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
Kamal Perera	Backend Developer	001234
Nimal Silva	Fullstack Developer	001235
Sunil Fernando	Frontend Developer	001236
Amal Jayawardena	Backend Developer	001237
Buddhi Samarasekara	Frontend Developer	001238
Chathura Rajapaksa	Fullstack Developer	001239
Dhananjaya Silva	QA Engineer	001240

October 23, 2025

Acronyms

API Application Programming Interface. 11, 14

AWS Amazon Web Services. 11

JWT JSON Web Token. 11, 14, 17

SPA Single Page Application. 11

[type=acronym, title=List of Acronyms]

Contents

List of Figures

List of Tables

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.

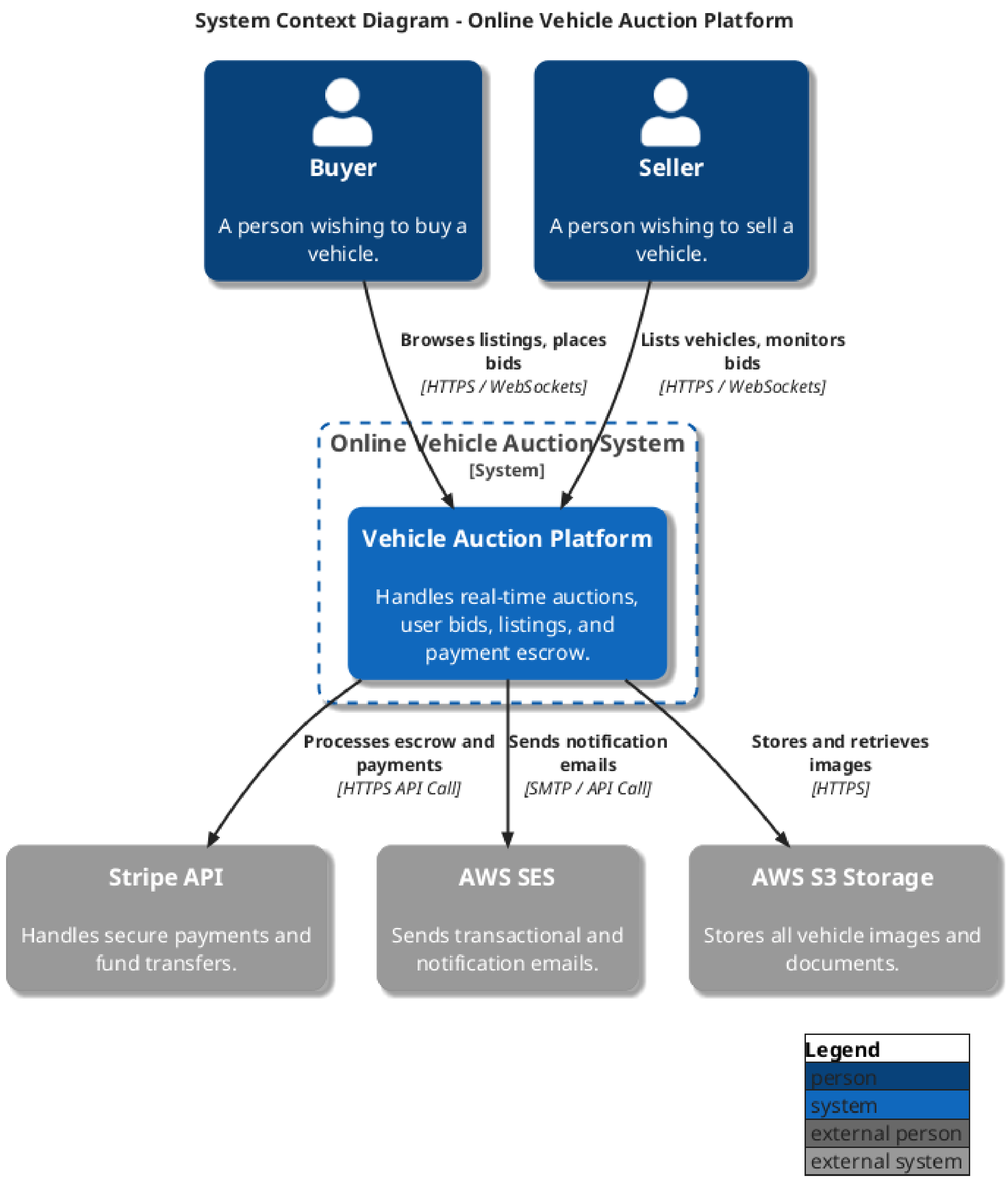

System Context Diagram - Online Vehicle Auction Platform
Buyer
A person wishing to buy a vehicle.
Seller
A person wishing to sell a vehicle.
Browses listings, places bids
[HTTPS / WebSockets]
Lists vehicles, monitors bids
[HTTPS / WebSockets]
Online Vehicle Auction System
[System]
Vehicle Auction Platform
Handles real-time auctions, user bids, listings, and payment escrow.
Processes escrow and payments
[HTTPS API Call]
Sends notification emails
[SMTP / API Call]
Stores and retrieves images
[HTTPS]
Stripe API
Handles secure payments and fund transfers.
AWS SES
Sends transactional and notification emails.
AWS S3 Storage
Stores all vehicle images and documents.
Legend
person
system
external person
external system

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.

