Team J

QuickBid
Software Requirements Specification For
E-Bidding System

Version 1.0

Revision History

Date	Version	Description	Author
11/08/24	1.0	Initial Draft + Document Setup	Arihant, Azim, Rahat, Ivan, Tej
11/10/24	1.0	Added Use Cases + Petri Nets	Ivan
11/10/24	1.0	Added Collaboration Class + Pseudocode	Azim
11/10/24	1.0	Added Major GUI Screens + Pseudocode	Tej, Rahat
11/11/24	1.0	Added E-R Diagram + Pseudocode	Arihant

Table of Contents

1. Introduction 4

1.1. Purpose 4

1.2. Collaboration Class Diagram 4

2. All Use Cases 5-9

2.1. Scenarios 5

2.1.1. Visitor Applies to be User 5

2.1.2. User Checks Profile 6

2.1.3. User Deposit and Withdrawal 7

2.1.4. User Lists a Product 8

2.1.5. Users Makes a Bid 8

2.1.6. Conducting Transaction 9

2.1.7. Rate Anonymously 10

2.1.8. SuperUser - Handling Appeals Complaints, and Sign-ups 11

3. Entity Relation Diagram 12

4. Detailed Design 13- 23

4.1. Frontend Pseudocode 11-16

4.2. Backend Pseudocode 17-22

5. System Screens 23-27

5.1. Landing Page 23

5.2. Home Page 24

5.3. Login Page 25

5.4. Shopping Cart Page 26

5.5. Selected Item Page 27

6. **Group Meetings** 28-31

6.1. Meeting Logs 28-30

6.2. Work Distribution 30-31

7. Github Repository 31

Software Requirements Specification

1. Introduction

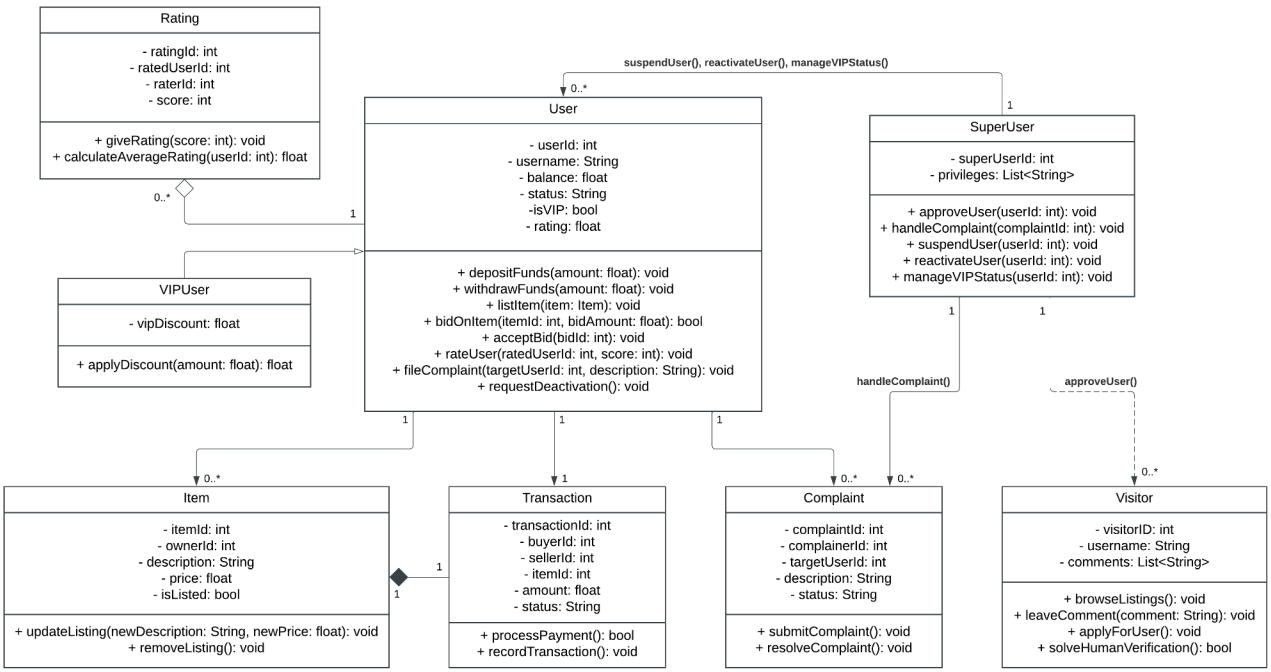
1.1 Purpose

This report serves as an overview of the structure behind our system. The collaboration class, Entity Relation, Use Case, and Petri-net diagrams outline the overview of our system and how different parts interact with each other. This report also provides pseudocode for relevant methods, detailing backend logic, as well as screenshots of our major GUI screens, with explanations of its layout and features. Lastly, the notes for our group meetings, how work was distributed for the development of our system, as well as the link to the GitHub repository will be covered in this report as well.

1.2 Collaboration Class Diagram

Link to Lucidchart Diagram

- https://lucid.app/lucidchart/44ebee19-b3e3-47ca-8802-75906bf00ed8/edit?invitationId=inv_a2760ad3-9214-4b84-ada9-a8f013ad8edc



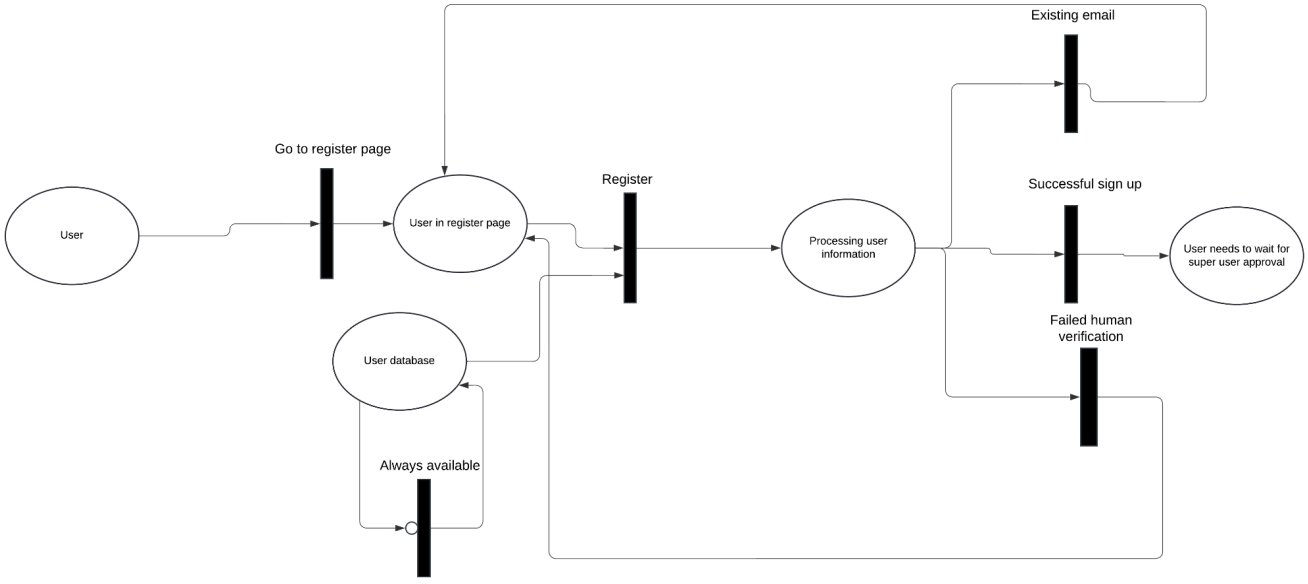
2. All Use Cases

Section 2.1 presents an overview of all use cases of our system, as well as a description for each scenario, with the corresponding petri-net. The lucidchart link to view all our petri nets is: https://lucid.app/lucidchart/b46e9c74-82de-4878-8ad9-5276f670feb6/edit?viewport_loc=-14371%2C-3699%2C26296%2C12931%2CHWEp-vi-RSFO&invitationId=inv_1ee7cffd-eb25-4abf-867a-0139b20d9565

