Auction Architects

CarFax Bidding
Software Requirements Specification
For Bidding Subsystem

Version 2.0

Shahed Ahmed, Rafid Rahman, Alex Abraham, Muhammad Ahmed

AuctionArchitects	Version: 2.0
Software Requirements Specification	Date: 12/11/24
<document identifier>	

Revision History

Date	Version	Description	Author
15/10/24	1.0	<details>	<name>
12/11/24	2.0	<details>	<name>

AuctionArchitects	Version: 2.0
Software Requirements Specification	Date: 12/11/24
<document identifier>	

Table of Contents

AuctionArchitects	Version: 2.0
Software Requirements Specification	Date: <12/11/24
<document identifier>	

Software Requirements Specification

1. Introduction

1.1 Purpose

The purpose of this Software Requirements Specification (SRS) is to provide a detailed and comprehensive outline of the functionalities, user roles, system behaviors, and security measures of Auction Architect, a secure online marketplace for car auctions. This e-bidding platform will allow users to list, view, bid on, and purchase cars in a streamlined and user-friendly environment. Auction Architect aims to provide a marketplace that caters to various user types, such as Visitors, Registered Users, VIP Members, and Super-Users (administrators). Each user type will have access to different features and functionalities depending on their role, allowing them to engage in transactions, manage listings, and, for super-users, administer the platform to ensure security and quality standards.

The system also enables users to access in-depth information on each vehicle, such as:

- Vehicle History: Previous ownership, accident records, maintenance logs, etc.
- Current Condition: Condition reports, damage details, and wear-and-tear specifics.
- Fair Market Value: Estimated value based on data from similar cars, market trends, and the car's condition.

Additionally, the platform will implement robust security protocols to protect personal and financial information, ensuring users can transact confidently. By setting clear requirements for each function and user interaction, this document will guide the development process to ensure that Auction Architect meets the needs of all stakeholders and provides a reliable, secure car auction experience.

1.2 Scope

The Auction Architect platform is designed to facilitate secure and user-friendly car auctions with diverse features tailored for different user roles. The system will support four primary user types, each with specific permissions and capabilities:

- Visitor (V): Browses the site and views car listings but cannot participate in bidding or purchases.
- User (U): A registered user with the ability to create listings, place bids, participate in transactions, and leave ratings and reviews.
- VIP Member (VIP): A high-status user who gains additional privileges, including exclusive access to premium car listings, transaction discounts, and complete vehicle history reports.
- Super-User (S): An administrator with enhanced privileges, such as the ability to approve or deny user verifications, manage listings, handle complaints, and monitor system security.

Key features of Auction Architect include:

- Listing Management: Users can list cars for sale by providing details, setting a reserve price, and uploading images.
- Bidding System: Registered users can place bids on cars, with real-time updates to ensure transparency and fair competition.
- Secure Transactions: The system will integrate with a payment gateway to handle transactions securely, using encryption to protect sensitive information.
- Rating and Feedback System: Users can rate each other post-transaction to foster trust and accountability within the community.
- VIP Privileges: VIP members receive special privileges, such as exclusive access to luxury vehicles and a discount on transaction fees, making the platform more appealing for high-value users.
- Detailed Vehicle Information: Integrations with external APIs, like CarFax, will provide comprehensive reports on each vehicle's history and condition.

The platform's requirements are set to ensure seamless interactions, robust security, and high performance, providing users with a reliable and engaging car auction experience.

1.3 Definitions, Acronyms, and Abbreviations

- V: Visitor – A non-registered user who can view listings but cannot interact with them.
- U: User – A regular, registered user with full access to the bidding system.
- VIP: High-status user with enhanced access to premium listings, discounts, and full vehicle history.
- S: Super-user – Administrator with extended privileges for managing the platform.
- SRS: Software Requirements Specification – A document detailing system requirements.
- GUI: Graphical User Interface – The visual component through which users interact with the system.
- API: Application Programming Interface – External data sources like CarFax for vehicle history.
- 2FA: Two-Factor Authentication – Security measure requiring two forms of user verification.
- SSL/TLS: Secure Sockets Layer / Transport Layer Security – Protocols for encrypted data transmission.

1.4 References

- 2019 Acura TLX Tech A-SPEC PKGS." IAAI.Com, www.iaai.com/VehicleDetail/40848380~US. Accessed 16 Oct. 2024.
- CarFax Vehicle History API Documentation – Provides vehicle history and report dat

1.5 Overview

This SRS document specifies the functional and non-functional requirements of the Auction Architect platform. It outlines the roles and permissions for each user type, the use cases, and system behaviors, including listing, bidding, transactions, and administrative features. The document also details system requirements like

security, performance, and usability, ensuring the platform is scalable, secure, and user-friendly. The remaining sections provide specific details on each use case, supplementary requirements, and supporting information, facilitating a robust framework for design and development.

2. Overall Description

2.1 Use-Case Model Survey

The Auction Architect platform supports a range of primary use cases designed to meet user needs effectively:

1. Browsing: Visitors and registered users can browse car listings, view detailed vehicle information (e.g., make, model, year, condition, reserve price), and access photos. VIP users have access to premium listings with luxury vehicles.
2. Bidding: Registered users can place bids on cars listed for auction. The bidding system updates in real-time, showing the highest bid and notifying users of overbids. The bidding feature will include:
    a. Automatic Bidding: Users can set a maximum bid amount, and the system will automatically bid on their behalf up to this limit.
    b. Reserve Price: A minimum threshold set by the seller that must be reached for the sale to proceed.
3. Transaction Handling: When the auction ends, if the highest bid meets or exceeds the reserve price, the system will process the transaction. This includes:
    a. Payment Processing: Integration with a payment gateway to securely handle transaction amounts.
    b. Seller and Buyer Notifications: Both parties are notified of the transaction's success and receive transaction details.
4. Rating and Feedback: After a successful transaction, buyers and sellers can rate each other and leave feedback, contributing to a transparent community and trust-building. Ratings are visible on user profiles.
5. VIP Privileges: VIP users gain additional benefits such as:
    a. Access to high-value listings with luxury vehicles (e.g., Lamborghini, Porsche).
    b. Discounts on transaction fees and exclusive access to in-depth vehicle histories.
    c. Priority bidding on premium vehicles.

2.2 Assumptions and Dependencies

1. Secure Payment Gateway: All financial transactions will be processed through a secure, encrypted payment gateway, with support for major payment methods.
2. Account Verification: Users must verify their accounts by providing identification, such as a driver's license and banking information, to prevent fraud.
3. Data Security: Sensitive information (personal and financial) will be encrypted using SSL/TLS to protect data integrity and privacy.
4. API Integration for Vehicle History: The system will depend on reliable external APIs, like CarFax, to retrieve accurate vehicle history reports.
5. Internet Dependence: The platform requires stable internet connectivity for real-time updates, external API requests, and secure transactions.

3. Specific Requirements

3.1 Use-Case Reports

Visitor (V): Can browse available car listings, view details of each car, and apply to become a registered user by completing a human-verification question.

