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	Fullstack Developer	12345
Kulasuriya, DKKS	Frontend Developer	12345
Rajapaksha, TAW	Backend Developer	12345
Katugampala, KTN	Frontend Developer	12345
Suneth, SWND	Fullstack Developer	12345
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 (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.

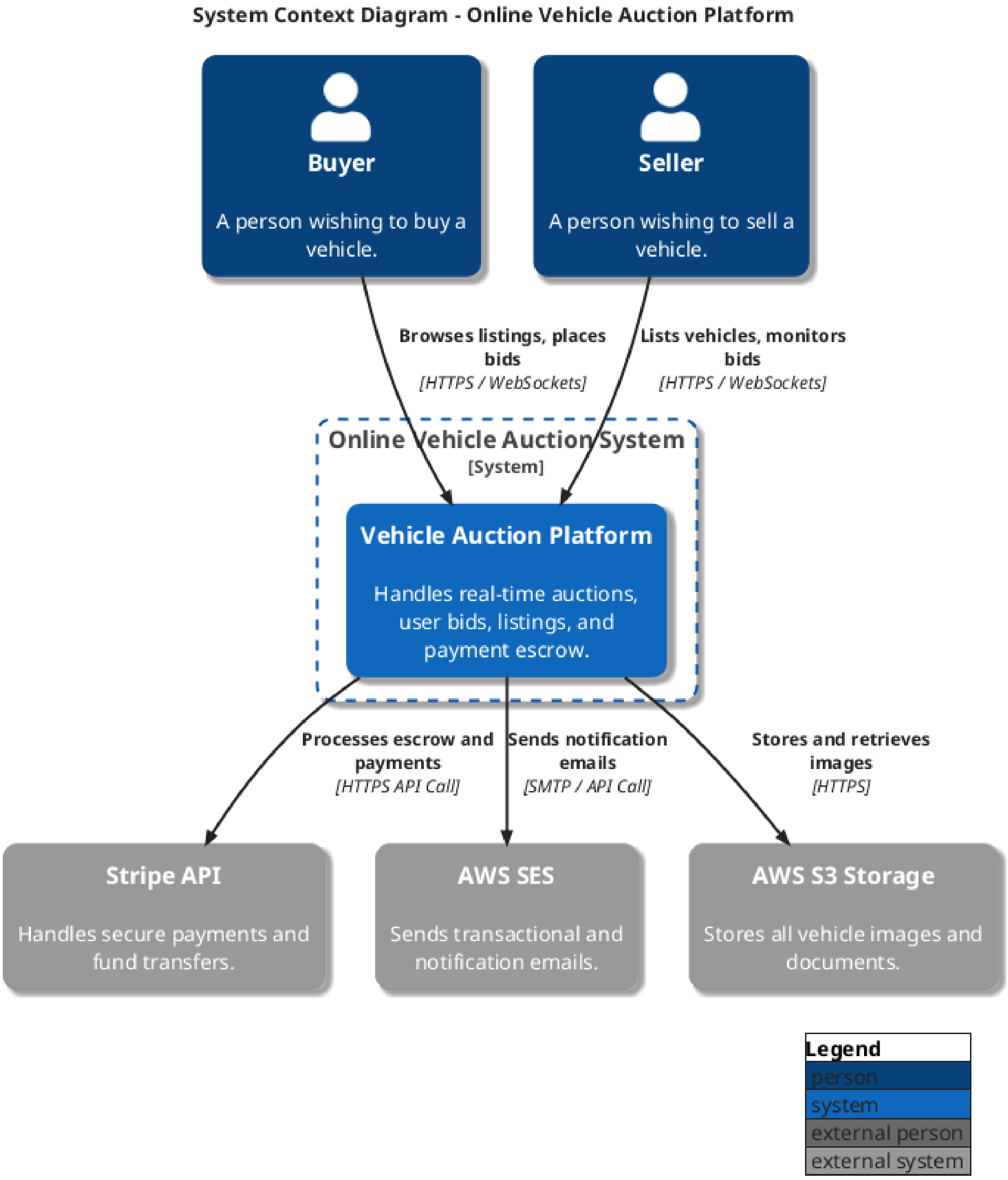

Figure 3.1: High-Level System Architecture

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.