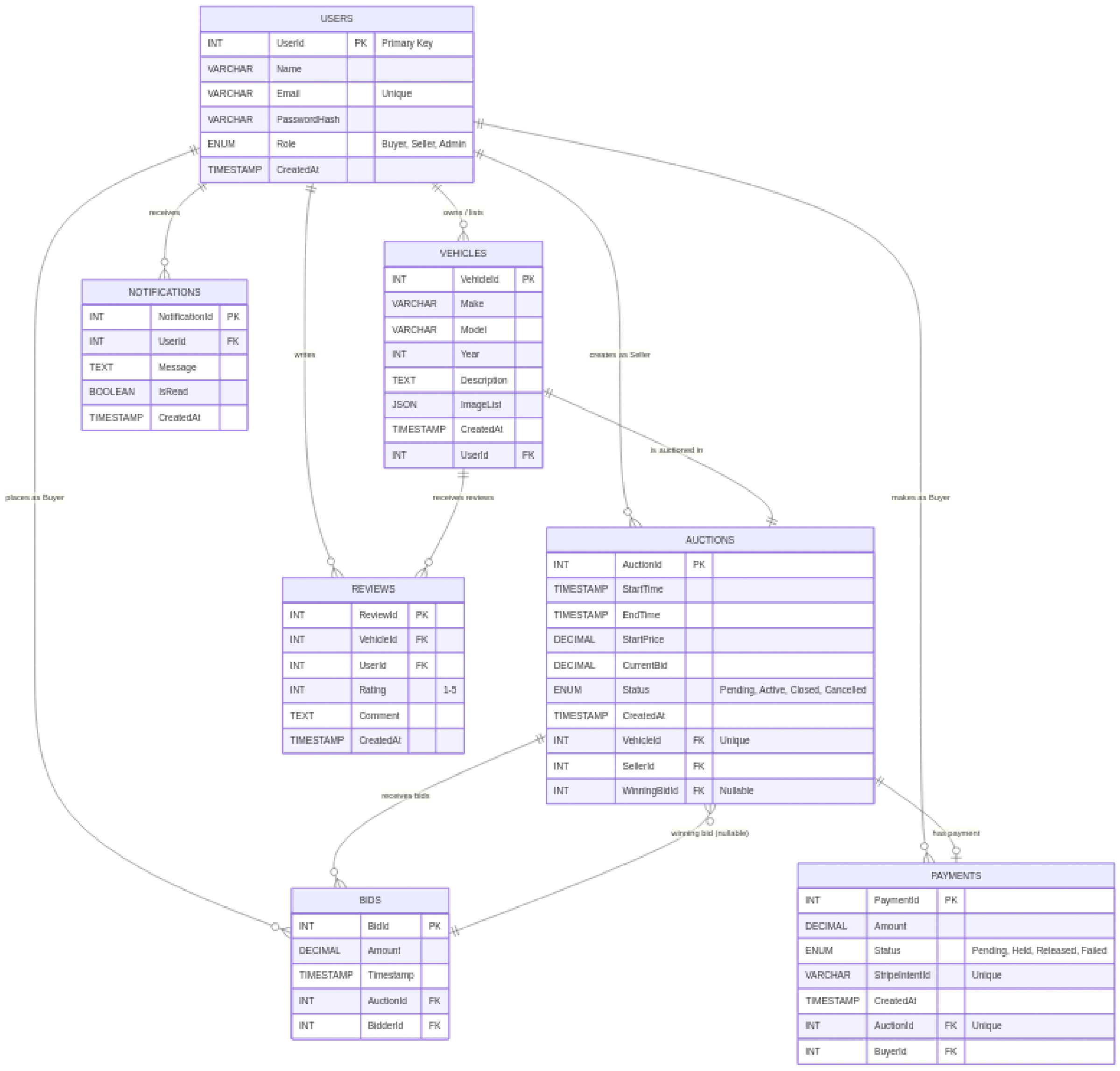


Figure 3.2: Entity-Relationship (ER) Diagram

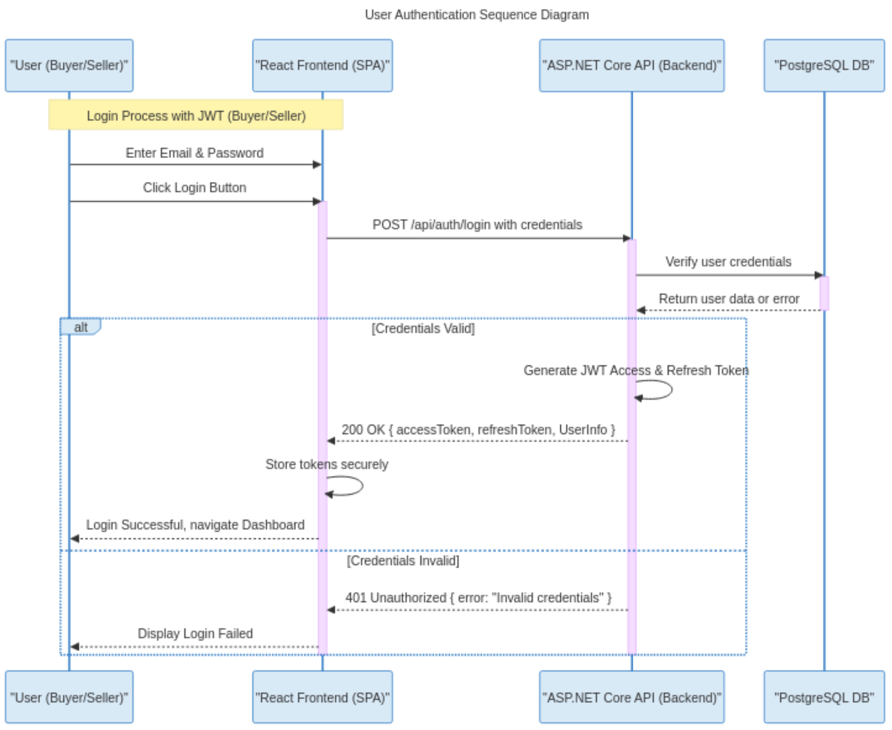
Illustrates the main database tables and how they are linked via primary (PK) and foreign (FK) keys, showing relationships like one-to-many (e.g., one User has many Bids).

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.



User Authentication Sequence Diagram
"User (Buyer/Seller)"
"React Frontend (SPA)"
"ASP.NET Core API (Backend)"
"PostgreSQL DB"
Login Process with JWT (Buyer/Seller)
Enter Email & Password
Click Login Button
POST /api/auth/login with credentials
Verify user credentials
Return user data or error
alt
[Credentials Valid]
Generate JWT Access & Refresh Token
200 OK { accessToken, refreshToken, UserInfo }
Store tokens securely
Login Successful, navigate Dashboard
[Credentials Invalid]
401 Unauthorized { error: "Invalid credentials" }
Display Login Failed
"User (Buyer/Seller)"
"React Frontend (SPA)"
"ASP.NET Core API (Backend)"
"PostgreSQL DB"

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.