User (U): Has full access to the bidding system, can list vehicles, place bids, view bid history, participate in transactions, rate other users post-transaction, and leave comments on listings. Users can request to become verified sellers by providing additional documents.

VIP User: VIP status is granted to users with a balance exceeding $5,000 or those who have completed multiple successful transactions. VIP users enjoy benefits such as:

- Discounts on transaction fees.
- Access to premium listings and detailed vehicle history.
- Early or exclusive access to specific auctions for high-value vehicles.

Super-User (S): Administrators with enhanced privileges, such as:

- Approving or rejecting new user verifications.
- Managing complaints, handling account suspensions, and responding to user inquiries.
- Monitoring flagged listings, resolving disputes, and canceling fraudulent or problematic transactions.

3.2 Supplementary Requirements

1. Security:
   - User Authentication: Two-factor authentication (2FA) will be implemented to ensure user account security.
   - Encryption: SSL/TLS encryption will secure all data transmissions.
   - Audit Logs: Actions such as account creation, listing updates, and bid placements will be logged for security auditing and traceability.
2. Performance:
   - Scalability: The system should handle thousands of concurrent users during peak times, such as popular listing auctions.
   - Response Time: Actions such as placing bids and updating listing details should have a response time under 1 second.
3. User Interface (UI):
   - Customizable Interface: The GUI should adapt to different user roles (visitor, user, VIP, and admin), providing relevant information and functionality for each type.
   - Notifications and Alerts: Users will receive real-time notifications on bidding updates, overbids, listing status, transaction status, and account changes.
4. System Integrity and Backup:
   - Data Backups: Regular backups will be implemented to safeguard against data loss.
   - Fraud Detection and Prevention: Automated monitoring of suspicious activity, such as repeated overbidding by the same account, will be implemented to protect users and ensure fair play.

4. Supporting Information

Index

Appendices

The appendices include additional supporting materials and clarify whether they are integral to the requirements.

Appendix A: Use-Case Storyboards

Detailed storyboards for key use cases, covering normal and exceptional interactions for scenarios such as browsing listings, bidding, selling, and account management.

Appendix B: User Interface Prototypes

Visual representations of main user interface screens, such as the homepage, listing detail pages, bidding pages, and user profile management screens.

Appendix C: Glossary of Terms

Definitions for key terms, acronyms, and abbreviations used throughout the SRS (e.g., API, SSL/TLS, VIP user, etc.), to aid understanding for both technical and non-technical stakeholders.

Appendix D: Security and Compliance References

Documentation of security standards and protocols (e.g., SSL/TLS encryption, two-factor authentication) that the platform adheres to, along with relevant compliance requirements.

Appendix E: Design References

Sources and design guidelines referenced for UI/UX design, performance standards, and external integrations (e.g., third-party APIs, data sources like CarFax for vehicle history).

Appendix F: Testing Procedures and Results

Details of test cases, methodologies, and results from functional, performance, and security testing, ensuring the platform meets all specified requirements.

Appendix G: Entity-Relationship (ER) Diagrams

Visual representations of the data model, showing the relationships between different entities (e.g., users, listings, bids) within the platform.

Appendix H: System Architecture and Class Diagrams

Technical diagrams illustrating the overall architecture, major system components, and interactions among classes/modules. Useful for developers to understand the design and flow of the platform.

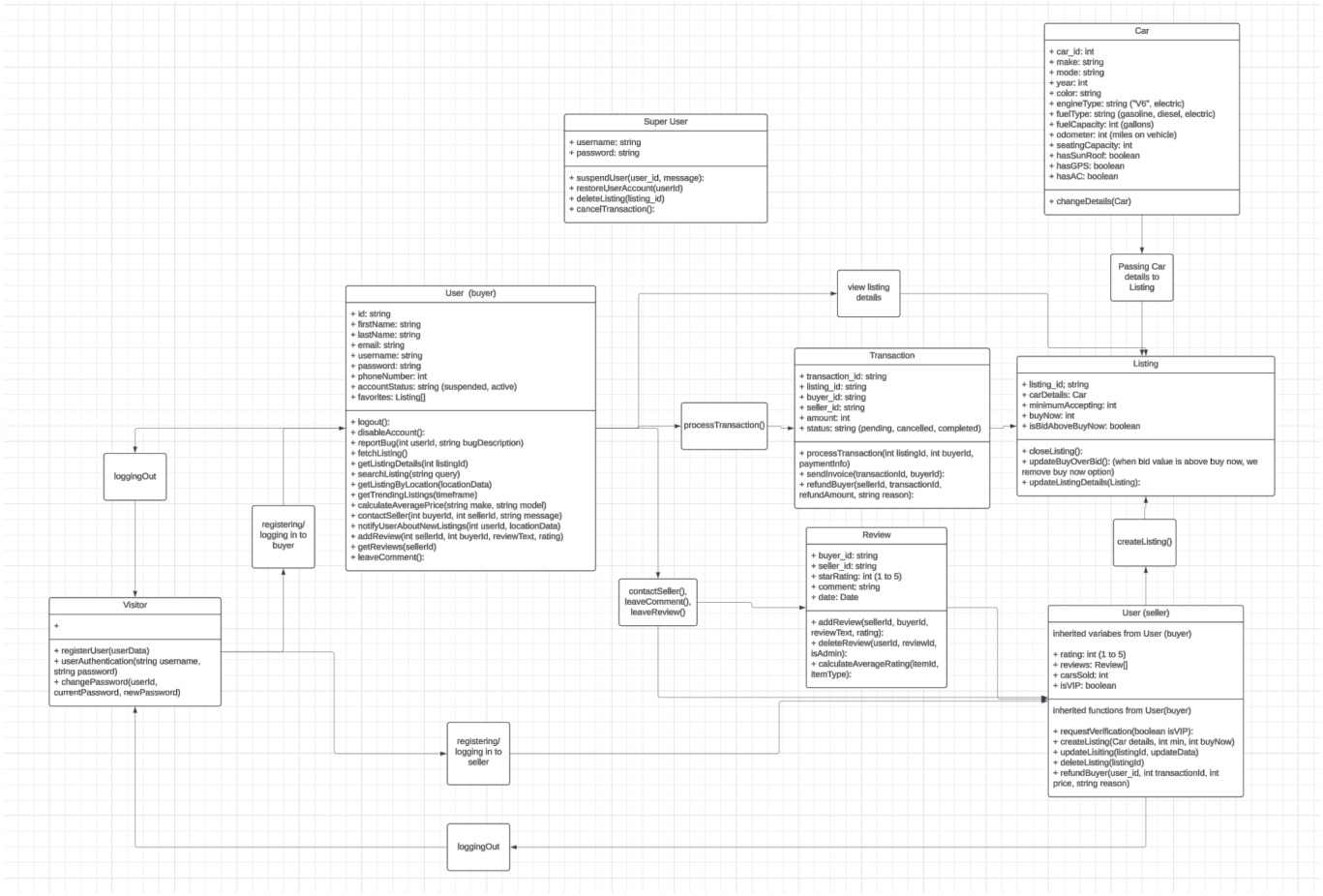
Appendix I: Meeting Memos and Decisions Log

Documentation of group meetings, key decisions made, and any significant changes in project direction or requirements. This can help maintain transparency and record decisions.

