configuration management**.

4.4 Key Architectural Decisions

Monolithic vs Microservices: Chose **Monolithic** for faster development and simpler deployment suitable for project scope.

ORM/Data Access: Selected **Dapper** for performance and SQL control (*Adjust if EF Core was used*).

Frontend Approach: Used **Single Page Application (React)** for superior UX and real-time updates.

API Style: Implemented **RESTful API** for simplicity, standardization, and tooling.

Authentication Method: Employed **JWT** for stateless, scalable, and cross-platform authentication.

Database Choice: Utilized **PostgreSQL** for Atomicity, Consistency, Isolation, Durability (ACID) compliance, performance, and robust features.

Caching Strategy: Implemented **Redis** (distributed cache) for handling high-frequency reads of live auction data.

4.5   Design Pattern and Architecture Summary

> **Architecture & Pattern Summary**
>
> The system's architecture, based on **MVC**, separates concerns effectively. Key patterns like **Repository**, **DI**, and **Observer (WebSockets)** enhance modularity and testability. Decisions favoring **REST**, **SPA**, **PostgreSQL**, **Redis**, and **JWT** prioritize performance, security, and real-time capabilities within the project scope.

Chapter 5

Implementation Details

5.1 Overview and Development Environment

This chapter details the practical realization of the Online Vehicle Auction System's architecture. It outlines the tools, technologies, and key code implementations that brought the design to life. The development process emphasized a modern, cross-platform workflow using industry-standard tools.

Our development environment leveraged flexibility and powerful tooling:

- **Operating Systems:** ⊞ Windows, 🐧 Linux (Arch Linux used by some), 🍎 macOS.

- **Editors/IDEs:** </> Visual Studio Code, Neovim (often paired with >_ Tmux for terminal management).

- **Version Control:** git Git was used universally. ⚙ GitHub hosted the backend (`VelocityAPI`) repository, while 🦊 GitLab managed the frontend repository and its CI/CD pipeline.

- **Containerization:** 🐳 Docker facilitated consistent development and deployment environments (`Dockerfile`, `docker-compose.yml`).

- **API Testing:** 🚀 Postman was extensively used for manual and automated API endpoint testing.

5.2 Backend Implementation (`VelocityAPI`)

The backend was developed using **ASP.NET Core 9**, focusing on performance, security, and real-time capabilities.

5.2.1 Core Components

Authentication (🔑 JWT) A custom token service (`Application/Authentication/Services`) generates and validates stateless JWT access, refresh, and potentially session tokens. Secure password hashing uses BCrypt.Net-Next. An `AuthorizationFilter` protects endpoints.

Real-time Bidding (WebSockets) ASP.NET Core's built-in WebSocket support (or SignalR) powers the real-time bidding hub. It manages client connections per auction and broadcasts updates efficiently.

Escrow Payment (stripe Stripe) Integration with the Stripe API handles the secure escrow payment flow, including creating payment intents, holding funds, and releasing them upon confirmation.

Caching (Redis) Distributed caching with Redis (configured via `Configuration/RedisConfig.cs`) stores active auction states (current bids, timers) and user session data for low-latency reads.

Data Access (PostgreSQL) Dapper was used for high-performance, raw Structured Query Language (SQL) queries against the PostgreSQL database, providing fine-grained control for critical operations. Schema is defined in `schema/schema.sql`.

Image Storage (S3) The `Application/S3/S3.cs` service handles uploading and retrieving vehicle images securely to/from an AWS S3 bucket.

Error Handling A custom `ExceptionHandlerMiddleware` provides global error handling, logging exceptions and returning standardized error responses.

```
1  // Located within a Realtime Hub or Service
2  public async Task BroadcastBidUpdate(string auctionId, Bid newBid
      )
3  {
4      // Simplified: Get bidder name, format message
5      var message = new {
6          AuctionId = auctionId,
7          Amount = newBid.Amount,
8          Bidder = newBid.Bidder.Name // Assuming navigation
              property
9      };
10
11     // Send to all clients connected to this auction's group
12     await _hubContext.Clients.Group($"auction_{auctionId}")
13                           .SendAsync("ReceiveBidUpdate",
                                  message);
14     _logger.LogInformation("Broadcasted bid for Auction {
          AuctionId}", auctionId);
15 }
```

Listing 5.1: Example: Simplified WebSocket Broadcast Snippet

5.3 Frontend Implementation (React SPA)

The ⚛ **React.js** frontend provides the user interface for buyers and sellers across different platforms.

- **UI Components:** Developed using functional components and hooks, creating reusable elements for auction lists, details, bidding forms, and user dashboards.

- **Real-time Connection:** A dedicated service manages the WebSocket connection, subscribing to auction-specific updates and updating the application state when messages like `ReceiveBidUpdate` are received.

- **State Management:** Utilized React Context API or a library like Zustand for managing global state (e.g., user authentication, live auction data).

- **API Interaction:** Employed `axios` or `fetch` for communicating with the `VelocityAPI` backend, handling request/response cycles and JWT token management.

- **CI/CD (🦊):** The GitLab repository was configured with CI/CD pipelines for automated testing and deployment of the frontend application.

5.4 Database Schema and Cache Configuration

5.4.1 PostgreSQL Schema (🗄)

The database schema, defined in `schema/schema.sql`, follows relational principles (3NF). Key tables include `Users`, `Cars`, `Auctions`, `Bids`, and `Payments`. Triggers (`triggers.sql`) might be used for automated actions.

Sample PostgreSQL Table Definition (`schema.sql`)

```sql
CREATE TABLE Auctions (
    AuctionId UUID PRIMARY KEY DEFAULT gen_random_uuid(),
    CarId UUID NOT NULL REFERENCES Cars(CarId),
    SellerId UUID NOT NULL REFERENCES Users(UserId),
    StartTime TIMESTAMPTZ NOT NULL,
    EndTime TIMESTAMPTZ NOT NULL,
    StartPrice DECIMAL(12, 2) NOT NULL,
    CurrentBid DECIMAL(12, 2) NOT NULL,
    Status VARCHAR(10) NOT NULL
        CHECK (Status IN ('Pending', 'Active', 'Closed', '
            Cancelled')),
    CreatedAt TIMESTAMPTZ DEFAULT CURRENT_TIMESTAMP
);
CREATE INDEX idx_auctions_status_end_time ON Auctions (Status
    , EndTime);
```

5.4.2   Redis Configuration ()

Redis connection details are managed via `appsettings.json` and loaded into `Configuration/RedisConfig.cs`. Health checks (`RedisHealthCheck.cs`) monitor connectivity.