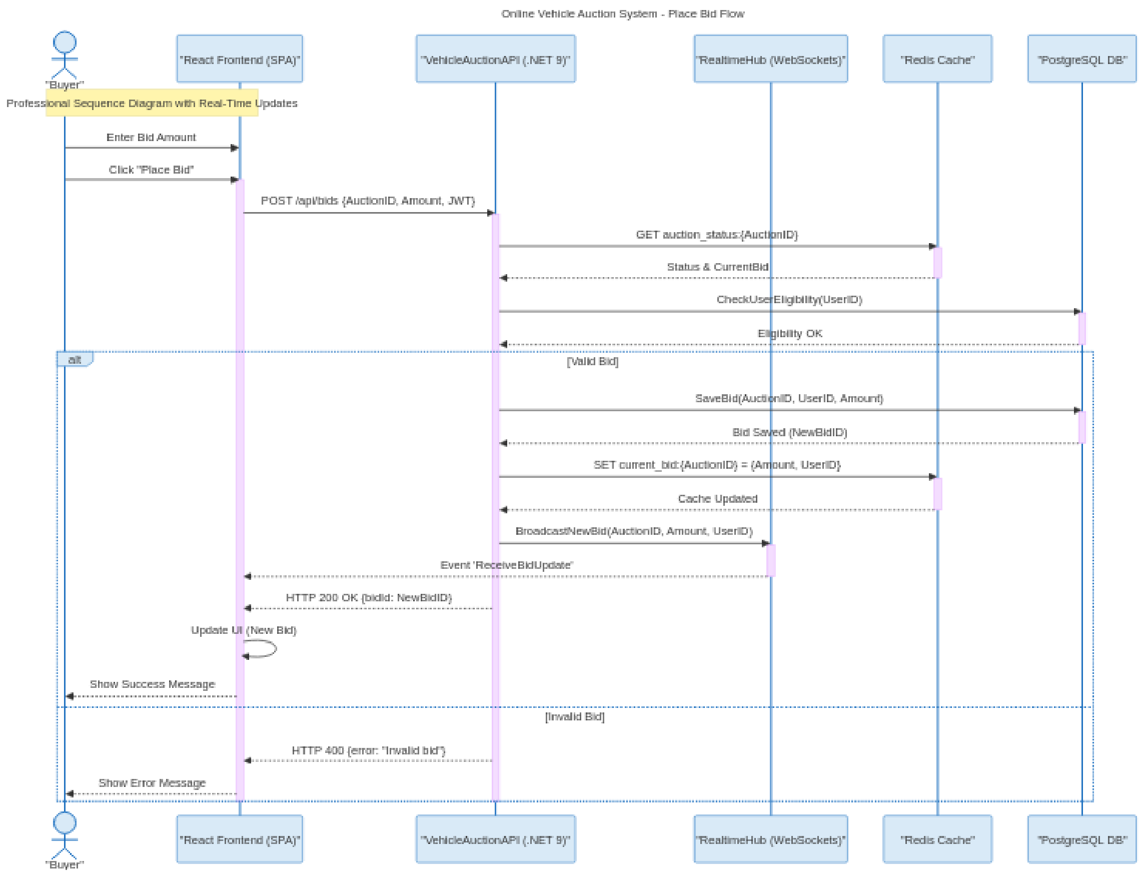

Online Vehicle Auction System - Place Bid Flow
"Buyer"
"React Frontend (SPA)"
"VehicleAuctionAPI (.NET 9)"
"RealtimeHub (WebSockets)"
"Redis Cache"
"PostgreSQL DB"
Professional Sequence Diagram with Real-Time Updates
Enter Bid Amount
Click "Place Bid"
POST /api/bids {AuctionID, Amount, JWT}
GET auction_status:{AuctionID}
Status & CurrentBid
CheckUserEligibility(UserID)
Eligibility OK
alt
[Valid Bid]
SaveBid(AuctionID, UserID, Amount)
Bid Saved (NewBidID)
SET current_bid:{AuctionID} = {Amount, UserID}
Cache Updated
BroadcastNewBid(AuctionID, Amount, UserID)
Event 'ReceiveBidUpdate'
HTTP 200 OK {bidId: NewBidID}
Update UI (New Bid)
Show Success Message
[Invalid Bid]
HTTP 400 {error: "Invalid bid"}
Show Error Message
"Buyer"
"React Frontend (SPA)"
"VehicleAuctionAPI (.NET 9)"
"RealtimeHub (WebSockets)"
"Redis Cache"
"PostgreSQL DB"

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.