Appendix J: API Documentation

Documentation for any internal or external APIs used by the platform, including endpoints, parameters, and sample requests/responses. This is valuable for developers integrating or extending platform features.

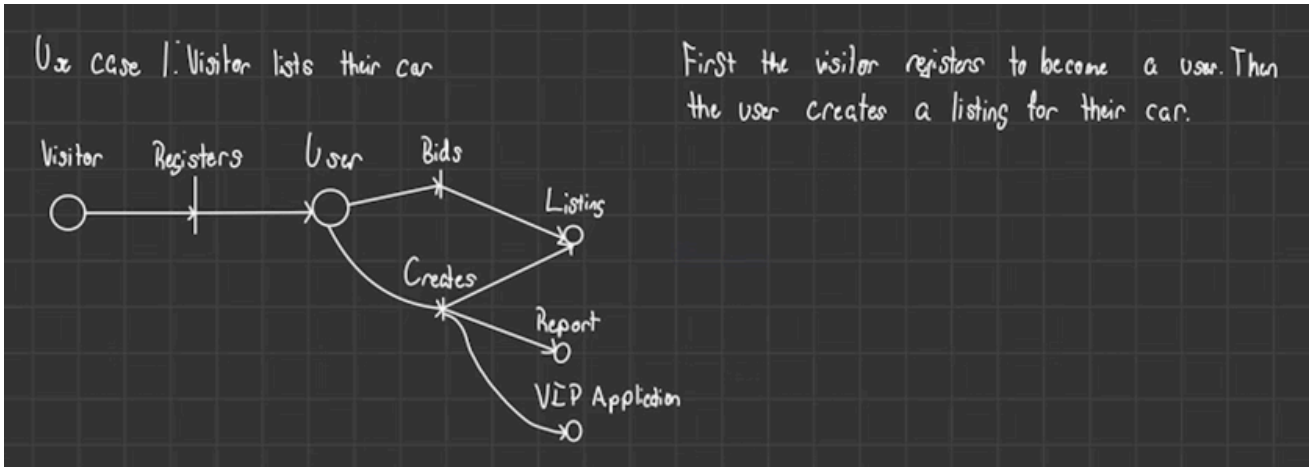
5. Collaboration Class Diagram

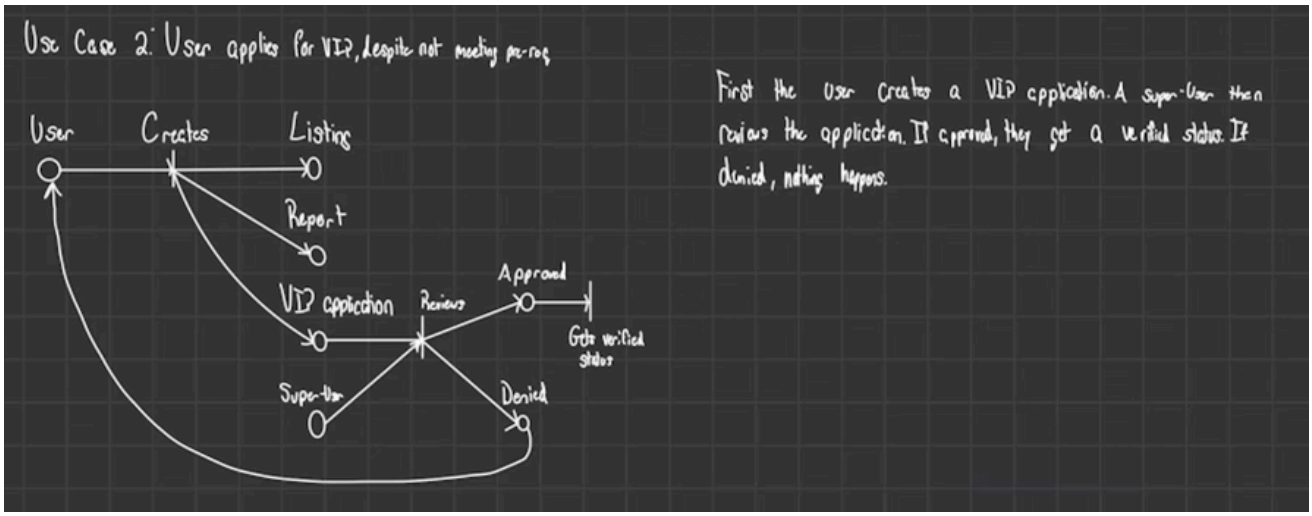


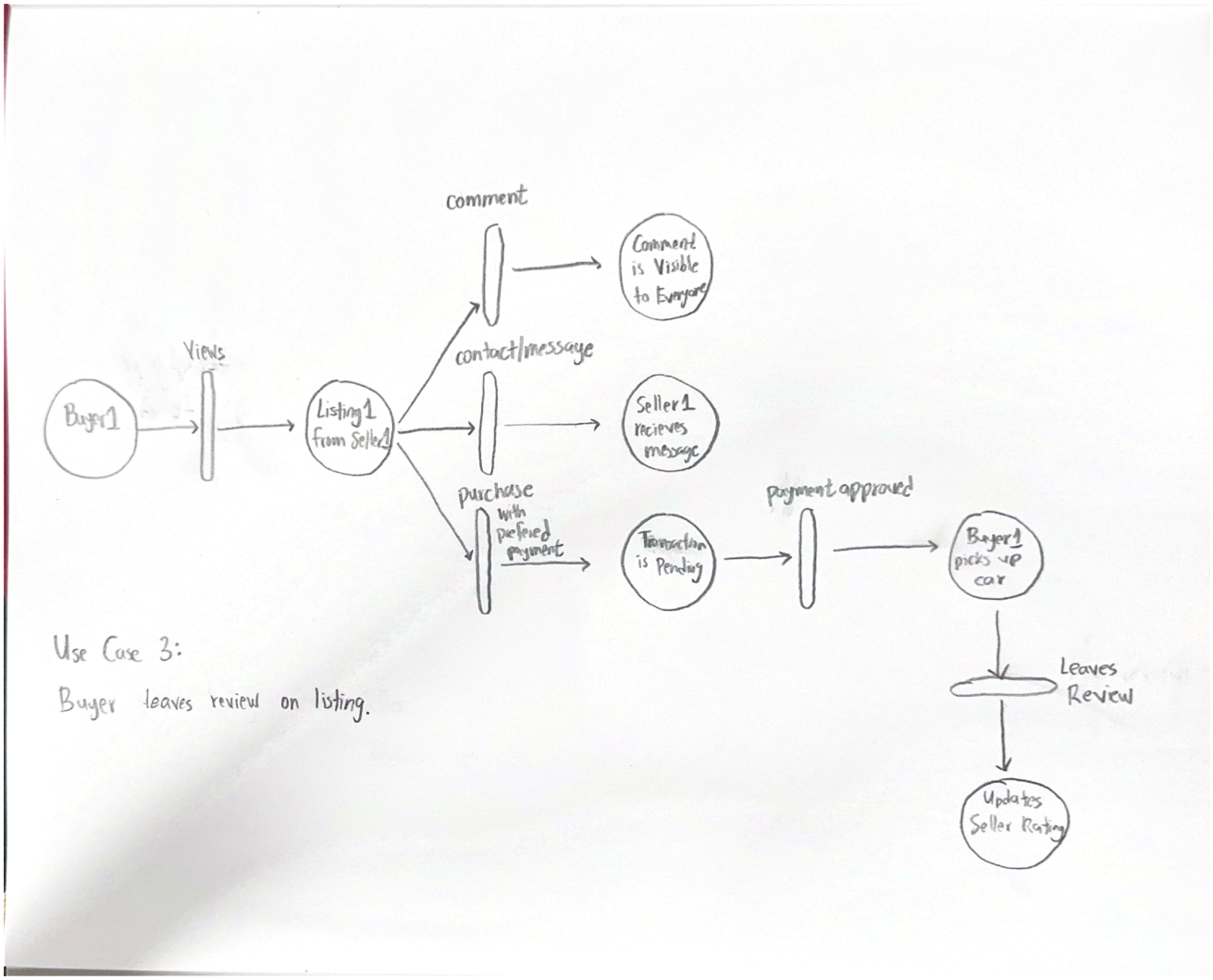
6. All Use Cases

6.1 Scenarios for each use case: normal AND exceptional scenarios

6.2 Collaboration or sequence class diagram for each use case, choose 3 or more use cases: draw the Petri-nets instead of class diagrams



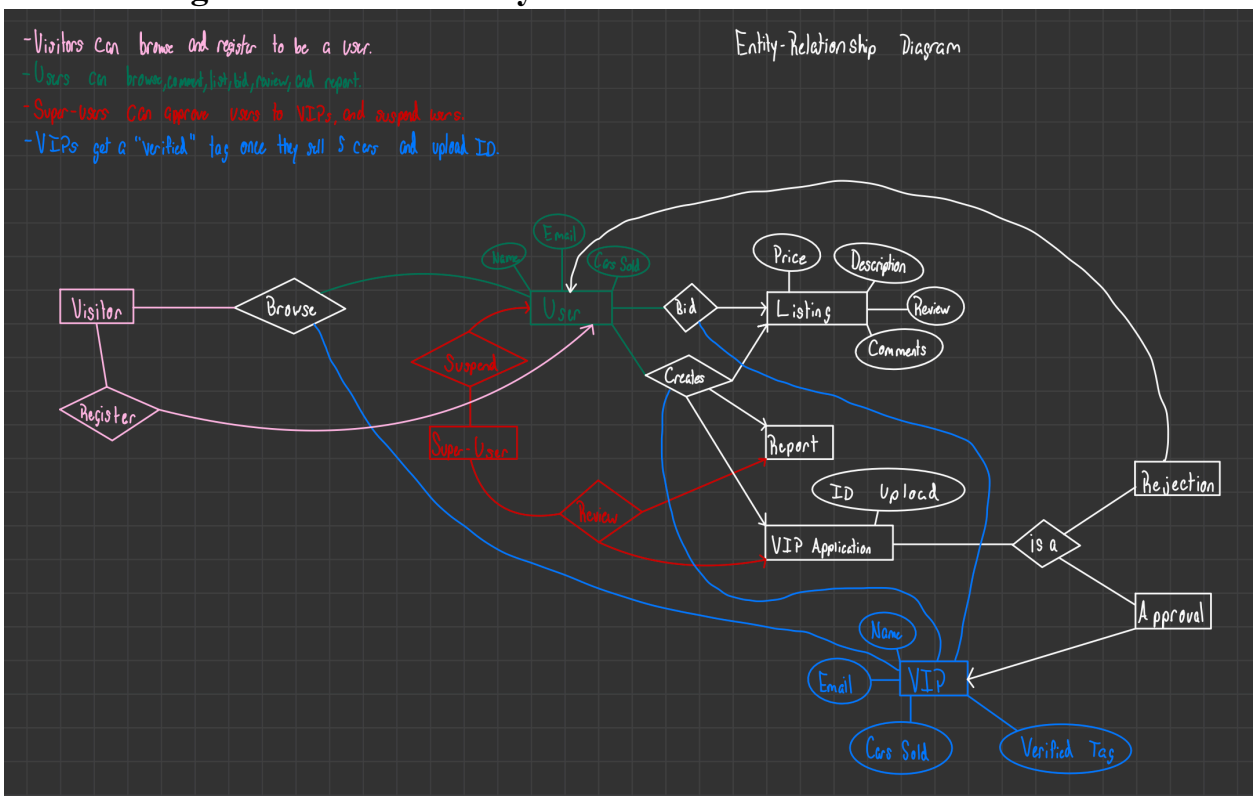




Use Case 3:

Buyer leaves review on listing.

7. E-R diagram for the entire system



8. Detailed Design using pseudo-code

User and Authentication Functions

logout - Ends the user's session by invalidating their authentication token.

```
Method: logout
Input: userId, authToken
Output: Success or error message

BEGIN logout(userId, authToken)
   Retrieve session associated with authToken
   IF session is valid AND belongs to userId
      Invalidate authToken (remove it from active sessions)
      Clear session data if applicable
      RETURN success message (logout successful)
   ELSE
      RETURN error message (invalid session or already logged out)
   END IF
END
```

registerUser - Registers a new user on the platform.

```
Method: registerUser
Input: userData (object with user details like name, email, password)
Output: Success message or error message`

BEGIN registerUser(userData)
   Validate userData (check email format, password strength, etc.)