Sample Redis Configuration (`appsettings.json`)

```
{
  "Redis": {
    "ConnectionString": "localhost:6379", // Or AWS
        ElastiCache endpoint
    "InstanceName": "VelocityCache_"
  }
}
```

5.5 Testing Strategy (✎)

A multi-layered testing approach was adopted to ensure code quality and system reliability:

- **Backend Unit Tests:** Using **xUnit/NUnit** (✎) to test individual services and logic components in isolation, employing mocking frameworks like Moq.

- **Frontend Unit/Component Tests:** Using **Jest** and **React Testing Library** (⚛) to verify individual React components and hooks.

- **Integration Tests:** Leveraging **ASP.NET Core's integration testing** framework to test the interaction between different backend components, including database access.

- **End-to-End (E2E) Tests:** Potentially using frameworks like **Cypress** or **Playwright** (🎭) to simulate real user flows through the entire application (Frontend + Backend).

- **Load Testing:** Using tools like **JMeter** or **k6** () to simulate high user load and identify performance bottlenecks.

- **API Testing:** Continuous testing of API endpoints using **Postman** (🚀) collections.

This comprehensive strategy aimed to catch bugs early and validate non-functional requirements like performance and security.

5.6 Deployment ()

The application is designed for cloud deployment on aws **AWS**.

- **Containerization (🐳):** The backend API is containerized using the `Dockerfile`, enabling consistent deployment across environments. `docker-compose.yml` orchestrates local development services (API, DB, Redis).

- **Cloud Services:** The production environment utilizes AWS services like Elastic Compute Cloud (EC2) (or Fargate/EKS) for hosting the containers, Relational Database Service (RDS) for PostgreSQL, ElastiCache for Redis, and S3 for image storage. A Load Balancer distributes incoming traffic.

This approach ensures scalability, reliability, and leverages managed cloud services for easier operations.

Chapter 6

Testing and Quality Assurance

6.1 Testing Strategy Overview (☑)

A multi-layered testing strategy ensured the quality and reliability of the Online Vehicle Auction System, following the testing pyramid principle. Emphasis was placed on unit tests, complemented by integration, API, and potentially end-to-end tests, aiming to catch defects early. Quality assurance was integrated throughout the Agile development lifecycle.

6.2 Unit Testing (🔬)

Unit tests verified individual components in isolation.

6.2.1 Backend Unit Tests (🖴)

Backend services and logic were tested using **xUnit/NUnit** and **Moq**. Tests covered business logic (bid validation), authentication/authorization, data mapping, and error handling, targeting high coverage for critical services.

6.2.2 Frontend Unit Tests (⚛)

Components and hooks were tested using **Jest** and **React Testing Library**, focusing on rendering, behavior based on props/state, and user interaction simulation.

6.3 Integration Testing (🖧)

Integration tests validated the interaction between backend components using ASP.NET Core's testing framework. Key scenarios tested included API controller flows interacting with services, database (PostgreSQL via Dapper), cache (Redis), and inter-service communication (e.g., triggering WebSocket broadcasts). This ensured components worked correctly together.

6.4 API Testing (🚀)

Manual and automated API testing was performed using **Postman**. A comprehensive collection covered all endpoints (Appendix A), validating successful responses (2xx), error handling (4xx/5xx), authentication rules, and data formats. Automated scripts aided regression testing.

6.5 Performance and Load Testing ()

Performance testing with **JMeter/k6** simulated concurrent users to ensure the system handled expected load, especially during auctions. Key metrics like response time (bid placement), throughput, and error rates were monitored. The positive impact of the Redis cache on read performance was specifically verified.

6.6 Security Testing ()

Basic security checks aimed to identify common vulnerabilities:

- Validated input sanitization against SQL Injection and XSS.

- Confirmed secure JWT handling (validation, expiration).

- Verified Role-Based Access Control (RBAC) enforcement.

- Checked for secure handling of sensitive data and enforced HTTPS.

6.7 Bug Tracking and Resolution (🐞)

Defects found were tracked (e.g., via GitHub Issues), prioritized by severity, and assigned for resolution. Regression testing followed fixes to prevent new issues.

6.8 Testing Summary (✅)

The rigorous testing strategy provided high confidence in the system's quality.

Testing Results Summary

- **Unit Tests:** Achieved target code coverage (>90

- **Integration Tests:** Confirmed successful interaction between backend components.

- **API Tests:** All defined Postman tests passed.

- **Performance:** Met response time goals under simulated load.

- **Security:** Basic vulnerability checks passed.

- **Bugs:** All critical/high-severity bugs resolved prior to release.

Testing confirmed the delivery of a reliable, performant, and secure Online Vehicle Auction System.

Chapter 7

Individual Contributions

7.1 Contribution: Suneth, SWND

Student ID: 31744   **Role:** Fullstack Developer (System Design Focus)

7.1.1 System Design and High-Level Architecture (⌘)

My primary responsibility involved translating the initial project requirements into a viable technical blueprint. This encompassed defining the overall **Service-Oriented Architecture (SOA)**, layered upon a robust three-tier model, as depicted in Fig. 7.1. A key focus was establishing clear boundaries and interfaces between the frontend (SPA), backend (API), and data persistence layers (⬚,), ensuring loose coupling and designing for scalability under potential high-load scenarios typical of auctions.

I contributed significantly to the creation of initial sequence diagrams (⌘) for critical user flows such as the real-time bidding process and escrow payment sequence. These diagrams helped clarify component responsibilities and interaction patterns early in development.

7.1.2 Backend Development and Data Layer Implementation (⬚⬚)

As a fullstack developer, I implemented foundational backend features and contributed to the data layer using **ASP.NET Core 9**.

[leftmargin=1.1cm, itemsep=1pt]

- **Core Auction Logic:** Implemented API endpoints in `AuctionController` including Create, Read, Update, Delete (CRUD) operations and status transitions with business validations.

- **Database Schema:** Contributed to the `Auctions` table structure in PostgreSQL ensuring normalized design (3NF).