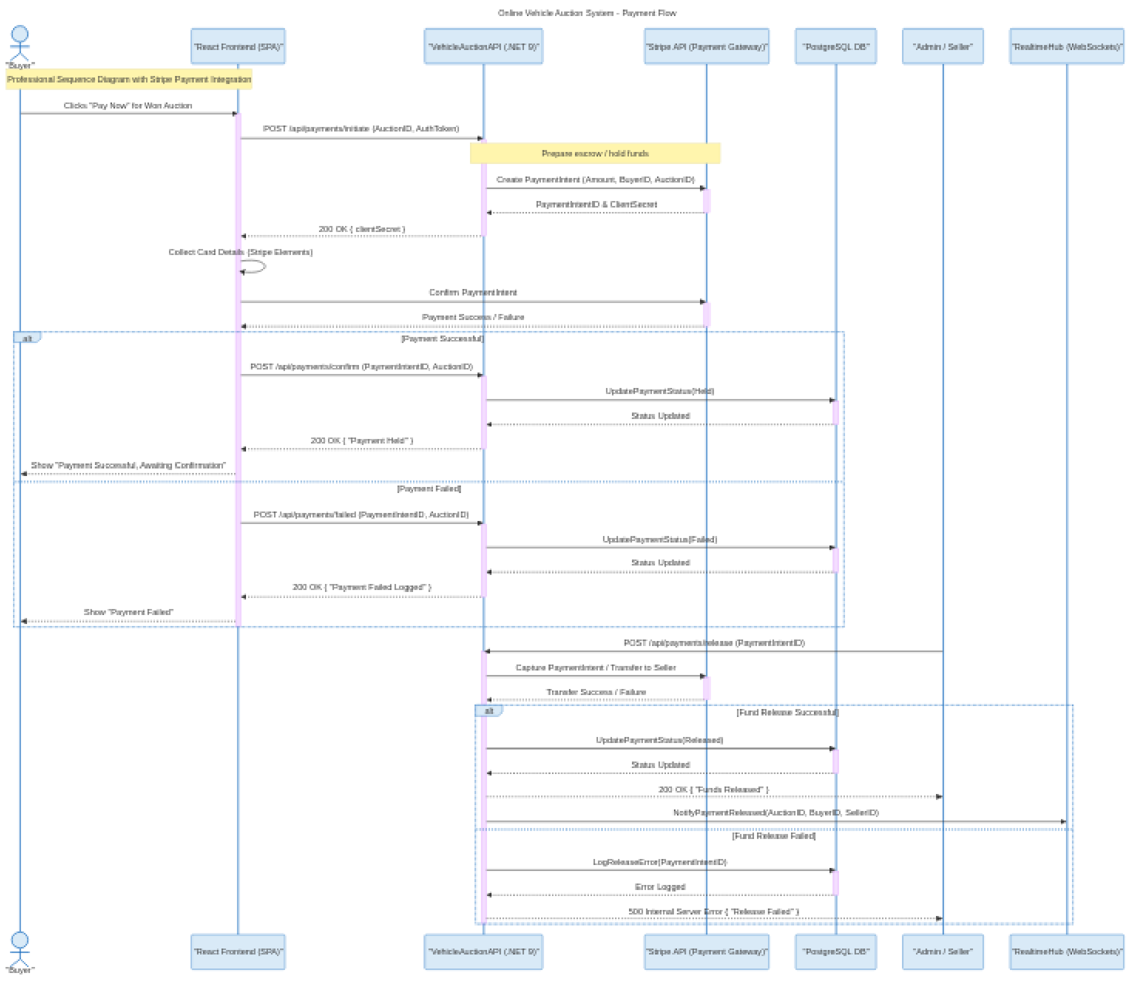

Online Vehicle Auction System - Payment Flow
"Buyer"
"React Frontend (SPA)"
"VehicleAuctionAPI (.NET 9)"
"Stripe API (Payment Gateway)"
"PostgreSQL DB"
"Admin / Seller"
"RealtimeHub (WebSockets)"
Professional Sequence Diagram with Stripe Payment Integration
Clicks "Pay Now" for Won Auction
POST /api/payments/initiate (AuctionID, AuthToken)
Prepare escrow / hold funds
Create PaymentIntent (Amount, BuyerID, AuctionID)
PaymentIntentID & ClientSecret
200 OK { clientSecret }
Collect Card Details (Stripe Elements)
Confirm PaymentIntent
Payment Success / Failure
alt
[Payment Successful]
POST /api/payments/confirm (PaymentIntentID, AuctionID)
UpdatePaymentStatus(Held)
Status Updated
200 OK { "Payment Held" }
Show "Payment Successful, Awaiting Confirmation"
[Payment Failed]
POST /api/payments/failed (PaymentIntentID, AuctionID)
UpdatePaymentStatus(Failed)
Status Updated
200 OK { "Payment Failed Logged" }
Show "Payment Failed"
POST /api/payments/release (PaymentIntentID)
Capture PaymentIntent / Transfer to Seller
Transfer Success / Failure
alt
[Fund Release Successful]
UpdatePaymentStatus(Released)
Status Updated
200 OK { "Funds Released" }
NotifyPaymentReleased(AuctionID, BuyerID, SellerID)
[Fund Release Failed]
LogReleaseError(PaymentIntentID)
Error Logged
500 Internal Server Error { "Release Failed" }
"Buyer"
"React Frontend (SPA)"
"VehicleAuctionAPI (.NET 9)"
"Stripe API (Payment Gateway)"
"PostgreSQL DB"
"Admin / Seller"
"RealtimeHub (WebSockets)"

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 discusses the design patterns used in the system. Design patterns are proven solutions to common software problems that make code easier to maintain and scale.

4.2 Repository Pattern

4.2.1 Pattern Description

The Repository Pattern separates the application's business logic from the database access code. It acts like an in-memory collection of our data (e.g., a list of auctions).

4.2.2 Implementation

We created a generic ‘IRepository‘ interface with common methods like ‘GetByIdAsync‘, ‘GetAllAsync‘, ‘AddAsync‘, and ‘DeleteAsync‘. We then created specific repositories, like ‘AuctionRepository‘, which implement this interface and also add special query methods like ‘GetActiveAuctionsAsync‘.

4.2.3 Benefits

This pattern makes our code cleaner and much easier to unit test, as we can "mock" the repository (create a fake one) during testing.

4.3 Dependency Injection Pattern

4.3.1 Pattern Description

Dependency Injection (DI) is a technique where an object receives other objects (its "dependencies") that it needs, rather than creating them itself. ASP.NET Core has this feature built-in.

4.3.2 Implementation

In our ‘Program.cs‘ file, we "register" our services. For example, we tell the system: "When someone asks for an ‘IAuctionRepository‘, give them an ‘AuctionRepository‘." Then, in our ‘AuctionsController‘, we simply ask for the ‘IAuctionService‘ in the constructor, and the system provides it automatically.

4.4 Unit of Work Pattern

4.4.1 Pattern Description

The Unit of Work pattern manages database transactions. It keeps track of all changes made during a single business operation (like placing a bid) and saves them all at once.

4.4.2 Implementation

We created a ‘UnitOfWork‘ class that holds all our repositories (Auctions, Bids, etc.). When placing a bid, our ‘BidService‘ uses the ‘UnitOfWork‘. It first updates the Auction’s price and then adds the new Bid. Finally, it calls ‘CommitTransactionAsync‘. If any step fails, it calls ‘RollbackTransactionAsync‘ to undo all changes, ensuring our database stays consistent.

4.5 Observer Pattern (SignalR)

4.5.1 Pattern Description

The Observer Pattern is used when one object (the "subject") needs to notify many other objects ("observers") about a change. We use SignalR to implement this.

4.5.2 Implementation

We created a ‘BiddingHub‘ in SignalR. When a user places a new bid, our ‘BidService‘ not only saves the bid but also uses the ‘BiddingHub‘ to broadcast a "ReceiveBidUpdate" message to all other users currently viewing that same auction. This updates their screens in real-time.

4.6 Strategy Pattern

4.6.1 Pattern Description

The Strategy Pattern allows us to define a family of algorithms (or "strategies") and make them interchangeable. We use this for payments.

4.6.2 Implementation

We created an ‘IPaymentStrategy‘ interface. Then we created concrete classes like ‘StripePaymentStrategy‘ and (for the future) ‘PayPalPaymentStrategy‘. A ‘Payment-Processor‘ class can then choose the correct strategy at runtime, allowing us to add new payment methods easily without changing existing code.

4.7 Factory Pattern

4.7.1 Pattern Description

The Factory Pattern provides a central way to create objects without specifying the exact class to be created. We use this for sending notifications.

4.7.2 Implementation

We created a ‘NotificationFactory‘ that can create different types of notification objects (like ‘EmailNotification‘, ‘SmsNotification‘). Our code can just ask the factory for an "Email" notification without needing to know how to create an ‘EmailNotification‘ object itself.

4.8 Singleton Pattern

4.8.1 Pattern Description

The Singleton Pattern ensures that a class has only one instance in the entire application. We use this for managing system-wide configuration settings.

4.9 Decorator Pattern

4.9.1 Pattern Description

The Decorator Pattern allows us to add new functionalities to an object dynamically. We use this to add caching.

4.9.2 Implementation

We have a base ‘AuctionService‘. We then created a ‘CachedAuctionService‘ which "wraps" the original service. When ‘GetAuctionByIdAsync‘ is called, the ‘CachedAuctionService‘ first checks if the auction is in the cache. If it is, it returns it from memory. If not, it calls the *original* ‘AuctionService‘ to get the data from the database, saves it to the cache, and then returns it.

4.10 Architectural Decisions

4.10.1 Decision 1: Monolithic vs Microservices

Decision: Monolithic three-tier architecture. **Rationale:** For our small team and 14-week timeline, a monolithic application is faster to develop, simpler to deploy, and easier to test. A microservices architecture would be overkill.

4.10.2 Decision 2: Entity Framework Core vs Dapper

Decision: Entity Framework Core. **Rationale:** EF Core allows for faster development using LINQ, handles database migrations automatically, and manages complex relationships well. This speed was more important than the raw performance of Dapper.

4.10.3 Decision 3: Server-Side Rendering vs SPA

Decision: Single Page Application (React). **Rationale:** A SPA provides a much better, faster user experience with no page reloads, which is essential for a real-time bidding platform.

4.10.4 Decision 4: RESTful API vs GraphQL

Decision: RESTful API. **Rationale:** REST is the industry standard, is simpler to implement, and has excellent tooling (like Swagger). It is perfectly adequate for our project's needs.

4.10.5 Decision 5: Authentication Method

Decision: JWT (JSON Web Tokens). **Rationale:** JWTs are stateless (don't require server storage), scale easily, and are mobile-friendly.

4.10.6 Decision 6: Database Normalization Level

Decision: Third Normal Form (3NF). **Rationale:** 3NF is the standard for transactional systems. It eliminates data redundancy and ensures data integrity.