   Check if email already exists in database
   IF email is unique
      Hash the password
      Insert new user record into the database
      RETURN success message
   ELSE
      RETURN error message (email already registered)
   END IF
END
```

userAuthentication - Authenticates a user for login.

```
Method: userAuthentication
Input: email (string), password (string)
Output: Authentication token or error message

BEGIN userAuthentication(email, password)
   Retrieve user record from database where email matches
   IF user exists
      Compare provided password with stored password hash
      IF passwords match
         Generate authentication token
         RETURN token
      ELSE
         RETURN error message (incorrect password)
      END IF
   ELSE
      RETURN error message (user not found)
   END IF
END
```

changePassword - Allows a user to change their password

```
Method: changePassword
Input: userId, currentPassword, newPassword
Output: Success or error message

BEGIN changePassword(userId, currentPassword, newPassword)
   Retrieve user by userId
   IF current password matches
      Hash newPassword
      Update password in database
      RETURN success message
   ELSE
      RETURN error (incorrect current password)
```

```
    END IF
END
```

verifySeller - Verifies a seller's account for selling privileges.

```
Method: verifySeller
Input: userId, verificationDocuments
Output: Success or error message

BEGIN verifySeller(userId, verificationDocuments)
    Validate verificationDocuments
    Update user status to "verified seller"
    RETURN success message
END
```