- **Redis Caching Strategy (⚡):** Designed cache for critical volatile data (current bid, status, timer) and implemented update logic for consistency.
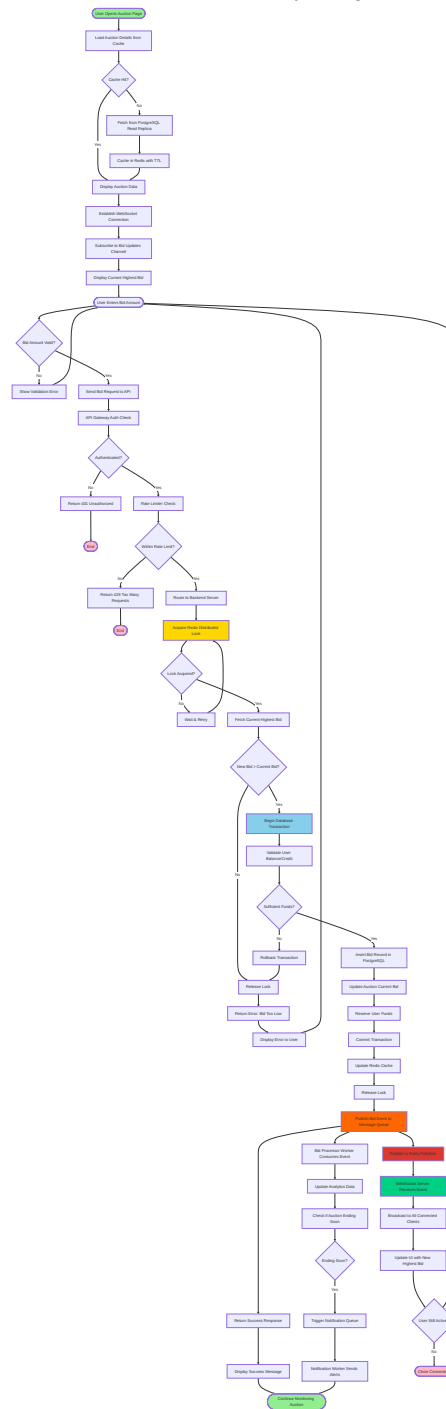
Figure 7.1: High-Level System Architecture Overview.

7.1.3 Core Domain Model Design (Class Diagram - 🕸)

A clear domain model was necessary for implementation. I created and refined the primary **Class Diagram**, a mandatory deliverable shown below.

Figure 7.1.3: Core Domain Class Diagram

entities, relationships, and service interfaces forming backend structure. *Key*

Technologies and Concepts Studied & Applied

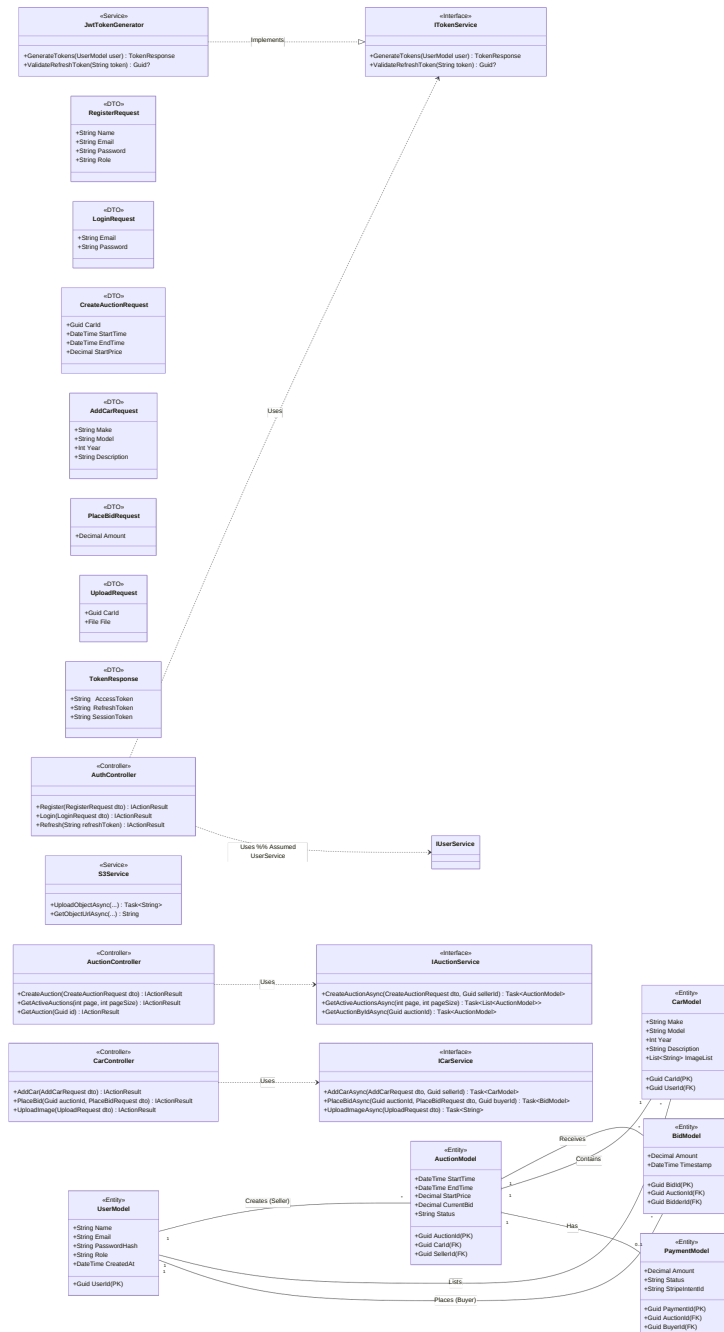| Area	What I Learned / Used in the Project
Software Architecture	SOA, 3-Tier Design, Component Responsibilities
Backend Development	ASP.NET Core 9, C#, RESTful API Design, Controllers & Services
Databases	PostgreSQL, Schema Design, Indexing, 3NF Normalization
Caching & Performance	Redis Cache Keys, Consistency Control, High-Read Optimization
Modeling Techniques	UML Class & Sequence Diagrams via PlantUML
Dev Tools	VS Code, Neovim, Dapper ORM, Git Collaboration

Summary of Suneth SWND's Key Contributions

Area	Tasks / Technologies
Modeling	Class Diagram (Mandatory), Refinement
Architecture	SOA/3-Tier Design, Components, Sequence Diagrams
Backend Dev	ASP.NET Core 9 API, Auction Services
DB Work	PostgreSQL Schema (`Auctions`)
Caching	Redis Design + Update Logic
Tools Used	PlantUML, VS Code/Neovim, .NET 9, Dapper, Redis

7.2 Contribution: Kodituwakku, VT

Student ID: 27298   **Role:** Backend Developer (Authentication & Security Focus)

7.2.1 User Authentication Implementation (🔑)

My core responsibility was the design and implementation of the complete user authentication system for the `VelocityAPI` backend. This involved creating a secure and robust mechanism for user registration, login, and session management using JWT.

Key implementation tasks included:

- **Token Service (`ITokenService`):** Developed the primary service (`JwtTokenGenerator.cs`) responsible for generating JWT Access Tokens (short-lived), Refresh Tokens (long-lived, for re-authentication), and potentially Session Tokens (for Redis-based session tracking). Configured token parameters (issuer, audience, expiry, signing key) securely via application settings.