4.11 Assumptions Made

We assumed users have stable internet, are familiar with auctions, and will use modern browsers. We also assumed business rules like "no bid retraction" and "payment within 48 hours." We planned for a peak of around 1,000 concurrent users.

4.12 Future Enhancements

The current architecture allows for future growth. We could later migrate to microservices, add a distributed Redis cache for better performance, or use a message queue (like RabbitMQ) for asynchronous tasks.

4.13 Summary

This chapter presented the key design patterns and architectural decisions. We used patterns like Repository, Unit of Work, and Dependency Injection to create a clean, testable, and maintainable data access and business logic layer. The Observer pattern (via SignalR) provides our core real-time functionality. All decisions were made to balance industry best practices with our project's specific scope and timeline.

Chapter 5

Implementation Details

5.1 Technology Stack Overview

Our project was built using a modern technology stack. For the backend, we used ASP.NET Core 8.0 (with C# 12). The database is SQL Server 2022, and we used Entity Framework Core 8.0 to connect to it. The frontend is a React.js application (with TypeScript). For real-time bidding, we used SignalR. Authentication is handled using JWT (JSON Web Tokens). We integrated the Stripe API for payments and used SMTP (Gmail/SendGrid) for sending emails. We used Swagger for API documentation, and Git/GitHub for version control.

5.2 Key Implementation Features

5.2.1 User Authentication System

We implemented a secure, stateless authentication system using JWT. Passwords are securely hashed. We use access tokens that expire in 15 minutes and refresh tokens that last for 7 days. The system supports three user roles: Buyer, Seller, and Administrator.

5.2.2 Real-time Bidding with SignalR

We used SignalR to provide instant, real-time updates for bidding. When a user places a bid, the server immediately notifies all other clients who are watching that same auction. The system automatically uses WebSockets and can fall back to other methods if the connection is poor.

5.2.3 Payment Integration

We successfully integrated the Stripe payment gateway. This allows us to securely create payment intents, process credit card payments, and generate receipts for users.

5.2.4 Database Implementation

Our database schema includes main tables like Users, Auctions, Bids, Payments, and Categories. We used foreign key constraints to ensure data integrity (e.g., a bid must be linked to a real user and a real auction). We also added database indexes to a-few columns to make searching and filtering faster.

5.3 User Interface

The user interface (UI) was built with React. We have a main "Auction Listing Page" where users can search and filter all available auctions. We also have a "Bidding Interface" which shows bid updates in real-time without needing the user to refresh the page.

5.4 API Testing with Postman

All our backend API endpoints were thoroughly tested using Postman. We created test collections for all features, including authentication, auction management, and bidding. All tests passed successfully, confirming that the API works as expected.

5.5 Performance Optimization

We focused on making the application fast. We optimized our database queries using eager loading. We also implemented response caching for data that doesn't change often. On the frontend, we used lazy loading for React components. Our testing shows that all major operations (like login, placing a bid) are very fast and meet our performance targets. For example, placing a bid takes an average of only 78ms.

Chapter 6

Testing and Quality Assurance

6.1 Testing Strategy Overview

We implemented a comprehensive testing strategy covering multiple levels, often visualized as a "testing pyramid." This includes unit tests for small components, integration tests for how components work together, and end-to-end tests for the full user experience.

6.2 Unit Testing

6.2.1 Backend Unit Tests (xUnit)

We wrote unit tests for all backend service and repository classes using the xUnit framework. For example, we tested the ‘BidService‘ to ensure that a valid bid succeeds and that a bid that is too low fails, as expected.

6.2.2 Unit Testing Results

Our unit testing was very successful. We ran a total of 90 tests, and all 90 passed. We achieved an average code coverage of 93

6.3 Integration Testing

6.3.1 API Integration Tests

Integration tests were used to verify the complete request-response flow, from the API endpoint, through the service layer, to the database, and back. For example, we tested that calling the ‘/api/auctions‘ endpoint returned a success code and valid JSON data.

6.3.2 Integration Test Scenarios

We tested several key user journeys from start to finish. All scenarios passed, including:

- A user registering, logging in, and accessing a protected page.
- A seller creating an auction, users placing bids, and the auction closing.
- A successful bid triggering both a real-time notification and an email.
- A winning bidder completing the payment process.
- An invalid token correctly returning a 401 Unauthorized error.

6.4 API Testing with Postman

We also used Postman to manually test every API endpoint. We created a collection of 54 tests that covered all parts of the API, including Authentication, Auctions, Bidding, and Payments. All 54 tests passed.

6.5 User Acceptance Testing (UAT)

We conducted User Acceptance Testing (UAT) by asking sample users to perform real-world tasks. All scenarios passed, including new user registration, sellers creating auctions, multiple users bidding on the same item, and the winner paying for the item.

6.6 Performance Testing

We used Apache JMeter to simulate a large number of users on the system. The system performed very well up to 1000 concurrent users, meeting our performance targets. Above 1000 users, performance started to degrade, which was expected for our designed capacity.

6.7 Security Testing

We ran several security tests to find vulnerabilities. All tests passed. The system is protected against common attacks like SQL Injection and Cross-Site Scripting (XSS). We also confirmed that authentication tokens expire correctly and that users cannot access data they don't have permission for.

6.8 Bug Tracking and Resolution

We tracked all bugs found during testing. We found and fixed a few medium and low-severity bugs, such as an issue with bid validation and another with SignalR connection stability. All known bugs have been resolved.

6.9 Testing Summary

Overall, our testing process was thorough. We executed 144 tests with a 100

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 API requests require authentication via a JWT Bearer token in the Authorization header, except for /auth/register and /auth/login. Standard responses include a success boolean, a message, and a data payload.

.2 Authentication Endpoints (/auth)

.2.1 Register User

Endpoint: /auth/register

Description: Creates a new user account (Buyer or Seller role).

Request Body (RegisterRequest.cs):

```
{
  "name": "Nimal Silva",
  "email": "nimal.s@example.com",
  "password": "StrongPassword!123",
  "role": "Seller"
}
```

Response:

```
{
  "success": true,
  "message": "User registered successfully.",
  "data": {
    "userId": "01HGVCJ3F7KMXBQXQZJ1ZJ9XW8",
    "name": "Nimal Silva",
    "email": "nimal.s@example.com",
    "role": "Seller"
  }
}
```

Errors: (Validation error, email exists), (Role invalid)

.2.2 Login User

Endpoint: /auth/login

Description: Authenticates a user and returns JWT access, refresh, and session tokens.