2.1 Scenarios

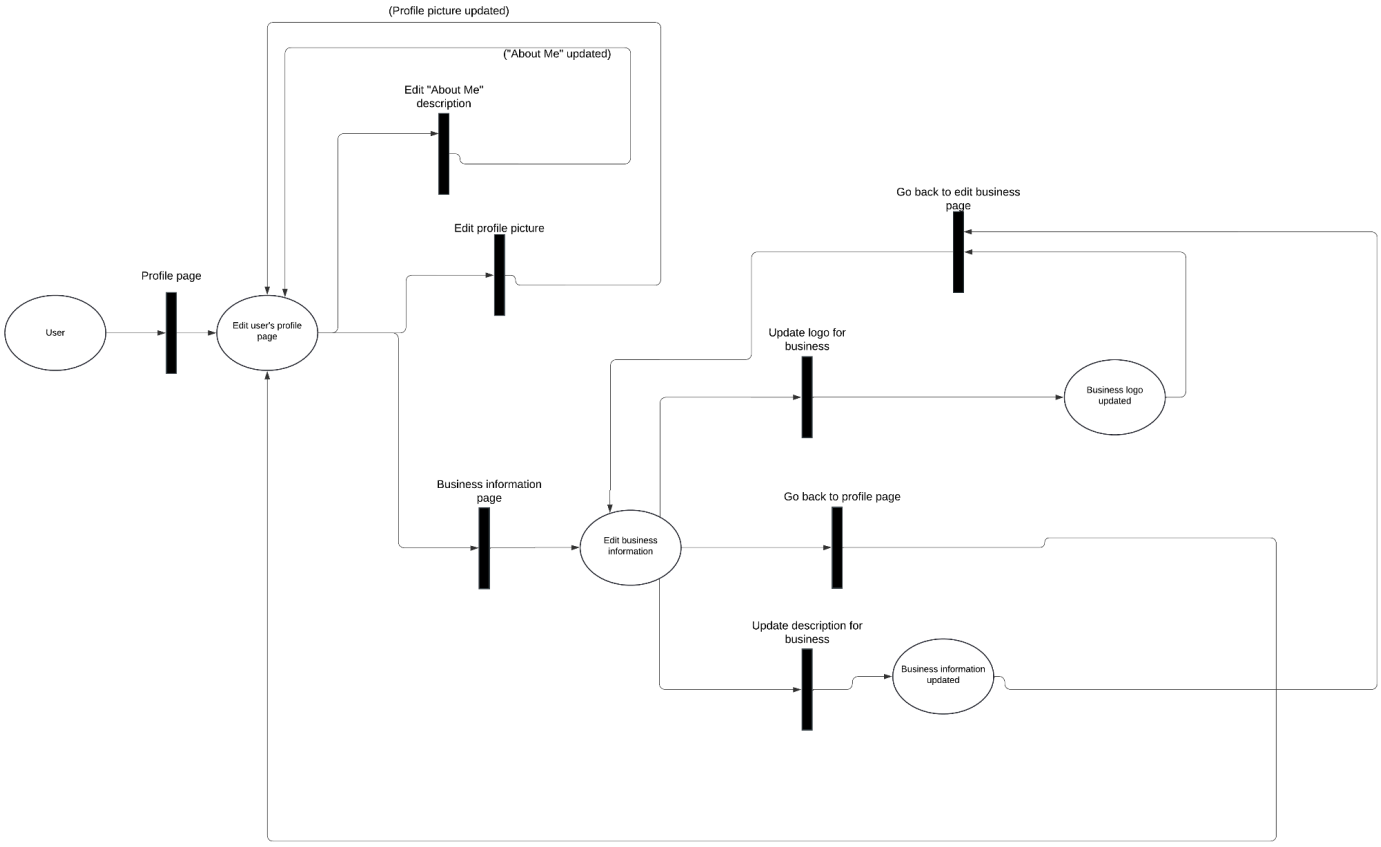
2.1.1 Visitor Applies to be User

- Normal Scenario: Visitors can apply to become users through our sign up functionality assuming that they don't have an existing account in our database, but this request needs approval from super users.
- Exceptional Scenario: Existing email when signing up which indicates an existing account associated with the visitor.



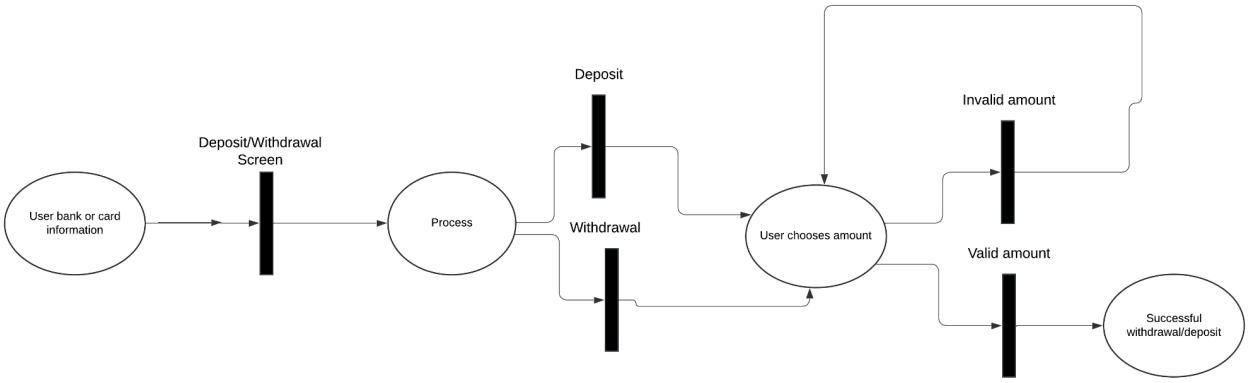
2.1.2 User Checks Profile

- Normal Scenario: Registered users are shown their profile screen containing previous records and personal details. This information may include “About Me” description, information related to their business, profile picture and phone number.
- Exceptional Scenario: NA



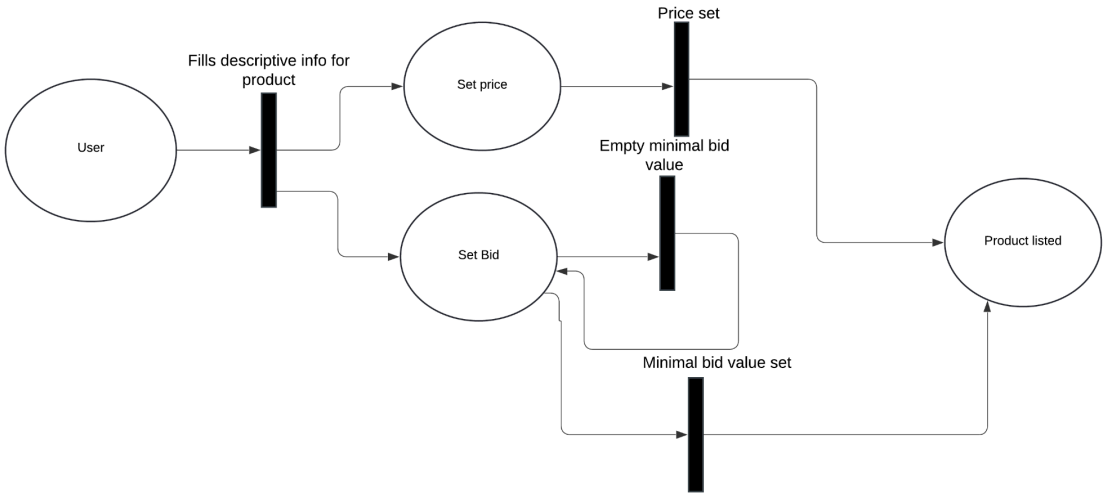
2.1.3 User Deposit and Withdrawal

- Normal Scenario: Registered users can deposit or withdraw funds to purchase products posted by other business owners/representatives.
- Exceptional Scenario: Insufficient funds are being withdrawn which prevents the user from doing so, therefore prompting the user to check their account balance.