requestVerification - Allows a user to submit a verification request.

```
Method: requestVerification
Input: userId, verificationDocuments (list of documents), verificationDetails (object with additional
information)
Output: Success or error message

BEGIN requestVerification(userId, verificationDocuments, verificationDetails)
    Retrieve user by userId
    IF user exists AND user is not already verified
        Validate verificationDocuments and verificationDetails (e.g., file format, required fields)
        IF validation is successful
            Store verification request in database with userId, documents, and details
            Update user status to "verification pending"
            Notify admin team of new verification request
            RETURN success message (verification request submitted)
        ELSE
            RETURN error message (invalid documents or details)
        END IF
    ELSE
        RETURN error message (user already verified or not found)
    END IF
END
```

addToFavorites - Adds a listing to a user's favorites.

```
Method: addToFavorites
Input: userId, listingId
Output: Success or error message

BEGIN addToFavorites(userId, listingId)
    IF listingId not in user's favorites
        Add listingId to user's favorites
        RETURN success message
    ELSE
        RETURN error (listing already in favorites)
```

```
    END IF
END
```

reportBug - Allows users to report a technical bug on the platform.

```
Method: reportBug
Input: userId, bugDescription
Output: Success or error message

BEGIN reportBug(userId, bugDescription)
    Record bug in bug tracking system
    RETURN success message
END
```

disableAccount - Allows a user to disable their account.

```
Method: disableAccount
Input: userId, confirmation (boolean)
Output: Success or error message

BEGIN disableAccount(userId, confirmation)
    IF confirmation is true
        Retrieve user by userId
        IF user account is active
            Update user status to "disabled"
            Log account disabling event with timestamp
            Notify user of successful account disablement
            RETURN success message (account disabled)
        ELSE
            RETURN error message (account already disabled or not found)
        END IF
    ELSE
        RETURN error message (confirmation required to disable account)
    END IF
END
```

Listing Management Functions

f**etchListings -** Retrieves a list of all car listings.

```
Method: fetchListings
Input: None
Output: Array of car listing objects

BEGIN fetchListings
    Initialize empty array `listings`
    Connect to database
    Execute query to retrieve all car listings
```

```
   FOR each listing in query results
      Add listing to `listings` array
   END FOR
   RETURN `listings`
END
```

getListingDetails - Retrieves detailed information for a specific listing.

```
Method: getListingDetails
Input: listingId (string or integer representing the ID of the car listing)
Output: Detailed car listing object

BEGIN getListingDetails(listingId)
   Connect to database
   Execute query to find listing with matching listingId
   IF listing is found
      RETURN listing object
   ELSE
      RETURN null (or error message)
   END IF
END
```

createListing - Allows users to create a new car listing.

```
Method: createListing
Input: listingData (object with details like make, model, year, price, description, images)
Output: Success message or error message

BEGIN createListing(listingData)
   Validate listingData (check required fields, data formats, etc.)
   IF listingData is valid
      Connect to database
      Execute query to insert new listing into database
      RETURN success message
   ELSE
      RETURN error message
   END IF
END
```

updateListing - Updates the details of an existing listing.

```
Method: updateListing
Input: listingId, updateData
Output: Success or error message

BEGIN updateListing(listingId, updateData)
   IF listing exists
      Update listing with updateData
      RETURN success message
   ELSE
      RETURN error (listing not found)
```

```
    END IF
END
```

deleteListing - Deletes a car listing from the platform.

```
Method: deleteListing
Input: listingId
Output: Success or error message

BEGIN deleteListing(listingId)
    Delete listing by listingId
    RETURN success message
END
```

closeListing - Closes a listing, marking it as inactive.

```
Method: closeListing
Input: userId, listingId, isAdmin (boolean)
Output: Success or error message

BEGIN closeListing(userId, listingId, isAdmin)
    Retrieve listing by listingId
    IF listing exists AND listing status is "active"
        IF listing owner is userId OR isAdmin is true
            Update listing status to "closed"
            Set closedTimestamp to current time
            Log closure action with userId or adminId
            Notify any active bidders or interested buyers of the closure (optional)
            RETURN success message (listing closed)
        ELSE
            RETURN error message (unauthorized closure attempt)
        END IF
    ELSE
        RETURN error message (listing not found or already closed)
    END IF
END
```

Search and Filtering Functions

searchListings - Searches for car listings using keywords.

```
Method: searchListings
Input: query (string of search keywords)
Output: Array of car listing objects matching search

BEGIN searchListings(query)
    Initialize empty array `searchResults`
    Connect to database
    Execute query to retrieve listings where title, description, or tags match `query`
    FOR each matching listing
        Add listing to `searchResults`
```

```
    END FOR
    RETURN `searchResults`
END
```

getListingsByLocation - Retrieves listings within a geographic location.

```
Method: getListingsByLocation
Input: locationData (city, state, zip code)
Output: List of listings within location

BEGIN getListingsByLocation(locationData)
    Execute query to retrieve listings based on locationData
    RETURN listings within specified location
END
```

getTrendingListings - Retrieves listings with the most views in a timeframe.

```
Method: getTrendingListings
Input: timeFrame (e.g., "24 hours")
Output: List of trending listings

BEGIN getTrendingListings(timeFrame)
    Retrieve listings with highest views within timeFrame
    RETURN trending listings
END
```

calculateAveragePrice - Calculates the average price for a specific make or model.

```
Method: calculateAveragePrice
Input: make, model
Output: averagePrice (float)

BEGIN calculateAveragePrice(make, model)
    Retrieve prices of cars with specified make and model
    Calculate and return average price
END
```

Transaction and Payment Functions

cancelTransaction - Allows an admin to cancel a transaction

```
Method: cancelTransaction
Input: adminId, transactionId, reason
Output: Success or error message

BEGIN cancelTransaction(adminId, transactionId, reason)
    Verify admin privileges for adminId
    Retrieve transaction by transactionId
    IF transaction exists AND transaction status is "completed" or "in progress"
        Update transaction status to "canceled"
```

```
      Log cancellation reason and adminId for tracking
      Notify both buyer and seller of the cancellation and reason
      IF payment was processed
         Initiate refund process (call refundBuyer function if applicable)
      RETURN success message (transaction canceled)
   ELSE
      RETURN error message (transaction not found or already canceled)
   END IF
END
```

refundBuyer - Allows a seller to initiate a refund for a buyer

```
Method: refundBuyer
Input: sellerId, transactionId, refundAmount, reason
Output: Success or error message

BEGIN refundBuyer(sellerId, transactionId, refundAmount, reason)
   Retrieve transaction by transactionId
   IF transaction exists AND transaction belongs to sellerId AND transaction status is "completed"
      Validate refundAmount does not exceed total transaction amount
      Update transaction status to "refund initiated"
      Record refund reason and refundAmount in transaction history
      Process refund to buyer's payment method (e.g., call payment API)
      Notify buyer of refund initiation and amount
      RETURN success message (refund initiated)
   ELSE
      RETURN error message (transaction not found or invalid refund amount)
   END IF
END
```

processTransaction - Processes a transaction for a car purchase.

```
Method: processTransaction
Input: listingId, buyerId, paymentInfo
Output: Success or error message

BEGIN processTransaction(listingId, buyerId, paymentInfo)
   Validate paymentInfo
   IF listing is available
      Deduct payment from buyer's account
      Update listing status to "sold"
      RETURN success message
   ELSE
      RETURN error (listing unavailable or payment failed)
   END IF
END
```