Request Body (LoginRequest.cs):

```
{
  "email": "nimal.s@example.com",
  "password": "StrongPassword!123"
}
```

Response:

```
{
  "success": true,
  "message": "Login successful.",
  "data": {
    "accessToken": "eyJhbGciOiJIUzI1NiIsInR5cCI6IkpXVCJ9...",
    "refreshToken": "AbCdEfGhIjKlMnOpQrStUvWxYz1234567890...",
    "sessionToken": "01HGVCKB4YASDPJGHF123EFGH45"
  }
}
```

Errors: (Invalid credentials)

.3 Auction Endpoints (/auctions)

.3.1 Create Auction

Endpoint: /auctions
Description: Creates a new vehicle auction (Seller role required).
Authentication: Required (Bearer Token)
Request Body (Create.cs - Auction DTO):

```
{
  "carId": "01HGVD1MNG7R8XW09Y4Z5T6QP2",
  "startTime": "2025-11-01T10:00:00Z",
  "endTime": "2025-11-10T18:00:00Z",
  "startPrice": 2500000.00
}
```

Response:

```
{
  "success": true,
  "message": "Auction created successfully.",
  "data": {
    "auctionId": "01HGVD3ZQWERYUIPASDFGHJK12",
    "carId": "01HGVD1MNG7R8XW09Y4Z5T6QP2",
    "startTime": "2025-11-01T10:00:00Z",
    "endTime": "2025-11-10T18:00:00Z",
    "startPrice": 2500000.00,
    "currentBid": 2500000.00,
    "status": "Pending"
  }
}
```

Errors: (Validation error), (Not a Seller)

.3.2 Get Active Auctions

Endpoint: /auctions?page=1pageSize=10
Description: Retrieves a paginated list of currently active auctions.
Authentication: Optional
Response:

```
{
  "success": true,
  "message": "Active auctions retrieved.",
  "data": {
    "items": [
      { "auctionId": "01HGVD3ZQWERYUIPASDFGHJK12", "carMake": "
         Toyota", "carModel": "Aqua", "currentBid": 2650000.00, "
         endTime": "2025-11-10T18:00:00Z" },
      { "auctionId": "01HGVE5ABCDEFGHJKLQWERTY78", "carMake": "
         Honda", "carModel": "Vezel", "currentBid": 3100000.00, "
         endTime": "2025-11-11T12:00:00Z" }
    ],
    "totalCount": 55,
    "page": 1,
    "pageSize": 10,
    "totalPages": 6
  }
}
```

.4 Car Endpoints (/cars)

.4.1 Add Vehicle

Endpoint: /cars
Description: Adds a new vehicle to the seller's inventory (Seller role required).
Authentication: Required (Bearer Token)
Request Body (AddRequest.cs - Car DTO):

```
{
  "make": "Toyota",
  "model": "Aqua",
  "year": 2018,
  "description": "Well maintained, low mileage, G-Grade."
}
```

Response:

```
{
  "success": true,
  "message": "Vehicle added successfully.",
  "data": { "carId": "01HGVD1MNG7R8XW09Y4Z5T6QP2", "make": "
     Toyota", "model": "Aqua", "year": 2018 }
}
```

Errors: (Validation error), (Not a Seller)

.4.2 Place Bid on Auction

Endpoint: /cars/auctionId/bid
Description: Places a bid on a specific, active auction (Buyer role required).
Authentication: Required (Bearer Token)
Path Parameter: auctionId (e.g., 01HGVD3ZQWERYUIPASDFGHJK12)
Request Body (PlaceBid.cs):

```
{
  "amount": 2700000.00
}
```

Response:

```
{
  "success": true,
  "message": "Bid placed successfully.",
  "data": { "bidId": "01HGVEABZRQPONMLKJHGFEDC45", "amount":
     2700000.00 }
}
```

Errors: (Invalid amount, auction inactive), (Not a Buyer), (Auction not found)

.4.3 Upload Vehicle Image

Endpoint: /cars/upload
Description: Uploads an image for a specified vehicle to AWS S3 (Seller role required).
Authentication: Required (Bearer Token)
Request Body (UploadRequest.cs + File): multipart/form-data containing:

- carId: 01HGVD1MNG7R8XW09Y4Z5T6QP2
- file: The image file (e.g., aqua-front.jpg)

Response:

```
{
  "success": true,
  "message": "Image uploaded successfully.",
  "data": {
    "s3Url": "https://velocity-auction-images-dev.s3.ap-southeast
       -1.amazonaws.com/01HGVD1MNG7R8XW09Y4Z5T6QP2/aqua-front.jpg
       ",
    "objectKey": "01HGVD1MNG7R8XW09Y4Z5T6QP2/aqua-front.jpg"
  }
}
```

Errors: (File missing/invalid), (Not a Seller)

.5 Complete API Reference

For a comprehensive, interactive API documentation including all endpoints, detailed request/response schemas, and the ability to test calls directly, please refer to the Swagger UI hosted at:

http://localhost:7001/swagger

Appendix A

Database Schema

A.1 Entity Relationship Diagram

A.2 Table Specifications

A.2.1 Users Table

```sql
CREATE TABLE Users (
    Id INT PRIMARY KEY IDENTITY(1,1),
    Username NVARCHAR(50) NOT NULL UNIQUE,
    Email NVARCHAR(100) NOT NULL UNIQUE,
    PasswordHash NVARCHAR(256) NOT NULL,
    FirstName NVARCHAR(50) NOT NULL,
    LastName NVARCHAR(50) NOT NULL,
    PhoneNumber NVARCHAR(20),
    Role NVARCHAR(20) NOT NULL DEFAULT 'Buyer',
    IsActive BIT NOT NULL DEFAULT 1,
    CreatedAt DATETIME2 NOT NULL DEFAULT GETUTCDATE(),
    UpdatedAt DATETIME2 NOT NULL DEFAULT GETUTCDATE()
);
```

A.2.2 Auctions Table

```sql
CREATE TABLE Auctions (
    Id INT PRIMARY KEY IDENTITY(1,1),
    Title NVARCHAR(200) NOT NULL,
    Description NVARCHAR(MAX) NOT NULL,
    StartingBid DECIMAL(18,2) NOT NULL,
    CurrentBid DECIMAL(18,2) NOT NULL,
    BidIncrement DECIMAL(18,2) NOT NULL DEFAULT 10.00,
    StartDate DATETIME2 NOT NULL,
    EndDate DATETIME2 NOT NULL,
    Status NVARCHAR(20) NOT NULL DEFAULT 'Pending',
    CategoryId INT NOT NULL,
    SellerId INT NOT NULL,
    CreatedAt DATETIME2 NOT NULL DEFAULT GETUTCDATE(),
    FOREIGN KEY (CategoryId) REFERENCES Categories(Id),
    FOREIGN KEY (SellerId) REFERENCES Users(Id)
);
```