- **Password Security (🔒):** Integrated **BCrypt.Net-Next** library for securely hashing user passwords during registration and verifying them during login, protecting against rainbow table attacks.

- **Registration Logic:** Implemented the backend logic for the `/auth/register` endpoint, including validating user input (email format, password strength), checking for existing users, hashing the password, and saving the new user record to the PostgreSQL database.

- **Login Logic:** Implemented the backend logic for the `/auth/login` endpoint, which verifies credentials against the database and, upon success, generates and returns the necessary tokens, as illustrated in the sequence diagram (Fig. 7.2).

The authentication flow depicted above begins with the user submitting credentials via the React frontend. The backend API receives these, verifies the password hash against the stored hash in the PostgreSQL database, and upon successful verification, generates a set of JWT tokens (Access, Refresh, Session). These tokens are returned to the frontend, which then stores them securely to authenticate subsequent user requests.

Authentication Verification Redis Usage	
Aspect	**Implementation Detail**
Credential Verification	Backend compares provided password (after hashing with BCrypt) against the stored `PasswordHash` in the `Users` table (PostgreSQL).
Token Integrity	JWT signature is verified on every request using the securely stored secret key.
Session Validity ()	Active session identifiers (Session Tokens) are stored in Redis with an expiry matching the Refresh Token. Requests can optionally check Redis to confirm the session wasn't explicitly revoked (e.g., via logout).
Refresh Token ()	Valid Refresh Tokens (or their metadata) are potentially stored in Redis, allowing for secure validation and revocation capabilities.

Figure 7.2: User Authentication Flow Sequence Diagram.

Illustrates the interaction between User, Frontend, API, and Database during the login process, culminating in JWT issuance.

7.2.2 Authorization and Access Control (👤⚙)

Beyond authentication, I implemented the authorization mechanism:

- **Role-Based Access Control (Role-Based Access Control (RBAC)):** Configured ASP.NET Core authorization based on user roles ("Buyer", "Seller", "Admin").

- **Authorization Filter (`AuthorizationFilter.cs`):** Used custom filters or built-in attributes (`[Authorize(Roles = "...")]`) to protect specific API endpoints (e.g., only Sellers can call `POST /cars`).

- **Token Validation Middleware:** Configured the JWT Bearer middleware to validate tokens based on issuer, audience, lifetime, and signing key.

7.2.3 Security Best Practices (🔒)

Implemented security best practices including:

- Strong, salted password hashing (BCrypt).

- Short lifespans for Access Tokens (e.g., 15 mins) and longer for Refresh Tokens (e.g., 7 days).

- Secure storage considerations for Refresh Tokens (HttpOnly cookies or secure backend storage).

- Rigorous input validation.

- Enforced HTTP Secure (HTTPS) communication.

7.2.4 Redis Integration for Session/Token Management ()

Redis was integrated to enhance security and scalability:

- **Session Validation:** Active session identifiers (Session Tokens) stored in Redis allow quick server-side checks for validity and easier session revocation.

- **Refresh Token Storage/Validation:** Storing Refresh Token metadata in Redis enables easier revocation and rotation, enhancing security.

This provides server-side session control complementing JWT's stateless nature.
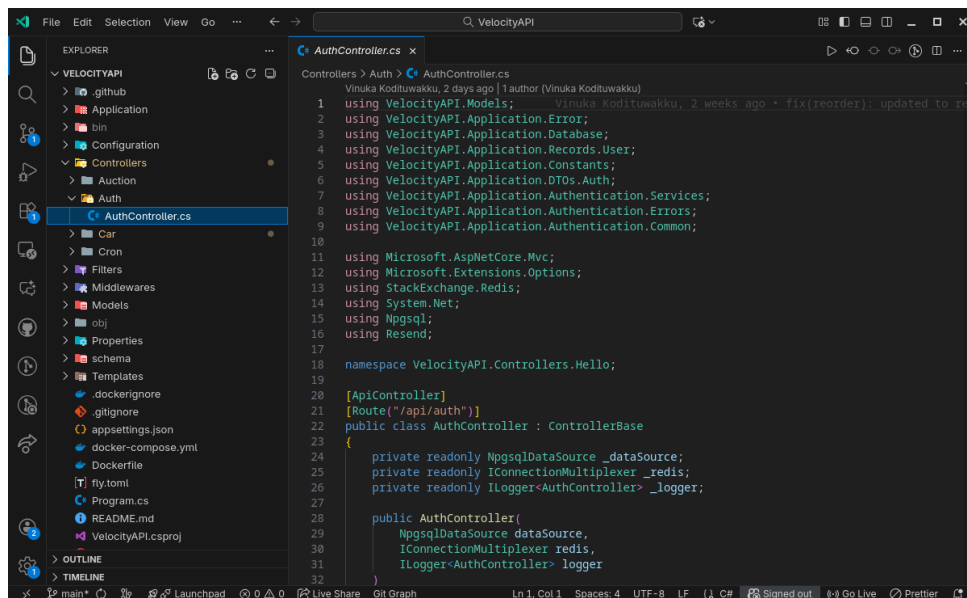

Figure 7.3: Backend Folder Structure Highlighting Authentication Components.
Organization of authentication code within `Application/Authentication`.

7.2.5 Testing and Verification (🖊)

Contributed to testing authentication/authorization:

- Wrote unit tests (xUnit/Moq) for the `JwtTokenGenerator` service.

- Created Postman requests to test registration, login, and protected endpoints, verifying token handling and role enforcement.

Summary of Kodituwakku VT's Key Contributions	
Area	**Specific Tasks / Technologies**
🔑 Authentication	JWT Token Service (Access/Refresh/Session), Login/Register Logic, BCrypt Password Hashing
👥 Authorization	Role-Based Access Control (RBAC), Authorization Filters/Attributes, JWT Middleware Config
Caching/Session	Redis integration for session validation / token management
🖊 Testing	Unit tests (xUnit/Moq), API tests (Postman) for Auth endpoints
🔒 Security	Token expiry, secure storage concepts, input validation relating to auth
Technologies	ASP.NET Core 9, C#, BCrypt.Net-Next, JWT Libraries, Stack-Exchange.Redis

7.3 Contribution: Katugampala, KTN

Student ID: 31934   **Role:** Frontend Developer

7.3.1 Frontend Component Development

My primary responsibility involved translating UI/UX designs into functional and reusable React.js components, establishing the core visual structure and interactivity of the user-facing application. Emphasis was placed on creating a modular and maintainable component architecture.