SendInvoice - Sends an invoice to the buyer after purchase.

```
Method: sendInvoice
Input: transactionId, buyerId
```

```
Output: Success or error message

BEGIN sendInvoice(transactionId, buyerId)
   Generate invoice with transaction details
   Send invoice to buyer
   RETURN success message
END
```

calculateTransactionFee - Calculates the platform fee for a transaction.

```
Method: calculateTransactionFee
Input: transactionAmount
Output: transactionFee (float)

BEGIN calculateTransactionFee(transactionAmount)
   Calculate fee as percentage of transactionAmount
   RETURN transactionFee
END
```

updateBuyOverbid - Updates the highest bid when a new bid is placed.

```
Method: updateBuyOverbid
Input: listingId, newBidAmount, buyerId
Output: Success or error message

BEGIN updateBuyOverbid(listingId, newBidAmount, buyerId)
   Retrieve listing by listingId
   IF listing exists AND listing status is "active"
      IF newBidAmount > listing's currentHighestBid
         Update listing's currentHighestBid to newBidAmount
         Update listing's highestBidder to buyerId
         Log bid update with timestamp
         Notify previous highest bidder of the overbid (optional)
         RETURN success message (highest bid updated)
      ELSE
         RETURN error message (bid too low)
      END IF
   ELSE
      RETURN error message (listing not found or not active)
   END IF
END
```

updateListingDetails - Allows a seller or admin to update details of an existing listing.

```
Method: updateListingDetails
Input: userId, listingId, updatedDetails (object with new details), isAdmin (boolean)
Output: Success or error message
```

```
BEGIN updateListingDetails(userId, listingId, updatedDetails, isAdmin)
   Retrieve listing by listingId
   IF listing exists AND listing status is "active"
      IF listing owner is userId OR isAdmin is true
         Validate updatedDetails (ensure fields have valid values)
         Update listing with new details from updatedDetails
         Log update action with userId or adminId and timestamp
         Notify watchers or interested users about updated listing (optional)
         RETURN success message (listing details updated)
      ELSE
         RETURN error message (unauthorized update attempt)
      END IF
   ELSE
      RETURN error message (listing not found or not active)
   END IF
END
```

Notification and Communication Functions

contactSeller - Allows a buyer to contact a seller directly.

```
Method: contactSeller
Input: buyerId, sellerId, message
Output: Success or error message

BEGIN contactSeller(buyerId, sellerId, message)
   Validate message content
   Send message to seller
   RETURN success message
END
```

notifyUserAboutNewListings - Notifies a user of new listings in their area

```
Method: notifyUserAboutNewListings
Input: userId, locationData
Output: Success or error message

BEGIN notifyUserAboutNewListings(userId, locationData)
   Retrieve new listings in locationData
   Send notification to user
   RETURN success message
END
```

Review and Feedback Functions

addReview - Allows buyers to leave a review for a seller.

```
Method: addReview
Input: sellerId, buyerId, reviewText, rating
Output: Success or error message

BEGIN addReview(sellerId, buyerId, reviewText, rating)
   Validate reviewText and rating (1-5)
   Add review to seller's profile
   Update seller's average rating
   RETURN success message
END
```

calculateAverageRating - Calculates the average rating for a listing or seller.

```
Method: calculateAverageRating
Input: itemId (listingId or sellerId), itemType ("listing" or "seller")
Output: averageRating (float) or error message

BEGIN calculateAverageRating(itemId, itemType)
   Retrieve all ratings for itemId based on itemType
   IF ratings exist AND count of ratings > 0
      Initialize sumOfRatings to 0
      FOR each rating in ratings
         Add rating to sumOfRatings
      END FOR
      Set averageRating to sumOfRatings divided by count of ratings
      RETURN averageRating
   ELSE
      RETURN error message (no ratings found)
   END IF
END
```

getReviews - Retrieves all reviews for a seller.

```
Method: getReviews
Input: sellerId
Output: Array of review objects

BEGIN getReviews(sellerId)
   Retrieve all reviews for sellerId from database
   RETURN array of reviews
END
```

deleteReview - Allows a user or an admin to delete a review from a listing.

```
Method: deleteReview
Input: userId, reviewId, isAdmin (boolean)
Output: Success or error message

BEGIN deleteReview(userId, reviewId, isAdmin)
```

```
  Retrieve review by reviewId
  IF review exists
    IF review was created by userId OR isAdmin is true
      Delete review from the database
      Log review deletion with userId or adminId and timestamp
      Notify listing owner of review deletion (optional)
      RETURN success message (review deleted)
    ELSE
      RETURN error message (unauthorized deletion attempt)
    END IF
  ELSE
    RETURN error message (review not found)
  END IF
END
```

leaveComment - Allows a user to leave a comment on a listing or item.

```
Method: leaveComment
Input: userId, listingId, commentText
Output: Success or error message

BEGIN leaveComment(userId, listingId, commentText)
  Validate commentText (check length, prohibited words, etc.)
  Retrieve listing by listingId
  IF listing exists AND commentText is valid
    Create new comment record with userId, listingId, commentText, and timestamp
    Save comment to listing's comment section in the database
    Notify listing owner of new comment (optional)
    RETURN success message (comment posted)
  ELSE
    RETURN error message (invalid comment or listing not found)
  END IF
END
```

Admin and Security Functions

suspendUser - Bans a user from the platform for violations.

```
Method: suspendUser
Input: userId
Output: Success message or error message

BEGIN suspendUser(userId)
  Update user status to "suspended"
  Log reason for suspension
  RETURN success message
```