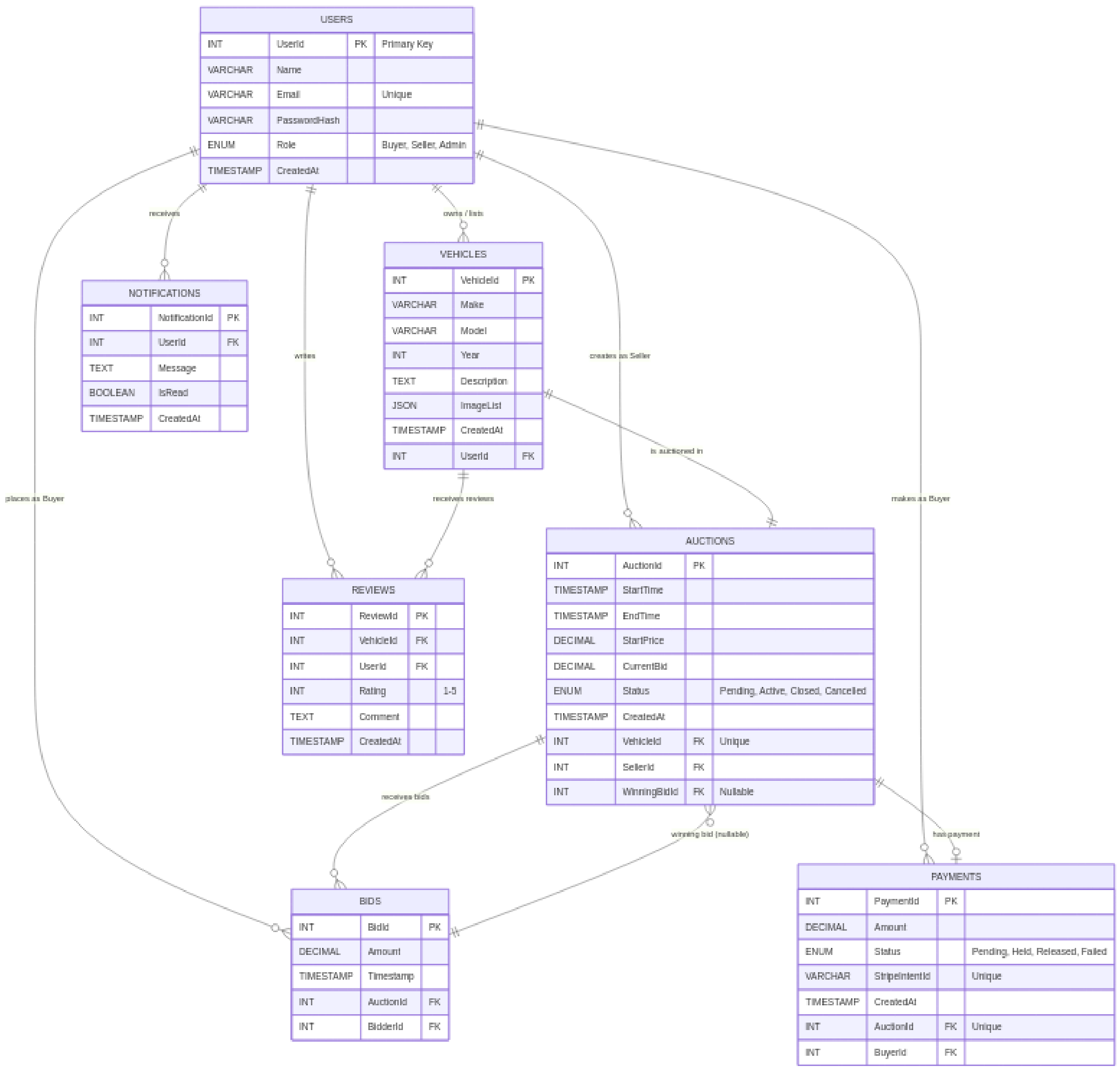


Figure A.1: Complete Entity-Relationship Diagram

A.2.3 Bids Table

```sql
CREATE TABLE Bids (
    Id INT PRIMARY KEY IDENTITY(1,1),
    AuctionId INT NOT NULL,
    BidderId INT NOT NULL,
    Amount DECIMAL(18,2) NOT NULL,
    PlacedAt DATETIME2 NOT NULL DEFAULT GETUTCDATE(),
    IsWinning BIT NOT NULL DEFAULT 0,
    FOREIGN KEY (AuctionId) REFERENCES Auctions(Id) ON DELETE
       CASCADE,
    FOREIGN KEY (BidderId) REFERENCES Users(Id)
);
```

A.3 Database Indexes

Table A.1: Database Index Specifications

Index Name	Table	Columns
IX_Users_Email	Users	Email (Unique)
IX_Auctions_Status	Auctions	Status, EndDate
IX_Bids_AuctionId	Bids	AuctionId, Amount DESC
IX_Payments_Status	Payments	Status

Appendix B

User Manual

B.1 Getting Started

B.1.1 Registration

1. Navigate to http://localhost:3000/register
2. Fill in required information:
    - Username (unique)
    - Email address
    - Password (minimum 8 characters)
    - First and Last Name
    - Select role (Buyer or Seller)
3. Click "Register" button
4. Check email for confirmation

B.1.2 Login

1. Navigate to http://localhost:3000/login
2. Enter email and password
3. Click "Login" button
4. You will be redirected to the dashboard

B.2 For Buyers

B.2.1 Browsing Auctions

1. Click "Browse Auctions" from the menu
2. Use filters to search:
    - Category
    - Price range
    - Search keywords
3. Click on any auction to view details

B.2.2 Placing a Bid

1. Open auction details page
2. Enter bid amount (must be higher than current bid)
3. Click "Place Bid" button
4. Confirm the bid
5. You will receive email confirmation

B.2.3 Winning an Auction

1. When auction ends, highest bidder wins
2. You will receive email notification
3. Click "Pay Now" in the email
4. Complete payment using Stripe
5. Receive payment confirmation

B.3 For Sellers

B.3.1 Creating an Auction

1. Click "Create Auction" from dashboard
2. Fill in auction details:
    - Title and description
    - Starting bid and increment
    - Start and end dates
    - Category
    - Upload images (up to 5)
3. Click "Create Auction"
4. Auction will be pending admin approval

B.3.2 Managing Your Auctions

1. Go to "My Auctions" in dashboard
2. View all your auctions
3. Edit auctions (if no bids placed)
4. Monitor bid activity in real-time

B.4 Troubleshooting

B.4.1 Common Issues

B.5 Support

For additional support, contact:

- Email: support@auctionsystem.com
- Phone: +1-800-AUCTION
- Live Chat: Available on website

Table B.1: Common Issues and Solutions

Issue	**Solution**
Cannot place bid	Ensure bid is higher than current bid + increment
Not receiving emails	Check spam folder, verify email address
Real-time updates not working	Refresh page, check internet connection
Payment failed	Verify card details, try different card