Core Components Implemented

- **Navigation System:** Implemented the primary `Navbar.jsx` component, incorporating responsive design elements and integrating client-side routing using React Router DOM for seamless page transitions across Home, Auctions, and Sell Vehicle pages.

- **Hero Section:** Developed the dynamic `Hero.jsx` component for the landing page, featuring background visuals and the main call-to-action button, as demonstrated in Figure 7.4.

- **Vehicle Cards:** Created the modular `CarCard.jsx` component for displaying individual vehicle summaries including images, titles, and descriptions fetched from the API. This component is reused across auction listings and search results (Figure 7.5).

- **Layout Components:** Established foundational layout elements including `Footer.jsx` and container components to enforce consistent page structure and styling baseline.

Styling was achieved using CSS Modules (`*.module.css`) to ensure component-level scope and prevent style conflicts, promoting a clean and maintainable CSS architecture.
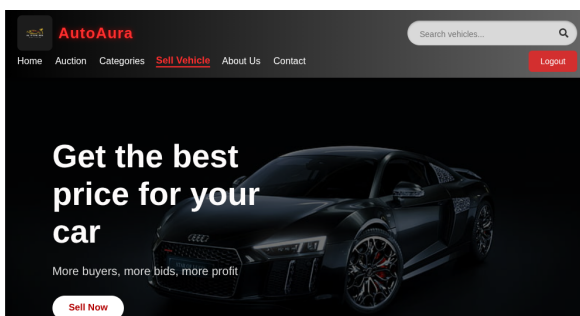
Figure 7.4: Homepage Hero Section and Navigation
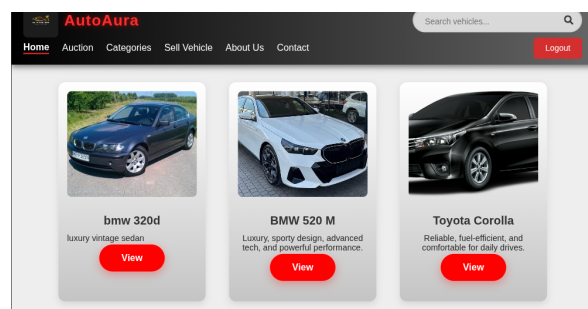
Figure 7.5: Vehicle Listings with CarCard Components

7.3.2 State Management and API Integration

Effective state management was crucial for handling user authentication status, auction data, and form inputs dynamically. The implementation utilized React Context API combined with custom hooks (`useAuth`, `useAuctions`) to manage global application state. Integrated Axios as the primary HTTP client for interacting with the backend VelocityAPI,

including services for fetching auction listings, handling user authentication, JWT token management, and comprehensive error handling mechanisms.

Library/Tool	Purpose
React.js	Core UI library for building component-based interfaces
React Router DOM	Client-side navigation and routing management
Axios	HTTP client library for robust API communication
Context API + Hooks	Global state management solution
CSS Modules	Component-scoped styling
Visual Studio Code	Primary integrated development environment

Table 7.1: Key Frontend Libraries and Development Tools

7.3.3 Code Implementation and Best Practices

The development process followed React best practices for writing clean, functional components. Figure 7.6 illustrates the development environment setup, while Listing 7.1 demonstrates the implementation approach.
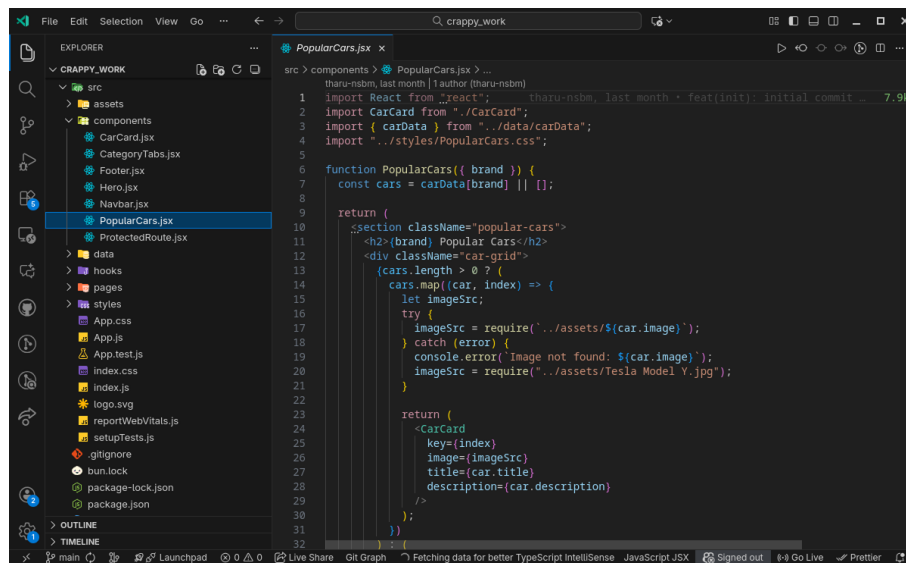

Figure 7.6: React Component Structure in Visual Studio Code

```
import React from 'react';
import CarCard from './CarCard';
import carData from '../data/carData';
import '../styles/PopularCars.css';

function PopularCars({ brand }) {
  const cars = carData[brand] || [];

  const loadImage = (imageName) => {
    try {
```

```
        return require('../assets/images/${imageName}');
    } catch (error) {
        console.error('Image not found: ${imageName}');
        return require('../assets/images/default.jpg');
    }
  };

  return (
    <section className="popular-cars">
      <h2>Popular {brand} Cars</h2>
      <div className="car-grid">
        {cars.length > 0 ? (
          cars.map((car, index) => (
            <CarCard
              key={car.id || index}
              image={loadImage(car.image)}
              title={car.title}
              description={car.description}
            />
          ))
        ) : (
          <p>No cars currently listed for this brand.</p>
        )}
      </div>
    </section>
  );
}

export default PopularCars;
```

Listing 7.1: Sample React Component (PopularCars.jsx)

The implementation demonstrates key React patterns including component composition, dynamic imports with error handling, conditional rendering, and proper key management. This modular architecture provides a solid foundation for future feature additions and maintenance.

7.4 Contribution: Wanigarathna, MM

Student ID: 31110   **Role:** Frontend Developer

7.4.1 Frontend User Interface (UI) Implementation

My core responsibility was translating the provided UI/UX designs into a functional, responsive, and aesthetically pleasing React.js application. This involved meticulous implementation of layout, styling, and interactivity for various sections of the auction platform.

Key implementation aspects included:

- **Design Translation:** Accurately converting visual mockups into HTML/CSS structures within React components, ensuring fidelity to the intended design language.