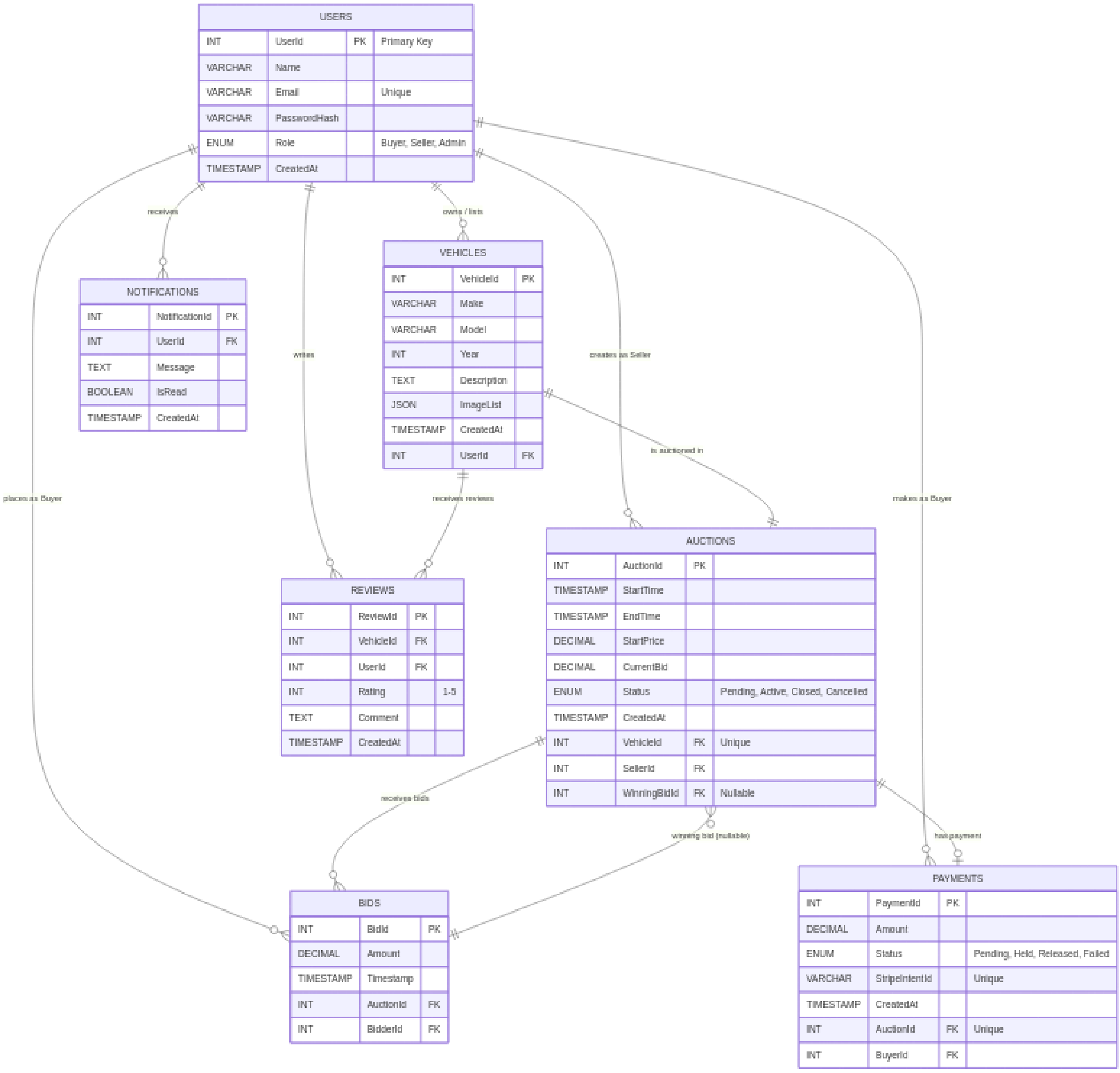

Figure 3.2: Entity-Relationship (ER) Diagram

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.