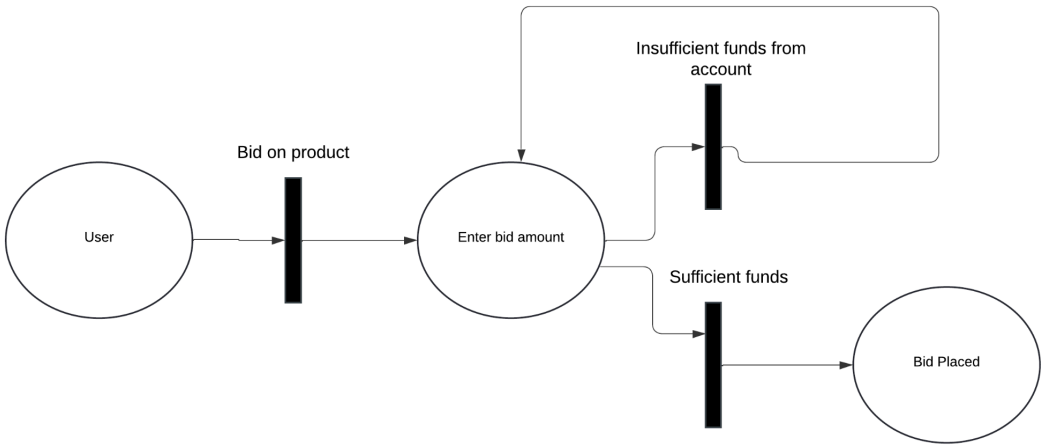
2.1.4 User Lists a Product

- Normal Scenario: Based on their preferences, users can list products either with a fixed price or by setting an initial bid amount that meets a minimum required value. Additionally, they can post requests for products they need, specifying a price range.
- Exceptional Scenario: Certain fields may need to be fulfilled before listing the product and missing requirements will prevent the listings from going live.



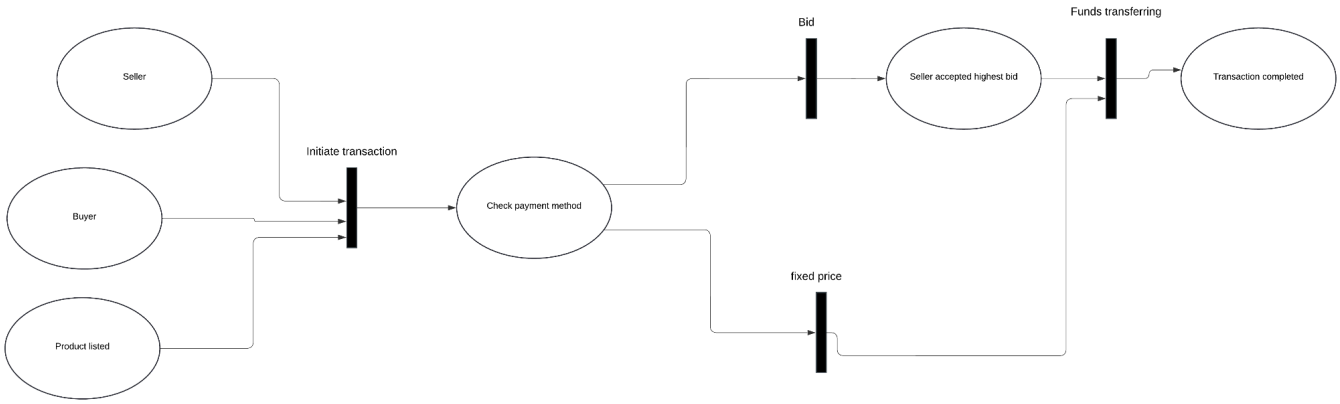
2.1.5 Users Makes a Bid

- Normal Scenario: Users can bid on other products listed but need sufficient funds depending on the existing value of the highest bid.
- Exceptional Scenario: User doesn't have sufficient funds to bid on a product,which prevents the bid from being placed.



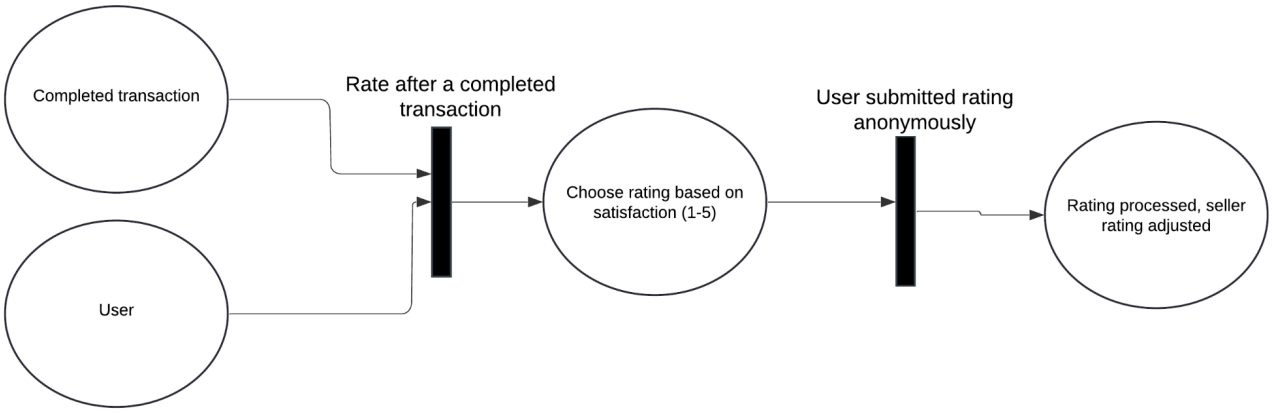
2.1.6 Conducting Transaction

- Normal Scenario: The transaction process varies based on the payment method set by the seller. If the payment method is a bid, the seller has the option to accept the current highest bid if they find it satisfactory. Upon acceptance, the funds are transferred. If the payment method is a fixed price, the transaction is automatically processed without requiring the seller's explicit approval, ensuring a seamless purchase experience.
- Exceptional Scenario: NA