- **Styling:** Employing CSS Modules or Tailwind CSS for component-scoped styling, focusing on creating a consistent theme (colors, fonts, spacing) across the application.

- **Responsiveness:** Utilizing modern CSS techniques (Flexbox, Grid, media queries) to ensure the application layout adapts seamlessly to various screen sizes, from mobile devices to desktops.

- **React Hooks for UI State:** Leveraging core React Hooks like `useState` extensively to manage local component state, such as form inputs, modal visibility, and UI toggles.
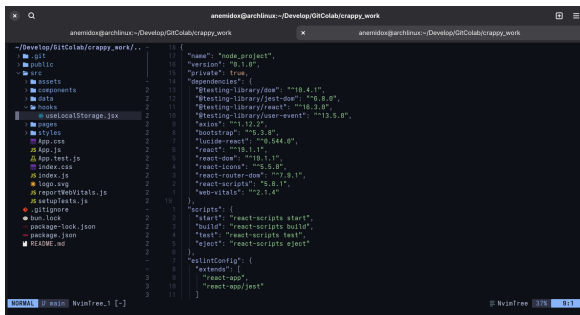
Figure 7.7: Development Environment in NEOVIM
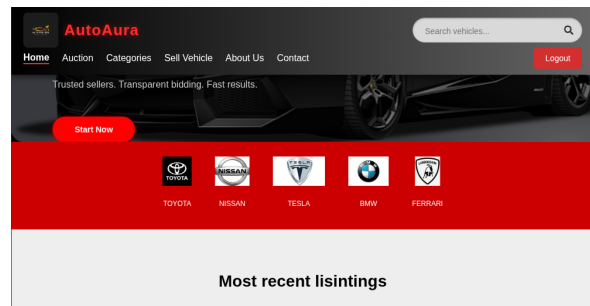
Figure 7.8: Auction Listing Display

7.4.2 API Integration and Data Fetching

Connecting the frontend to the backend API was crucial for displaying dynamic data and enabling user actions. My contributions involved implementing data fetching logic within components using the `useEffect` hook combined with Axios, managing loading and error states during API calls, integrating authentication context for JWT Bearer token inclusion in protected requests, and connecting frontend UI elements for payment processing to the backend Stripe escrow payment endpoints.

Figure 7.9: Car Card Display Implementation

Figure 7.10: Payment Gateway Integration

7.4.3 Real-time UI Updates

Ensuring the UI reflected live auction changes instantly was a key part of my work. I collaborated on integrating the WebSocket client service into relevant components, particularly the auction detail and bidding interfaces. Utilized hooks (`useState`, `useEffect`, or context updates) to dynamically update the UI based on messages received from the We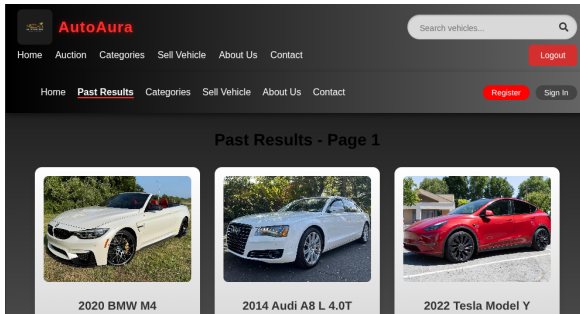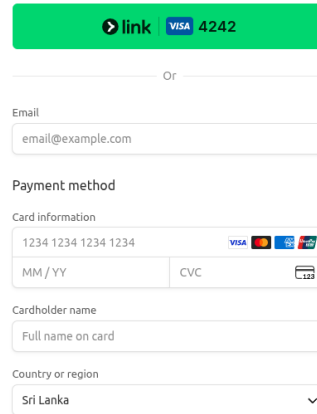bSocket server, including current bid updates, bidder information, and auction timers without requiring manual page refreshes. Implemented proper subscription management within components using `useEffect` to subscribe to relevant WebSocket events upon mounting and unsubscribe upon unmounting, preventing memory leaks and unnecessary updates.

Technology	Implementation Purpose
React Hooks	State management and lifecycle handling
Axios	HTTP client for API communication
WebSocket Client	Real-time bidding updates and notifications
CSS Modules/Tailwind	Component-scoped responsive styling
JWT Authentication	Secure API request authorization
Stripe Integration	Payment gateway frontend implementation

Table 7.2: Key Technologies and Implementation Areas

7.4.4 Summary

The frontend development contribution focused on creating a responsive, interactive user interface with seamless API integration and real-time updates. The implementation emphasized clean code practices, efficient state management, and user experience optimization through dynamic data fetching and WebSocket integration for live auction functionality.

7.5 Contribution: Rajapaksha, TAW

Student ID: 31541   **Role:** Database Developer

7.5.1 Database Schema Design and Implementation

My primary responsibility encompassed the design, implementation, and optimization of the PostgreSQL database architecture for the Velocity Auction System. The database schema supports complex auction workflows, real-time bidding, and transaction management while ensuring data integrity and optimal performance.