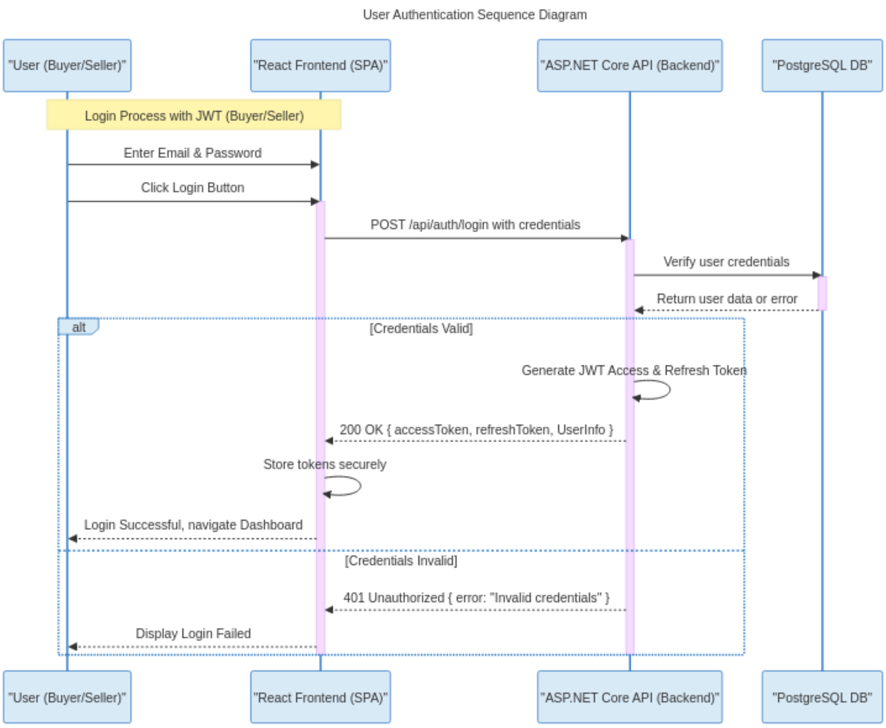

Figure 3.3: User Authentication Sequence Diagram

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.

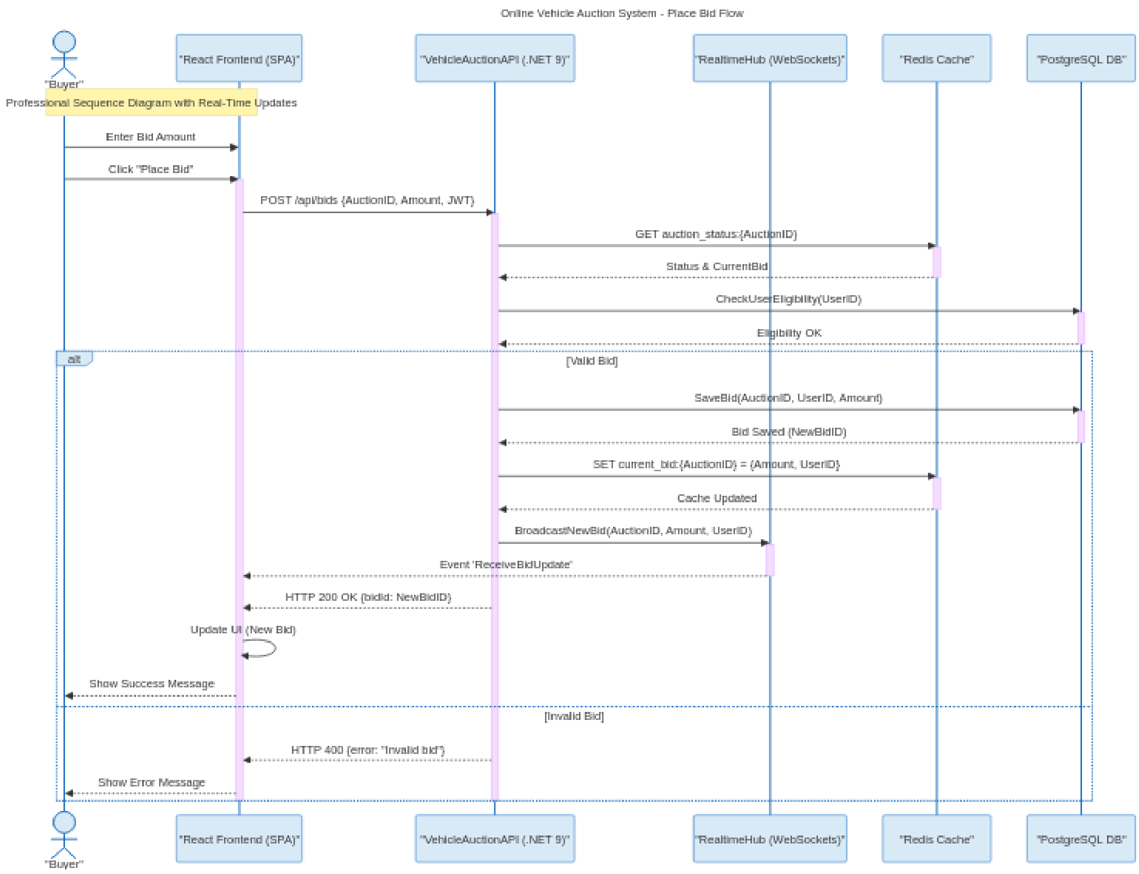

Figure 3.4: Real-time Bidding Sequence Diagram

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.