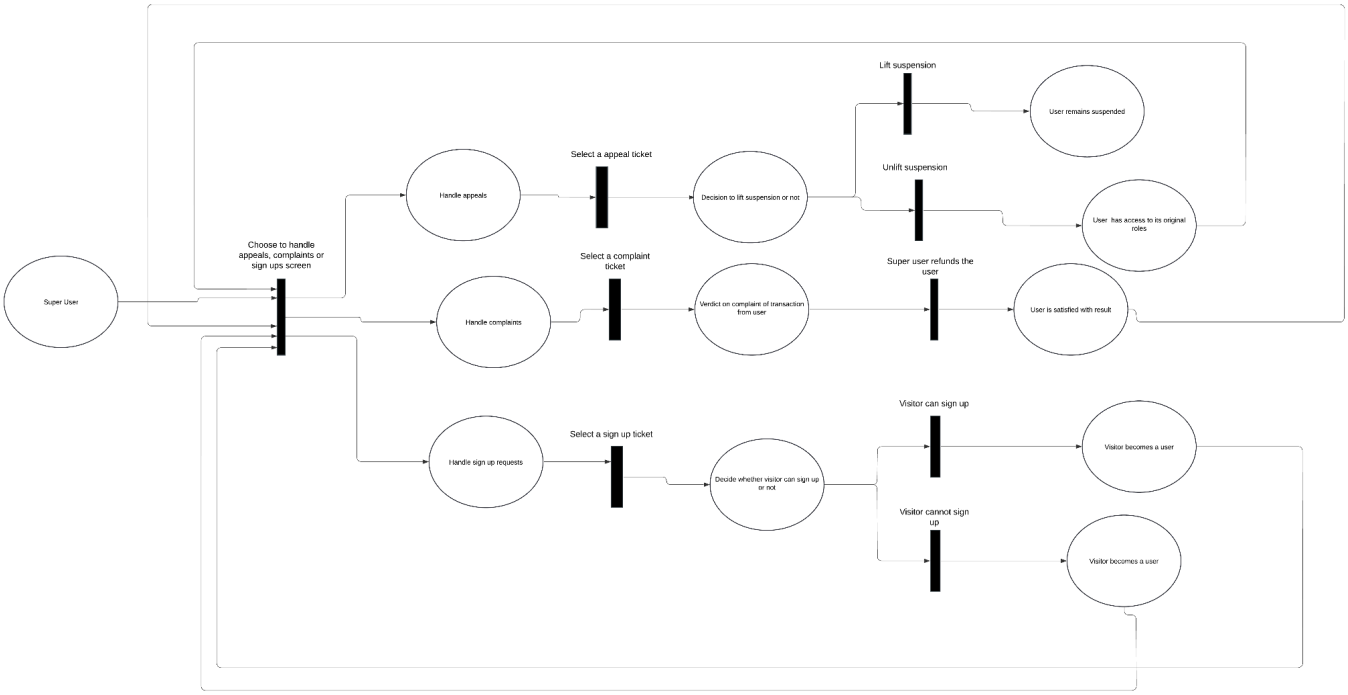
2.1.7 Rate Anonymously

- Normal Scenario: Participating users of completed transactions can rate anonymously.
- Exceptional Scenario: Participating user tries to submit multiple ratings that can ultimately affect the rating of the seller in a negative way, therefore only one submission is allowed.



2.1.8 SuperUser - Handling Appeals Complaints, and Sign-ups

- Normal Scenario: Super users have the ultimate verdict on complaints stemming from transactions between users, sign ups and appeals for suspension lift.
- Exceptional Scenarios: NA

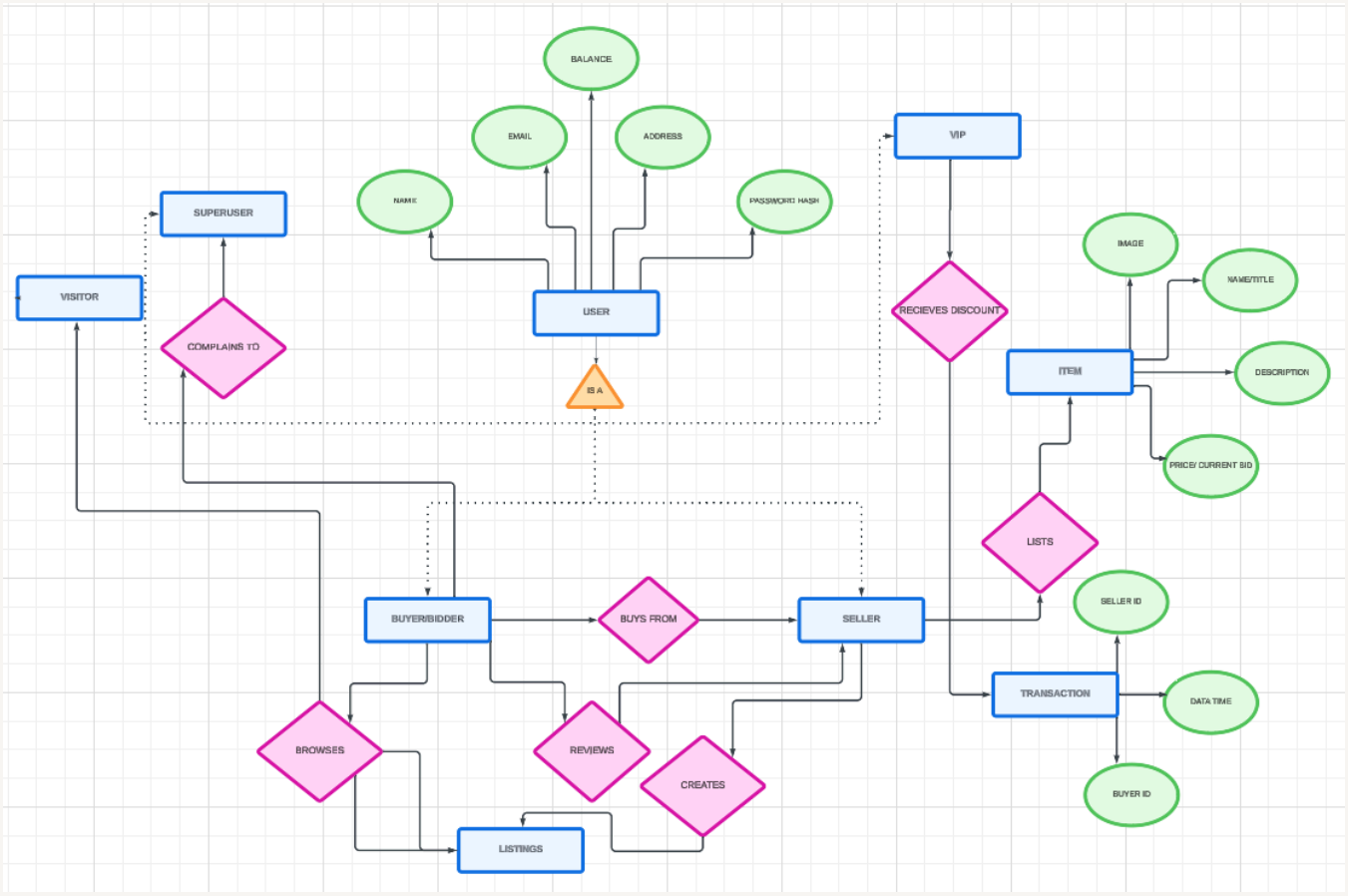


3. Entity Relation Diagram

The Entity Relation Diagram below outlines all functionalities of the system, displaying all entities, and their corresponding relationships and attributes.