The implementation consists of nine interconnected tables within the `velocity` schema: `_user` (with email verification and strike-based reputation system), `_vehicle` (comprehensive specifications and technical details), `_auctions` (status-based workflow with unique active-auction-per-vehicle constraint), `_bids` (payment proof and indexed amounts), `_transactions` (payment status tracking), `_bank_details` (secure payment information), `_brand` and `_vehicle_type` (normalized categorization), and `_ownership_transfers` (audit trail for vehicle ownership changes).
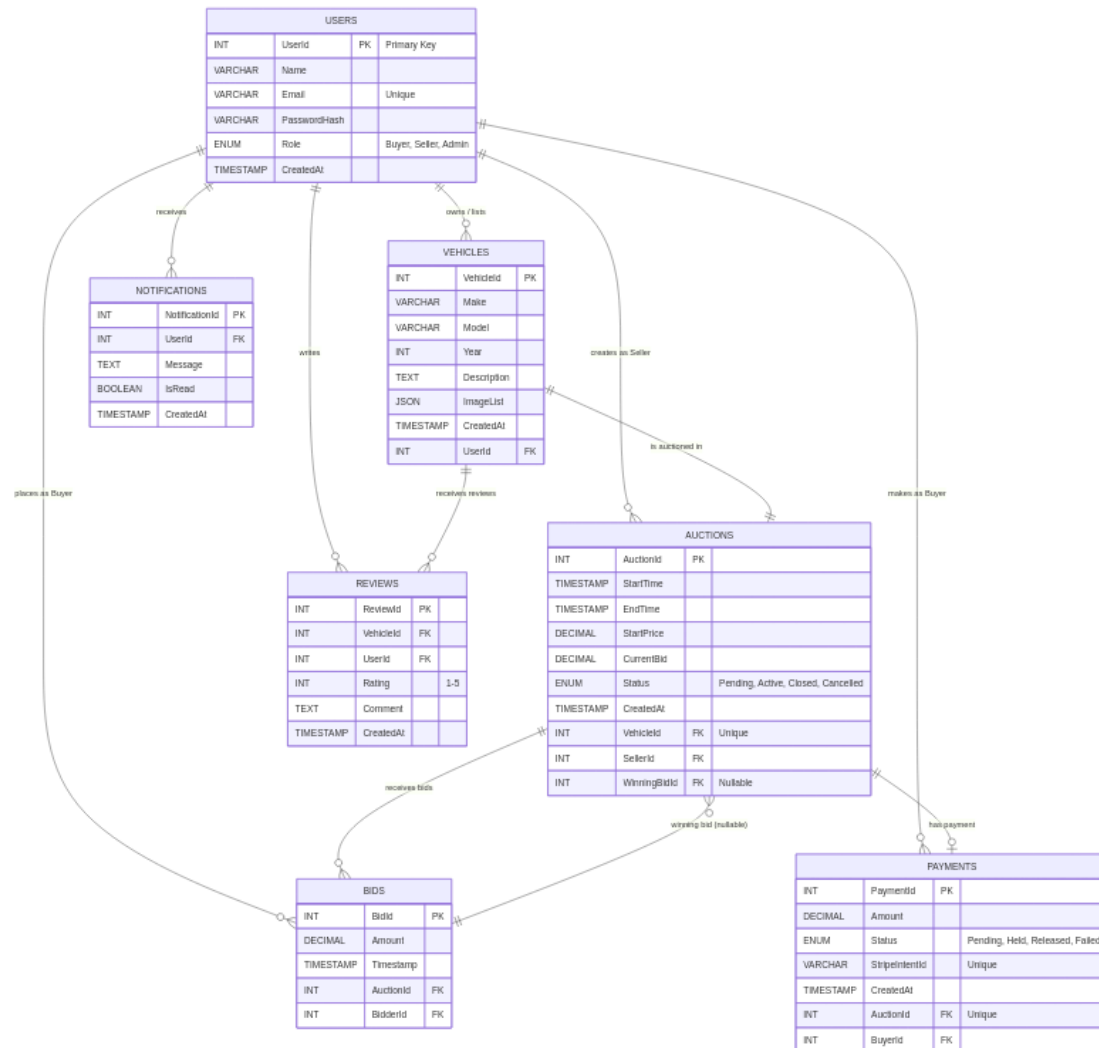

Figure 7.11: Entity-Relationship Diagram of Velocity Auction Database Architecture

7.5.2 Advanced Features and Optimization

The implementation leverages PostgreSQL capabilities including ULID primary keys via `pgx_ulid` extension for sortable 26-character identifiers, 25+ strategic indexes on foreign keys and frequently queried columns for sub-millisecond lookups, comprehensive CASCADE and RESTRICT constraints for referential integrity, check constraints enforcing business rules (strike limits 0-3, valid year ranges 1886-present, positive prices), unique constraints preventing duplicate active auctions and ensuring email/NIC/license plate uniqueness, and automated timestamp triggers for audit trailing. Array data types store multiple photos and features per vehicle, eliminating junction tables while maintaining query efficiency. The normalized design with separate brand and vehicle type tables enables efficient categorization through foreign key constraints.

Feature	Implementation Detail
ULID Primary Keys	Sortable 26-char identifiers via pgx_ulid extension
Strategic Indexing	25+ indexes on FKs, status fields, timestamps
Referential Integrity	CASCADE/RESTRICT constraints for data consistency
Business Rule Validation	Check constraints for strikes, prices, years, statuses
Unique Constraints	Prevention of duplicate auctions, emails, NICs
Automated Timestamps	Trigger-based updated_at management

Table 7.3: Key Database Features

The database architecture provides robust support for auction workflows, real-time bidding, secure payments, and ownership tracking while emphasizing performance through indexing, integrity through constraints, and maintainability through clear schema organization.

7.6 Contribution: Kulasuriya, DKKS

Student ID: 31883   **Role:** Fullstack Developer (API Focus)

7.6.1 API Endpoint Development (🔀)

My primary responsibility involved the design and implementation of several key API endpoints within the ASP.NET Core backend, focusing on vehicle management and user profile interactions. This included defining clear request/response DTOs, implementing business logic within service layers, and ensuring secure and efficient data handling.

[ADD CONTENT HERE: Detailed description of API endpoint development contributions...]

7.6.2 Integration with External Services (aws)

I developed the backend service responsible for integrating with AWS S3. This involved implementing the logic for securely uploading vehicle images submitted via the frontend, generating pre-signed URLs if needed for retrieval, and managing object keys associated with vehicle records in the database. Error handling for S3 operations was also implemented.

[ADD CONTENT HERE: Detailed description of external service integrations...]

7.6.3 Backend Business Logic (⚙)

Within the services related to my assigned endpoints, I implemented specific business logic. This included validating vehicle data before saving, handling image association logic, retrieving and formatting user profile information, and ensuring that data access patterns were efficient.

Key API Endpoints Implemented

Method	Endpoint Path	Primary Function
POST	/cars	Add a new vehicle to seller's inventory.
GET	/cars	Get list of seller's vehicles.
GET	/cars/id	Get details of a specific vehicle.
PUT	/cars/id	Update vehicle details.
DELETE	/cars/id	Delete a vehicle.
POST	/cars/upload	Upload an image for a vehicle to S3.
GET	/users/profile	Get current user's profile details.
PUT	/users/profile	Update current user's profile details.

Developed and tested these core endpoints, ensuring proper validation, authorization, and data handling.

7.7 Contribution: Dissanayake, DMDV

Student ID: 33041   **Role:** QA (Quality Assurance)

7.7.1 Test Planning and Strategy (📋)

My primary role involved establishing and executing the quality assurance strategy for the Online Vehicle Auction System. This began with creating a comprehensive test plan outlining the scope, objectives, resources, and schedule for testing activities. I defined different levels of testing, including unit, integration, API, and end-to-end testing, ensuring alignment with the project's Agile development process.

7.7.2 API Testing with Postman (🚀)

A major focus was validating the backend API endpoints. I created and maintained an extensive Postman collection covering all functionalities detailed in Appendix A. This involved:

- Writing test scripts within Postman to automate validation of response status codes, data schemas, and specific values.

- Testing both successful ("happy path") scenarios and various error conditions (e.g., invalid input, unauthorized access).