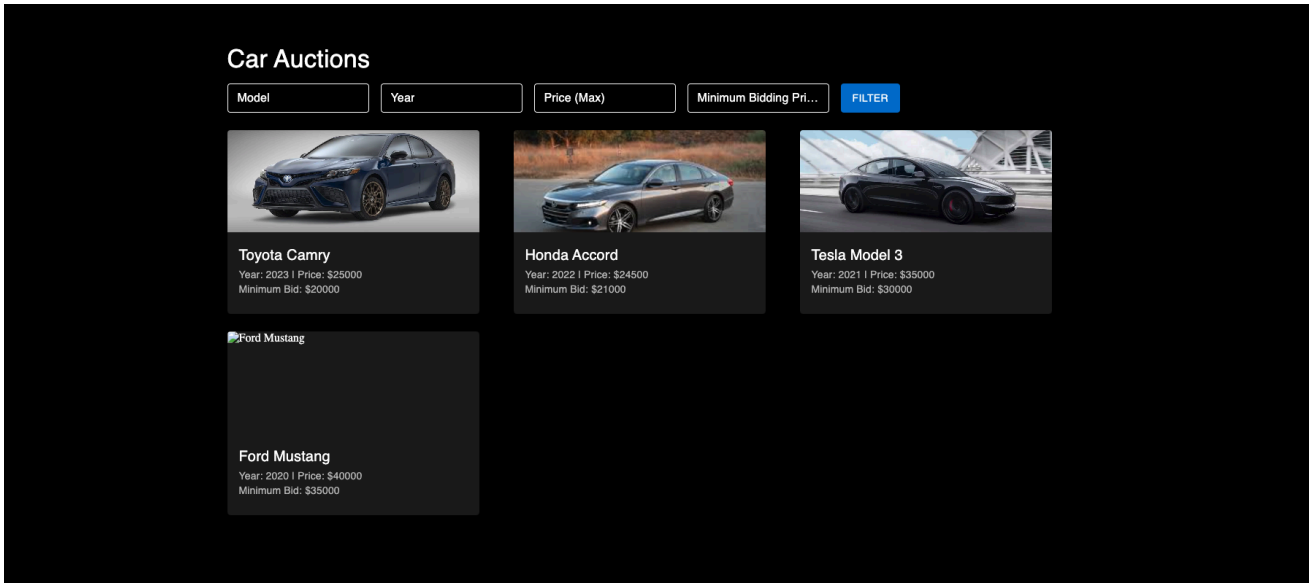
```
END
```

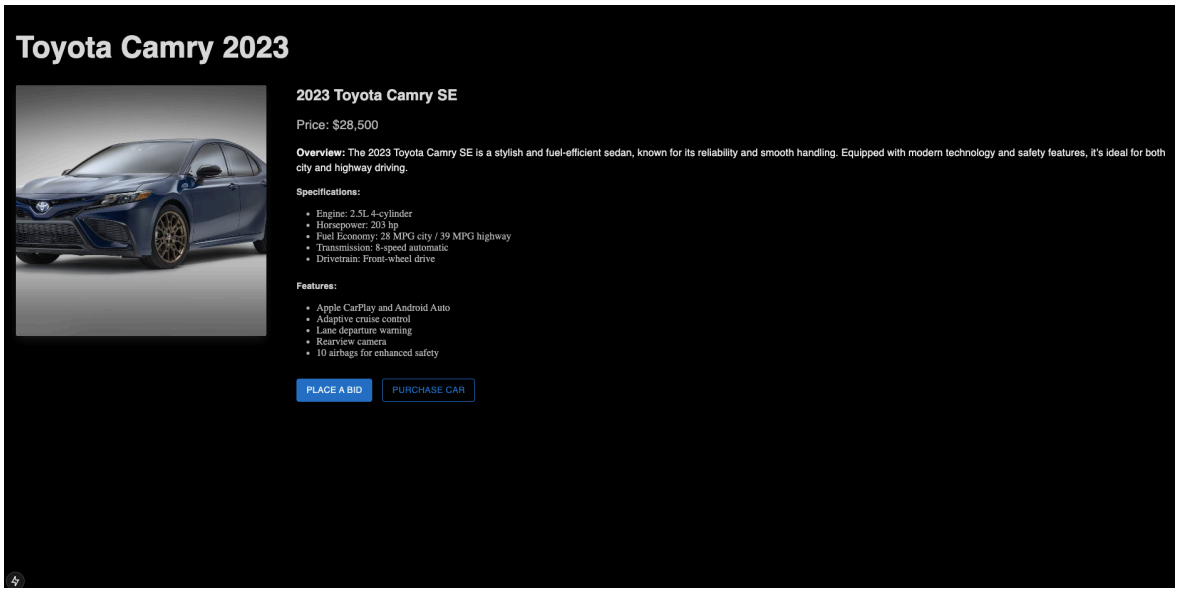
restoreUserAccount - Restores a suspended user's account.

```
Method: restoreUserAccount
Input: userId
Output: Success or error message

BEGIN restoreUserAccount(userId)
    Update user status to "active"
    RETURN success message
END
```

9. System screens: Major GUI screens and sample prototype


Car Auctions
Model
Year
Price (Max)
Minimum Bidding Pri…
FILTER
Toyota Camry
Year: 2023 | Price: $25000
Minimum Bid: $20000
Honda Accord
Year: 2022 | Price: $24500
Minimum Bid: $21000
Tesla Model 3
Year: 2021 | Price: $35000
Minimum Bid: $30000
Ford Mustang
Ford Mustang
Year: 2020 | Price: $40000
Minimum Bid: $35000


Toyota Camry 2023
2023 Toyota Camry SE
Price: $28,500
Overview: The 2023 Toyota Camry SE is a stylish and fuel-efficient sedan, known for its reliability and smooth handling. Equipped with modern technology and safety features, it's ideal for both city and highway driving.
Specifications:
Engine: 2.5L 4-cylinder
Horsepower: 203 hp
Fuel Economy: 28 MPG city / 39 MPG highway
Transmission: 8-speed automatic
Drivetrain: Front-wheel drive
Features:
Apple CarPlay and Android Auto
Adaptive cruise control
Lane departure warning
Rearview camera
10 airbags for enhanced safety
PLACE A BID
PURCHASE CAR