- The Rectangles represent entities, which include the actors and different object types, These are also reflected in our database table.
- The Diamonds represent relationships between those entities. These relationships include a “has-a” relationship, as well as any actions taken by one entity onto another.
- Green ovals represent fields or attributes of the entity. Some of these will be the data points in our database.
- Orange triangles represent the inheritance relationships which are of “is-a” type. This is later depicted in database by the role of each user including Visitor, User, VIP, Superuser

The link to the ER-Diagram lucid chart is:
https://lucid.app/lucidchart/b7f1dcbe-ea8d-4b28-8dc5-c850f0d5e420/edit?invitationId=inv_21cb20ff-be86-4fb1-8aea-fcdcdcb15ddf:



4. Detailed Design

It was ultimately decided that we will use React TypeScript with Vite to build our frontend, Auth0 for authentication, along with external dependencies like Tailwind CSS and prebuilt components from UI libraries. By utilizing these modern building tools, we can significantly accelerate the development speed and provide high quality results that will gain client satisfactions. The components we will build will serve as reusable instances throughout the codebase, promoting consistency and a streamlined development process. Additionally, these components will seamlessly interact with our server, enabling data exchange while ensuring security measures that will protect user's data. By maintaining this structure, we are creating robust building blocks for future scalability and expansion.

Note: *This includes a subset of functions and pseudocode. It provides a selection of functions and concepts related to the login and signup processes. The full functionality of the application is still a work in progress and will be implemented in the final product.*

4.1 Frontend Pseudocode

Sign In/Up:

States:

- **hasAccount: boolean**
  - checks if the user already has an account or needs to create one.
  - default: false
  - purpose: Display login page if true otherwise sign up page.

- **email: string**
  - holds the email inputted by the user.
  - default: ""
  - purpose: Sent to the backend for user verification.

- **password: string**
  - holds the password inputted by the user.
  - default: ""
  - purpose: Sent to the backend for authentication.

- **sendingLoginRequest: boolean**
  - signals when to send a login request.
  - default: false
  - purpose: prevents redundant login attempts.

- **sendingSignupRequest: boolean**
  - signals when to send a signup request.
  - default: false

  purpose: Prevents redundant signup attempts.

Functions:

- **toggleHasAccount( )**

Input: None
Output: None
Functionality: Toggles the state of hasAccount to switch between login and signup views.

Pseudocode:

```
function toggleHasAccount( ) {
  setHasAccount((previousState) => !previousState);
}
```

- **handleEmailChange( )**

Input: Event e
Output: None
Functionality: Updates email with the latest value from the input field.

Pseudocode:

```
function handleEmailChange(e) {
  setEmail(e.target.value);
}
```

- **handlePasswordChange( )**

Input: Event e
Output: None
Functionality: Updates password with the current value from the input field.

Pseudocode:

```
function handlePasswordChange(e) {
  setPassword(e.target.value);
}
```

- **signIn( )**

Input: None
Output: None
Functionality: Activates the login process by setting sendingLoginRequest to true.

Pseudocode:

```
function signIn( ) {
  setSendingLoginRequest(true);
}
```

- **signUp( )**

Input: None
Output: None
Functionality: Activates the login process by setting sendingLoginRequest to true.

Pseudocode:

```
function signUp() {
   if (!sendingSignupRequest) {
      setSendingSignupRequest(true);
      loginWithRedirect({
         screen_hint: "signup",
         email: email,
         redirect_uri: currentURL
      });
   }
}
```

- **handleAuthentication( )**

Input: None
Output: None
Functionality: Checks the authentication status and updates the app state accordingly

Pseudocode:

```
useEffect(() => {
   if (isAuthenticated) {
      setLoggedIn(true);
      clearInputFields();
   } else {
      setLoggedIn(false);
   }
}, [isAuthenticated]);
```

Landing Page Component: Import existing components like navbar and footer

- States:
  - searchQuery (string, initially ‘’)
  - Categories(array of objects, initially [])
  - Expanded(boolean, initially false)
- Functions:
  - handleSearchChange = (e) => {setSearchQuery(e.target.value)}
  - handleSetCategories = () => {Categories.map(card = > (<CategoryCard key={category.id} name={category.name} image={category.image}>))}
  - handleSetExpanded = () => {setExpanded(!Expanded)}
  - CategoryCard = ({image, name}) => {
    return (

    {/* Image Container for given category */}

    });
- Pseudocode:
  - return (
    <>
    <NavBar />

    {/* Search Section with Search Bar */} = Value of the search input will be searchQuery state and will update upon user input causing rerender for desired categories
    {/* Browser Categories Section with Browser button*/} = Clicking the browse button will browse all existing categories that exist in the Categories state

    <Footer />
    </>
    );

4.2 Backend Pseudocode

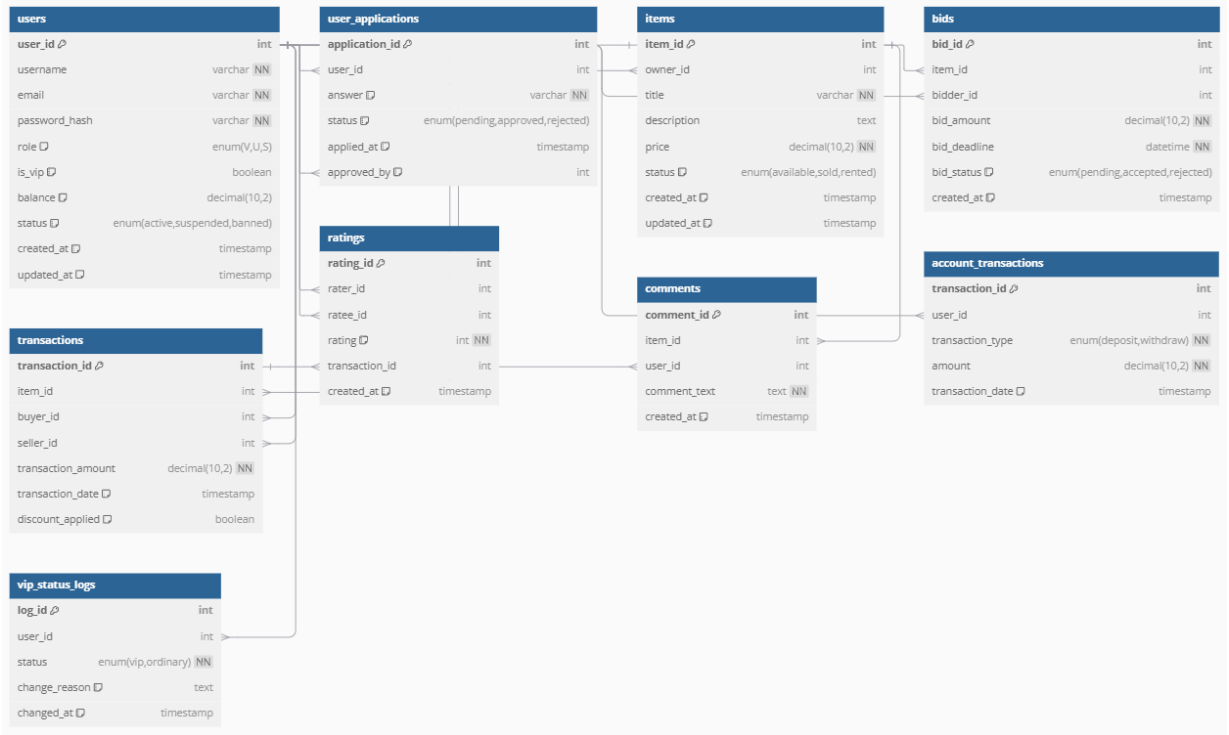
INTRODUCTION

We have decided to use “Express.js” Framework of javascript to make an HTTP server which will send and receive data connected to the Frontend which will be built with “React.js” and other basic Website development technologies like (HTML/CSS Tailwind, components).. In this designing idea for backend we will be discussing the backend database schema, how we decided on the schema, the API endpoints that will allow the frontend to get data from the server and the MVC control pattern (Model View Controller).