- Utilizing Postman environments to manage variables (like base URLs and authentication tokens) across different testing stages.

- Regularly running the collection to perform regression testing after new code changes.

7.7.3 Manual and Exploratory Testing (👤✓)

Beyond automated checks, I conducted manual testing on the integrated application, focusing on key user workflows from registration and login to placing bids and simulating the payment process. Exploratory testing was also performed to identify usability issues or edge cases missed by predefined test scripts.

7.7.4 Bug Reporting and Verification (🐛)

All identified defects were documented systematically using GitHub Issues. Each bug report included clear steps to reproduce, expected vs. actual results, severity level, and relevant environment details (e.g., browser). I collaborated with developers to clarify issues and rigorously verified fixes once implemented, ensuring that the reported problem was resolved and no regressions were introduced.

Key QA Metrics Summary

Metric	Value/Status
API Test Cases (Postman)	50+ Executed
Manual Test Scenarios	25 Covered
Bugs Reported	12 (All Closed)
Critical/High Severity Bugs	0 Open
Regression Test Pass Rate	100% (Final Cycle)

Simplified Testing Workflow

1. **Plan (⇋):** Define test scope, strategy, and create test cases based on requirements.

2. **Execute - API (🚀):** Run Postman collections against backend endpoints.

3. **Execute - Manual (👤):** Test user flows through the frontend application.

4. **Report (🐛):** Document any defects found with clear details.

5. **Verify (✔):** Retest fixed bugs and perform regression checks.

6. **Summarize (📊):** Collate results and report on overall quality.

My contributions ensured that the final delivered system met functional requirements, performed reliably, and adhered to quality standards.

Chapter 8

Conclusion and Future Work

8.1 Project Summary and Achievements

The Online Vehicle Auction System project successfully delivered a fully functional, secure, and scalable web application, meeting all initial requirements. Engineered using a modern three-tier, service-oriented architecture with ASP.NET Core 9, React.js, PostgreSQL, Redis, and WebSockets, the platform effectively addresses the limitations of traditional auctions. Key technical achievements include a high-performance real-time bidding engine, a secure escrow-based payment system via Stripe, and robust JWT-based authentication. Rigorous testing confirmed system stability and performance within the designed scope. The project demonstrated effective Agile methodology application and team collaboration, resulting in an on-schedule delivery of a high-quality system.

8.2 Limitations and Future Enhancements

While successful, the current system has limitations, including scalability constraints beyond the initial target, reliance solely on Stripe, English-only language support, and the absence of native mobile applications. Future enhancements are planned in phases. **Short-term** goals (3-6 months) include implementing advanced search (Elasticsearch), social logins, and a seller analytics dashboard. **Medium-term** plans (6-12 months) involve developing native iOS/Android apps and adding features like autobid and potentially live video streaming. The **long-term** vision (1-2 years) encompasses migrating to a microservices architecture to support global expansion (multi-language/currency) and exploring blockchain integration for enhanced transparency.

8.3 Learning Outcomes and Final Remarks

This project provided significant learning opportunities in modern web architecture, real-time systems, secure payment integration, and Agile teamwork. The team gained valuable technical expertise and developed crucial soft skills in communication and problem-solving. The Online Vehicle Auction System stands as a successful demonstration of applying sound architectural principles to deliver a production-ready solution. It provides a robust foundation for the planned future enhancements, positioning it well to evolve into a competitive commercial platform that enhances trust and efficiency in the vehicle resale market.

Appendix A: API Documentation

.1 API Overview

Base URL: `http://localhost:7001/api`

All endpoints (except `/auth/register` and `/auth/login`) require a JWT Bearer token **rfc7519**. Standard response fields:

- `success` : `bool` — true/false

- `message` : `string` — info or error

- `data` : `object` — payload

.2 Authentication (`/auth`)

Endpoint	Method	Description
`/auth/register`	POST	Create a new user (Buyer/Seller). Request body includes `name`, `email`, `password`, `role`. Returns userId and details.
`/auth/login`	POST	Login with email/password. Returns `accessToken`, `refreshToken`, `sessionToken`.

Table 1: Authentication Endpoints

.3 Auction Management (`/auctions`)

Endpoint	Method	Description
`/auctions`	POST	Create new auction (Seller only). Request body: `carId`, `startTime`, `endTime`, `startPrice`. Response: auctionId, status, currentBid.
`/auctions?page=&pageSize`	GET	Get paginated active auctions. Response includes `items` array, totalCount, page, totalPages.

Table 2: Auction Endpoints

.4 Vehicle Endpoints (`/cars`)

Endpoint	Method	Description
`/cars`	POST	Add vehicle to inventory (Seller only). Request: `make`, `model`, `year`, `description`. Response: carId and details.
`/cars/{auctionId}/bid`	POST	Place bid (Buyer only). Request: `amount`. Response: bidId, amount.
`/cars/upload`	POST	Upload vehicle image to S3 (Seller only) **awss3**. Multipart/form-data: `carId`, `file`. Response: s3Url, objectKey.

Table 3: Vehicle Endpoints

.5 Payment Endpoints (`/payments`)

Endpoint	Method	Description
`/payments/initiate`	POST	Buyer initiates payment after winning auction. Returns Stripe clientSecret **stripe**.
`/payments/confirm`	POST	Confirm payment after successful Stripe hold. Updates internal status to 'Held'.
`/payments/failed`	POST	Log failed payment attempt, status = 'Failed'.
`/payments/release`	POST	Admin/Seller triggers fund release to seller. Updates status to 'Released'.

Table 4: Payment Endpoints

.6 Notes & References

- Swagger UI for full interactive API documentation **swagger**: http://localhost:7001/swagger

- All requests require `Authorization:  Bearer <JWT>` header, except register/login.

- Responses are always in `{ success, message, data }` format.

- API design follows RESTful architectural principles **fielding2000**.

- Compact tables help reduce page count and maintain readability.

Additional References

1. Jones, M., Bradley, J., and Sakimura, N. (2015). *JSON Web Token (JWT)*. RFC 7519, IETF. Available at: https://tools.ietf.org/html/rfc7519

2. Amazon Web Services. (2024). *Amazon S3 API Reference*. Available at: https://docs.aws.amazon.com/s3/

3. Stripe Inc. (2024). *Stripe API Documentation*. Available at: https://stripe.com/docs/api

4. SmartBear Software. (2024). *Swagger UI - API Documentation Tool*. Available at: https://swagger.io/tools/swagger-ui/

5. Fielding, R. T. (2000). *Architectural Styles and the Design of Network-based Software Architectures*. Doctoral dissertation, University of California, Irvine.