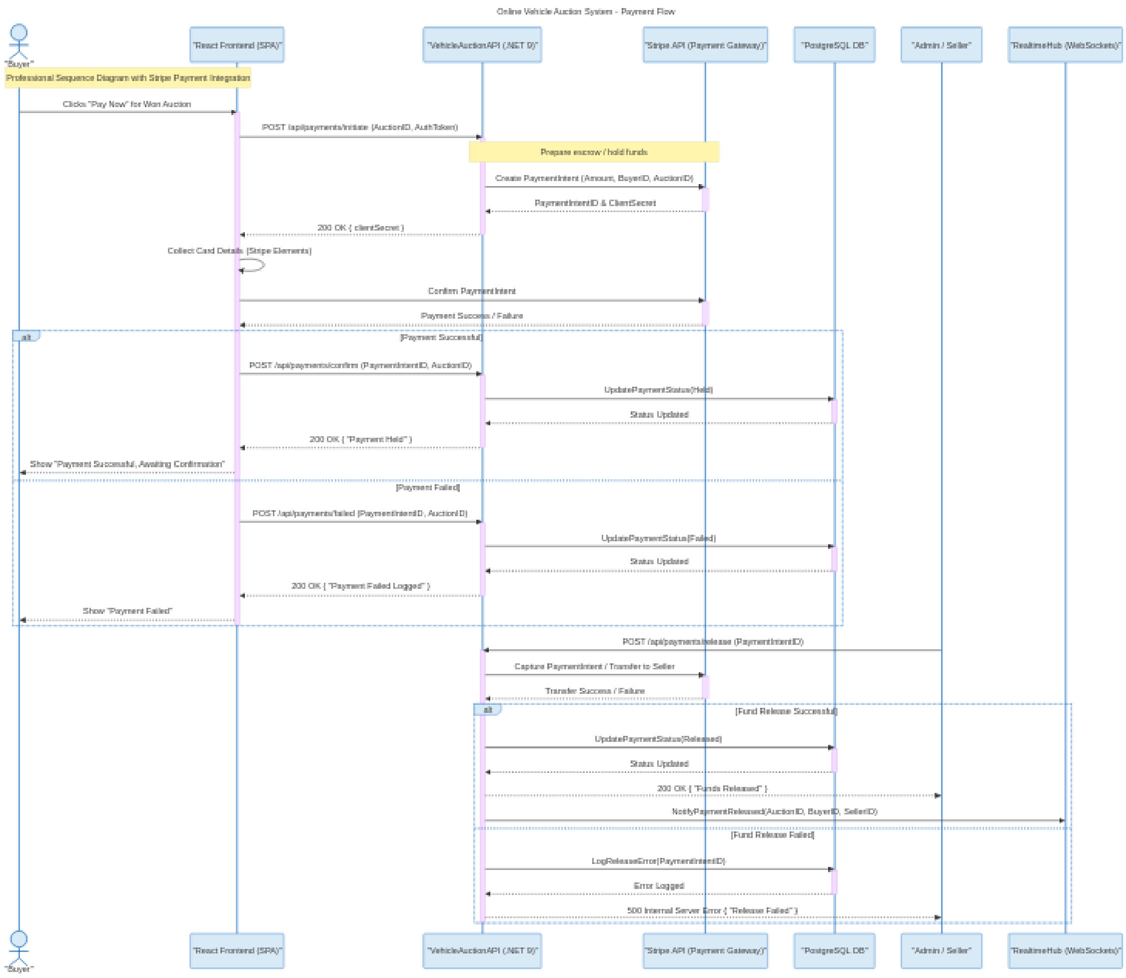

Figure 3.5: Escrow Payment Sequence Diagram

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

Chapter 5

Implementation Details

Chapter 6

Testing and Quality Assurance

Chapter 7

Individual Contributions

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. 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). 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.
`/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: http://localhost:7001/swagger
- All requests require `Authorization: Bearer <JWT>` header, except register/login.
- Responses are always in `{ success, message, data }` format.
- Compact tables help reduce page count and maintain readability.