DATABASE

As part of this detailed design for the backend we decided to present our planned database schema for the application. Each box represents a distinct database table, each row is the column within that table with the specific name and data types. Lines represent Many-to-Many relationships between junction tables and their respective key types. This database is designed using DBML on dbdiagram.io site. The code and original image can be found at

https://dbdiagram.io/d/QuikBID-Database-Design-6733fbdee9daa85aca3cd50f



DATABASE Designing idea:

Users -: we have decided to go with **user_id** as the main form of identifying users. This will be the foundation table capturing important details about each person on the platform, their username and email, role, balance and status. we have set the role to be of 2 types, User and Superusers. By including a role field, we're able to differentiate between regular users, super-users with administrative privileges. This makes it easy to control access levels across different parts of the platform. We've also added a balance field so users can conduct transactions directly through their accounts, which opens up possibilities for convenient, internal interactions. For users with premium or VIP privileges, the is_vip field helps us apply specific benefits, like discounts, without needing a separate lookup. Timestamps (created_at and updated_at) give us a clear history of each user's journey, allowing for easier data management and audit trails for some features.

User_Application -: The user_applications table is designed to manage the application process for Visitors who want to become full members, adding an extra layer of security and flexibility (This table will be filled during the application phase). This table links each application back to the users table with a user_id reference, which ties applications directly to specific users. For an additional verification step, we included an answer field to store responses to CAPTCHA or similar questions, helping keep bots at bay. Applications go through a straightforward approval process, tracked by status and linked to the approving super-user with approved_by. This setup makes it simple for super-users to review, approve, or reject applications, while preserving a record of each decision for accountability.

Items -: The items table organizes all listed items, with fields that capture ownership, pricing, descriptions, and current status (like available or sold). By tying each item to an owner_id, we ensure listings are traceable to specific users, which streamlines management and potential disputes. With title and description, users can clearly present their items, while the price (which will be updated each Higher Bids) and status fields keep things straightforward when it comes to evaluating items and managing their availability. The timestamps here (created_at and updated_at) help us track the lifespan and changes of each listing, which is particularly useful for time-sensitive platforms or analytics.

Bids -: The bids table manages competitive offers on items. Each bid is linked to both the bidder and the item being bid on, making it clear who's bidding on what. The bid_amount lets users submit specific offers, and by setting a bid_deadline, we create urgency and clear expectations for both buyers and sellers. The bid_status field is useful for tracking where each bid stands in the process—whether it's pending, accepted, or rejected—while a created_at timestamp logs exactly when each bid was placed. Altogether, this table enables smooth bidding interactions and clear tracking for all parties involved.

Transactions -: Our transactions table keeps track of successful sales or rentals, forming a reliable record of exchanges on the platform. Each transaction includes links to the buyer, seller, and item involved, providing complete clarity on all parties. The transaction_amount column stores the agreed-upon price, while the transaction_date timestamp confirms when the exchange took place. Additionally, the discount_applied field helps us record VIP perks, showing whether a special discount was applied to each transaction. This approach ensures transparency, creating a seamless record for users, platform administrators, and any analytics.

Ratings -: The ratings table encourages trust and quality by letting users rate each other after transactions. Each rating is tied to a specific transaction and includes references to the rater_id and ratee_id so we know who's rating whom. Ratings range from 1 to 5, which can be aggregated to give users a score, supporting a reputation system that helps other users make informed choices. The created_at timestamp allows us to track how recent each rating is, which can be useful in monitoring ongoing user behavior. This table ultimately helps foster a safe, trustworthy environment where user interactions are recorded and rated.

VIP status logs -: vip_status_logs table is designed for tracking users' VIP status changes over time. Each log entry includes the user_id and status fields, capturing whether the user was promoted to VIP or returned to an ordinary user level. The change_reason allows us to document why each change occurred, which could be useful for customer support or internal audits. The changed_at timestamp is key here, offering a chronological record of all status adjustments, which adds transparency and helps administrators manage VIP privileges effectively.

This database is planned to be built using inhouse hosted POSTGRESQL database (A better version of mySQL) since all of our teammates have experience in using mySQL. Alternatively if things get more complex and working with SQL might create issues, we have created a backup plan to pivot to using MONGODB Atlas hosted on the cloud.

BACKEND ROUTES (API Endpoints)

A route essentially maps an incoming HTTP request (like GET /home or POST /user) to a specific function or handler that processes it and returns a response (Basic CRUD functions). Routers play a key role in organizing and structuring code, especially in RESTful APIs, web applications, and microservices. We will be dividing our backend routes in different routers which make managing and developing the routes easier as the application grows. Below you will be able to see different routes and them bundled together in different sections for the router. The frontend can easily send the respective type of request to these ENDPOINTS to update the database, get relevant data regarding the application and perform CRUD operations.

We will be using AXIOS in frontend to send and manage sending these requests, A simple example of this can be using Axios to send a GET request to the endpoint "/api/items". Find below the sample code for that.

```
axios.get('/api/items')
  .then(response => {
        console.log(response.data);  // Logs the response data to the console
  })
  .catch(error => {
          console.error('Error fetching items:', error);  // Logs any error that occurs during the request
  });
```

Summary of the HTTP methods that we have based the API routes on

POST - Adding new data, frontend needs to send data with these requests to HTTP server

GET-Getting new data form HTTP server which is also connected to the PostgreSQL database

PUT - Use to update an already existing database entry or to simply update some statuses.

DELETE - Used for deleting user info and entries from database.

Authentication & User Management

1. **POST** /api/login - Authenticates the user and provides a session token.
2. **POST** /api/logout - Logs the user out by invalidating the session.
3. **POST** /api/register - Allows visitors (V) to sign up as a user (U), creating an account.
4. **GET** /api/users/:id - Fetches profile data for a specific user by their ID.
5. **PUT** /api/users/:id - Updates user profile information, such as username or balance. (CRUD: Update)
6. **DELETE** /api/users/:id - Soft-deletes a user's account (for deactivation or suspension).

Visitor to User Application Process

7. **POST** /api/applications - Submits a visitor's (V) application to become a user (U), including their CAPTCHA answer.
8. **GET** /api/applications/:id - Retrieves details of a specific application (for super-user review).
9. **PUT** /api/applications/:id/approve - Approves a visitor application to grant user (U) status. (CRUD: Update)
10. **PUT** /api/applications/:id/reject - Rejects a visitor application, denying user (U) status. (CRUD: Update)

Item Listings

11. **POST** /api/items - Creates a new item listing for sale or rent, provided by a user (U).
12. **GET** /api/items - Retrieves all available items (with optional filters like category, price range).
13. **GET** /api/items/:id - Retrieves details of a specific item by its ID.
14. **PUT** /api/items/:id - Updates an existing item listing with new details, such as price or status.
15. **DELETE** /api/items/:id - Deletes an item listing or marks it as sold/rented.

Bidding on Items

16. **POST** /api/bids - Submits a bid for a specific item, provided by a qualified user (U).
17. **GET** /api/bids/:item_id - Retrieves all bids for a particular item.
18. **PUT** /api/bids/:id/accept - Marks a bid as accepted, finalizing the deal with the buyer.
19. **PUT** /api/bids/:id/reject - Rejects a specific bid.

Transactions

20. **POST** /api/transactions - Creates a transaction entry after a bid is accepted, transferring ownership and funds.
21. **GET** /api/transactions - Retrieves a list of transactions involving a specific user (U), both as buyer and seller.
22. **GET** /api/transactions/:id - Retrieves details of a specific transaction.
23. **PUT** /api/transactions/:id/discount - Applies a VIP discount to a transaction. (CRUD: Update)

Ratings

24. **POST** /api/ratings - Adds a new rating for a user after a completed transaction.
25. **GET** /api/ratings/:user_id - Fetches all ratings given to a specific user.
26. **DELETE** /api/ratings/:id - Removes a rating entry (admin-only action).