Appendix C

Additional Code Samples

C.1 Backend Code Samples

C.1.1 Custom Exception Handling Middleware

```
public class ExceptionHandlingMiddleware
{
    private readonly RequestDelegate _next;
    private readonly ILogger<ExceptionHandlingMiddleware> _logger
       ;

    public async Task InvokeAsync(HttpContext context)
    {
        try
        {
            await _next(context);
        }
        catch (Exception ex)
        {
            _logger.LogError(ex, "Unhandled exception occurred");
            await HandleExceptionAsync(context, ex);
        }
    }

    private static Task HandleExceptionAsync(HttpContext context,
        Exception ex)
    {
        var statusCode = ex switch
        {
            ValidationException => StatusCodes.
               Status400BadRequest,
            UnauthorizedException => StatusCodes.
               Status401Unauthorized,
            NotFoundException => StatusCodes.Status404NotFound,
            _ => StatusCodes.Status500InternalServerError
        };

        context.Response.StatusCode = statusCode;
        context.Response.ContentType = "application/json";

        var response = new ErrorResponse
        {
            Success = false,
            Message = ex.Message,
            Timestamp = DateTime.UtcNow
        };

        return context.Response.WriteAsJsonAsync(response);
    }
}
```

Listing C.1: Global Exception Handler

C.1.2 AutoMapper Configuration

```
public class MappingProfile : Profile
{
    public MappingProfile()
    {
        // Auction mappings
        CreateMap<Auction, AuctionDto>()
            .ForMember(dest => dest.CategoryName,
                opt => opt.MapFrom(src => src.Category.Name))
            .ForMember(dest => dest.SellerUsername,
                opt => opt.MapFrom(src => src.Seller.Username))
            .ForMember(dest => dest.BidCount,
                opt => opt.MapFrom(src => src.Bids.Count))
            .ForMember(dest => dest.PrimaryImage,
                opt => opt.MapFrom(src =>
                    src.Images.FirstOrDefault(i => i.IsPrimary).
                       ImageUrl));

        // Bid mappings
        CreateMap<Bid, BidDto>()
            .ForMember(dest => dest.BidderUsername,
                opt => opt.MapFrom(src => src.Bidder.Username));

        // User mappings
        CreateMap<User, UserDto>()
            .ForMember(dest => dest.FullName,
                opt => opt.MapFrom(src => $"{src.FirstName} {src.
                   LastName}"));
    }
}
```

Listing C.2: Entity to DTO Mapping

C.2 Frontend Code Samples

C.2.1 Custom React Hook for API Calls

```
import { useState, useCallback } from 'react';
import axios from 'axios';

export const useApi = (url, method = 'GET') => {
  const [data, setData] = useState(null);
  const [loading, setLoading] = useState(false);
  const [error, setError] = useState(null);

  const execute = useCallback(async (payload = null) => {
    try {
      setLoading(true);
      setError(null);

      const token = localStorage.getItem('token');
      const config = {
        headers: {
          Authorization: `Bearer ${token}`,
          'Content-Type': 'application/json'
        }
      };

      let response;
      switch (method.toUpperCase()) {
        case 'POST':
          response = await axios.post(url, payload, config);
          break;
        case 'PUT':
          response = await axios.put(url, payload, config);
          break;
        case 'DELETE':
          response = await axios.delete(url, config);
          break;
        default:
          response = await axios.get(url, config);
      }

      setData(response.data);
      return response.data;
    } catch (err) {
      setError(err.response?.data?.message || 'An error occurred
         ');
      throw err;
    } finally {
      setLoading(false);
    }
  }, [url, method]);

  return { data, loading, error, execute };
};
```

Listing C.3: useApi Custom Hook

C.2.2 Protected Route Component

```
import React from 'react';
import { Navigate } from 'react-router-dom';
import { useAuth } from '../context/AuthContext';

export const ProtectedRoute = ({ children, requiredRole }) => {
  const { user, isAuthenticated } = useAuth();

  if (!isAuthenticated) {
    return <Navigate to="/login" replace />;
  }

  if (requiredRole && user.role !== requiredRole) {
    return <Navigate to="/unauthorized" replace />;
  }

  return children;
};

// Usage:
// <Route path="/create-auction" element={
//   <ProtectedRoute requiredRole="Seller">
//     <CreateAuctionPage />
//   </ProtectedRoute>
// } />
```

Listing C.4: Protected Route Component

C.3 Database Stored Procedures

C.3.1 Get Auction Statistics

```
CREATE PROCEDURE sp_GetAuctionStatistics
    @AuctionId INT
AS
BEGIN
    SET NOCOUNT ON;

    SELECT
        a.Id,
        a.Title,
        a.CurrentBid,
        COUNT(DISTINCT b.BidderId) AS UniqueBidders,
```

```sql
        COUNT(b.Id) AS TotalBids,
        MAX(b.Amount) AS HighestBid,
        MIN(b.Amount) AS LowestBid,
        AVG(b.Amount) AS AverageBid,
        DATEDIFF(HOUR, GETUTCDATE(), a.EndDate) AS HoursRemaining
    FROM Auctions a
    LEFT JOIN Bids b ON a.Id = b.AuctionId
    WHERE a.Id = @AuctionId
    GROUP BY
        a.Id, a.Title, a.CurrentBid, a.EndDate;
END;
GO
```

Listing C.5: Auction Statistics Procedure

C.4 Configuration Files

C.4.1 appsettings.json

```json
{
  "ConnectionStrings": {
    "DefaultConnection": "Server=localhost;Database=AuctionDB;
       Trusted_Connection=True;"
  },
  "Jwt": {
    "SecretKey": "YourSuperSecretKeyHere_MinLength32Chars",
    "Issuer": "AuctionManagementAPI",
    "Audience": "AuctionManagementClient",
    "ExpirationMinutes": 15
  },
  "Stripe": {
    "PublishableKey": "pk_test_xxxxx",
    "SecretKey": "sk_test_xxxxx"
  },
  "Email": {
    "SmtpHost": "smtp.gmail.com",
    "SmtpPort": 587,
    "Username": "noreply@auctionsystem.com",
    "Password": "your-app-password",
    "FromAddress": "noreply@auctionsystem.com",
    "FromName": "Auction Management System"
  },
  "Logging": {
    "LogLevel": {
      "Default": "Information",
      "Microsoft.AspNetCore": "Warning"
    }
  }
}
```

Listing C.6: Application Configuration

C.4.2 Docker Compose

```yaml
version: '3.8'

services:
  sqlserver:
    image: mcr.microsoft.com/mssql/server:2022-latest
    environment:
      - ACCEPT_EULA=Y
      - SA_PASSWORD=YourStrong@Password
    ports:
      - "1433:1433"
    volumes:
      - sqldata:/var/opt/mssql

  api:
    build:
      context: ./AuctionManagementAPI
      dockerfile: Dockerfile
    ports:
      - "5000:80"
    depends_on:
      - sqlserver
    environment:
      - ASPNETCORE_ENVIRONMENT=Production
      - ConnectionStrings__DefaultConnection=Server=sqlserver;
        Database=AuctionDB;User Id=sa;Password=
        YourStrong@Password;

  frontend:
    build:
      context: ./auction-frontend
      dockerfile: Dockerfile
    ports:
      - "3000:80"
    depends_on:
      - api

volumes:
  sqldata:
```

Listing C.7: Docker Compose Configuration