Sell Your Car

Name *

Address *

Phone Number *

VIN Number *

Minimum Accepting Bid *

Buy Now Price *

Car Description *

UPLOAD CAR PICTURES

SUBMIT CAR FOR AUCTION

Profile

R

Rafid Rahman
john.doe@example.com
123-456-7890
1234 Elm Street, Springfield, USA

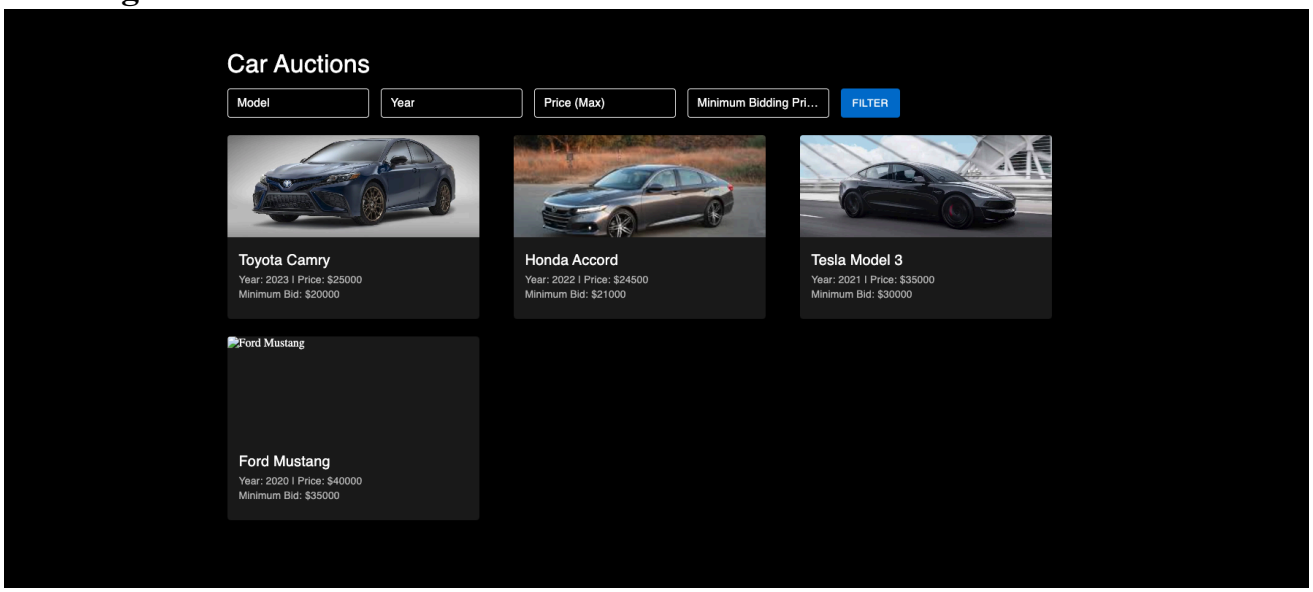
Cars Marked as Interested

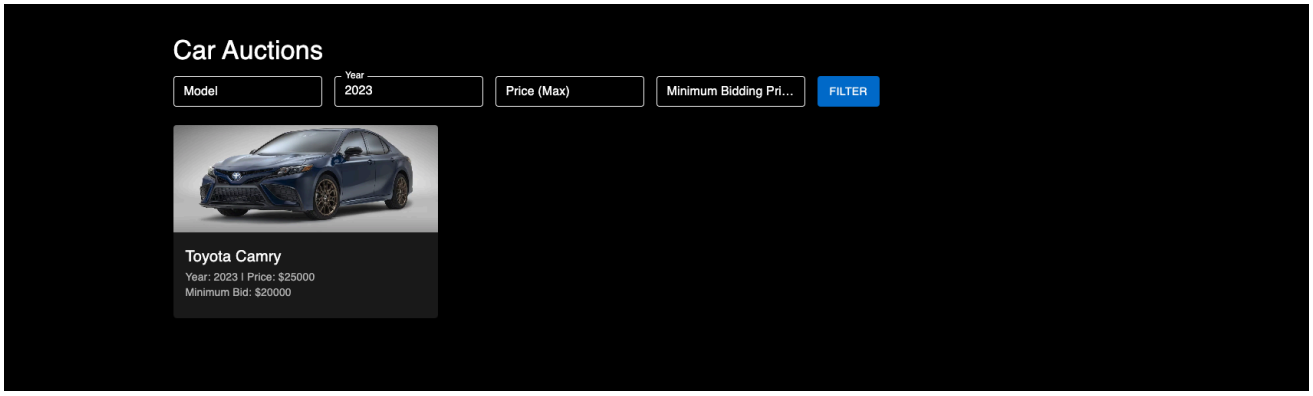
Toyota Camry 2023
$25,000

Honda Accord 2022
$24,500

Tesla Model 3
$35,000

Filtering Function Screenshots





Basic Filtering Function which filters the cars based on the filter

10. Memos of group meetings and possible concerns

Week of 11/13 - Initial Setup and Planning

Meeting Memo: 11/15 - User Authentication and Database Structuring

- **Agenda**:
  - Implement user authentication (JWT setup).
  - Define data models for users, listings, and bids.
- **Actions**:
  - Backend programmer to implement authentication, ensuring secure handling of user roles.
  - UI/UX team to sketch initial interfaces based on user types.
- **Concerns**:
  - Potential misalignment between authentication functionality and frontend design.
  - Security considerations need to be thoroughly discussed, as handling sensitive information is a top priority.

Week of 11/20 - Core Feature Development

Meeting Memo: 11/20 - Seller and Buyer Core Functionalities

- **Agenda**:
  - Develop listing functions (createListing, updateListing) for sellers.
  - Implement browsing (browseListings, getListingDetails) for buyers.
- **Actions**:
  - Backend programmer to start listing and bid functionalities.
  - UI/UX designers to build listing interface with real-time updates.
- **Concerns**:
  - High dependency between frontend and backend; delays in listing functionality may impact testing schedules.
  - Teamwork strain if designs don't align with backend capabilities, potentially requiring more meetings to clarify.

Meeting Memo: 11/23 - Bidding System and Real-Time Updates

- **Agenda**:
  - Implement placeBid and trackBids functions for buyers.
  - Integrate bid tracking into the frontend for real-time updates.
- **Actions**:
  - Backend programmer to ensure bid updates reflect immediately for all users.
  - JavaScript programmer to manage interactions between frontend and backend for smooth real-time bidding.
- **Concerns**:
  - Managing real-time data can be complex; communication is crucial to avoid delays in bid tracking functionality.
  - Testing for high-traffic scenarios may reveal performance issues that could require additional time.

Week of 11/27 - Transactions and Security

Meeting Memo: 11/27 - Secure Payment Processing

- **Agenda**:
  - Implement secure payment processing within the processTransaction function.
  - Configure transaction tracking and reporting for sellers and buyers.
- **Actions**:
  - Backend programmer to integrate payment gateway (Stripe or PayPal).
  - JavaScript programmer to test transaction flow with mock data for smooth checkout.
- **Concerns**:
  - Payment gateway integration complexity could delay timeline.
  - Security vulnerabilities may arise if encryption is not correctly implemented, requiring extra review sessions.

Meeting Memo: 11/30 - Admin Features and Marketplace Management

- **Agenda**:
  - Begin development of admin functions (suspendUserAccount, removeListing).
  - Discuss handling complaints and creating reports for admin users.
- **Actions**:
  - Backend programmer to set up suspendUserAccount and complaint tracking.
  - UI/UX team to design the admin dashboard for efficient management.
- **Concerns**:
  - Admin functions are highly sensitive; inadequate testing may result in user mismanagement.
  - Balancing user needs and admin oversight may introduce technical and ethical considerations.

Week of 12/4 - Testing and Refinement

Meeting Memo: 12/4 - End-to-End Testing

- **Agenda**:
  - Conduct end-to-end testing on core features: listings, bidding, transactions.
  - Identify and document any critical bugs for quick resolution.
- **Actions**:
  - Backend programmer to validate data integrity in the database during transactions.
  - UI/UX designers to collect user feedback on the overall interface and flow.
- **Concerns**:
  - Bug fixes may require changes in the backend structure, causing potential timeline shifts.
  - Communication among team members is crucial to avoid duplicated work in debugging.

Meeting Memo: 12/7 - Performance Testing and User Feedback

- **Agenda**:
  - Test application performance under simulated high-traffic conditions.
  - Gather feedback on the usability and adjust accordingly.
- **Actions**:
  - JavaScript programmer to run load tests and monitor server responses.
  - UI/UX team to adjust designs based on usability testing feedback.
- **Concerns**:
  - Delays in performance testing may disrupt final bug-fixing efforts.
  - Potential last-minute adjustments could increase tension if timeline extensions are required.

Week of 12/11 - Final Adjustments and Submission

Meeting Memo: 12/11 - Final Code Cleanup and Documentation

- **Agenda**:
    - Finalize code cleanup, documentation, and review of backend endpoints.
    - Prepare for project demo and submission.
- **Actions**:
    - Backend programmer to review all endpoints and validate error handling.
    - UI/UX team to polish interfaces and confirm navigation flow consistency.
- **Concerns**:
    - Last-minute technical difficulties could compromise demo quality.
    - Team stress may increase due to deadline pressure, making smooth communication essential.

11. Address of Git Repository

https://github.com/Shahed4/AuctionArchitects