Comments on Items

27. **POST** /api/comments - Adds a comment to a specific item listing.
28. **GET** /api/comments/:item_id - Retrieves all comments associated with an item.
29. **DELETE** /api/comments/:id - Deletes a specific comment.

Account Management

30. **POST** /api/account/deposit - Allows a user to deposit funds into their account balance.
31. **POST** /api/account/withdraw - Allows a user to withdraw funds from their account balance.
32. **GET** /api/account/transactions - Retrieves a user's account transaction history (e.g., deposits/withdrawals).

VIP Status Management

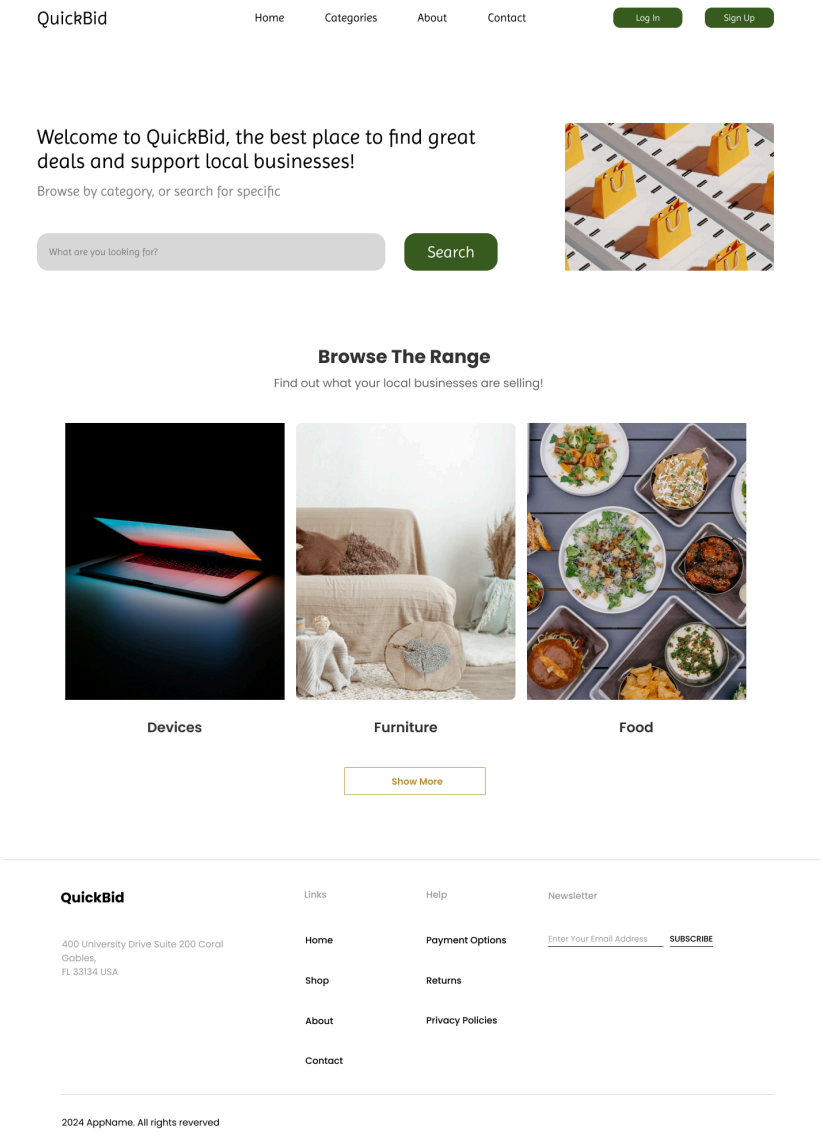
33. **GET** /api/vip-status/:user_id - Checks the VIP status of a user.
34. **PUT** /api/vip-status/:user_id/upgrade - Upgrades a user to VIP status if they meet requirements. (CRUD: Update)
35. **PUT** /api/vip-status/:user_id/revoke - Revokes VIP status and returns them to regular user privileges.

Complaints

36. **POST** /api/complaints - Submits a complaint against another user (U) for super-user review.
37. **GET** /api/complaints - Retrieves all complaints (admin-only access).
38. **PUT** /api/complaints/:id/resolve - Marks a complaint as resolved.Suspension & User Management by Super-Users
39. **PUT** /api/users/:id/suspend - Suspends a user with low ratings or due to violations (admin action).
40. **PUT** /api/users/:id/reactivate - Reactivates a suspended user after a penalty is paid. (CRUD: Update)
41. **DELETE** /api/users/:id/ban - Permanently bans a user after repeated suspensions.

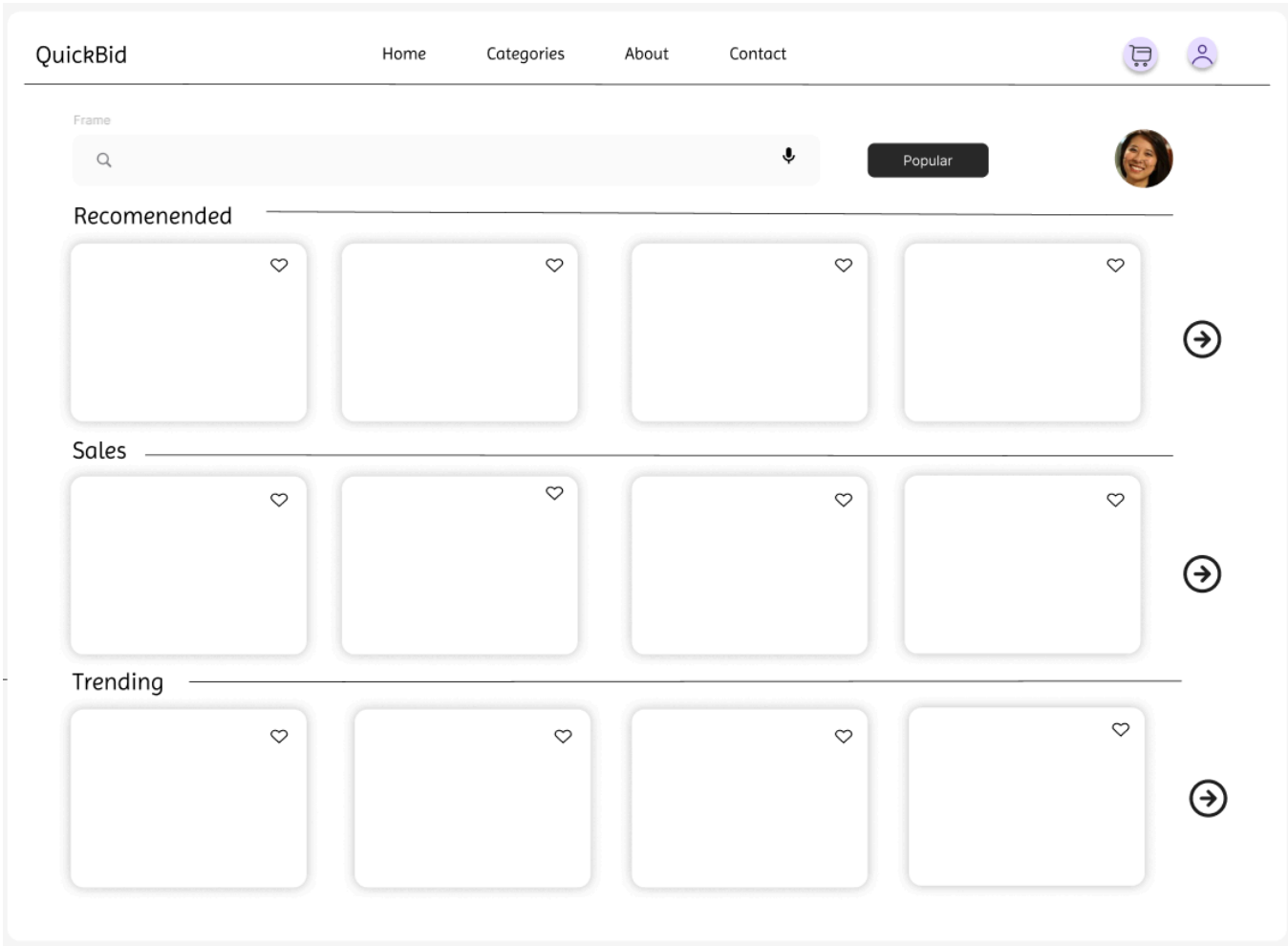
5. System Screens

5.1 Landing Page

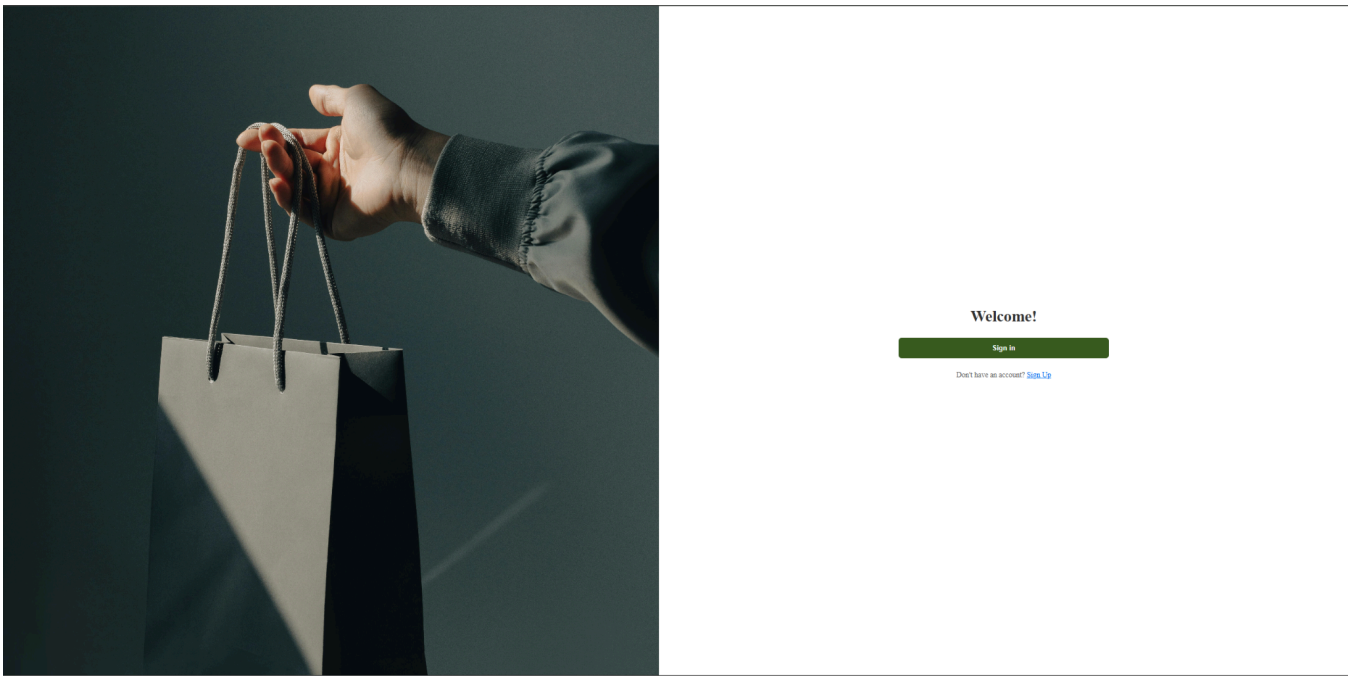


The above image is the ‘landing page’ of the application and it's the first page that all users see when visiting the website. It features a header navigation bar with easy-access buttons for Sign Up (for new users) and Login (for returning users). In the main section, visitors can search for specific items or explore various categories to view listings offered by different businesses. The page concludes with a footer that provides quick links for streamlined navigation. This layout aims to enhance user experience by organizing essential actions and information in a clean, accessible way.

5.2 Home Page

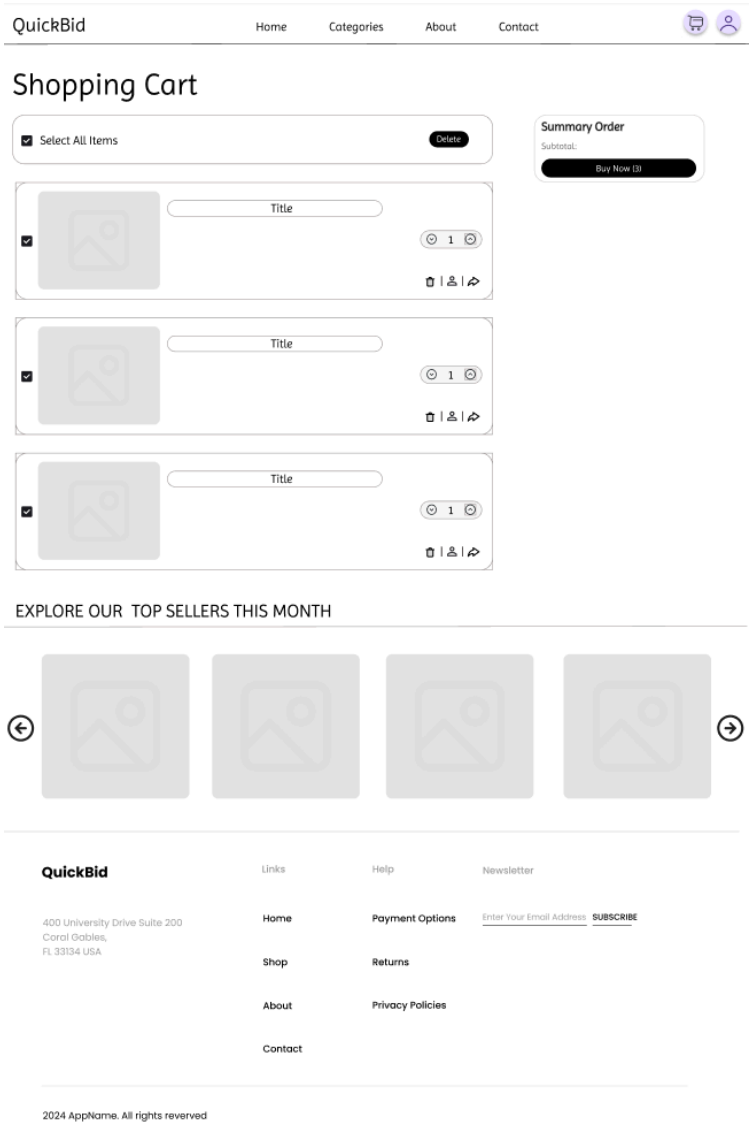

QuickBid
Home
Categories
About
Contact
Frame
Popular
Recomenended
Sales
Trending

5.3 Login Page



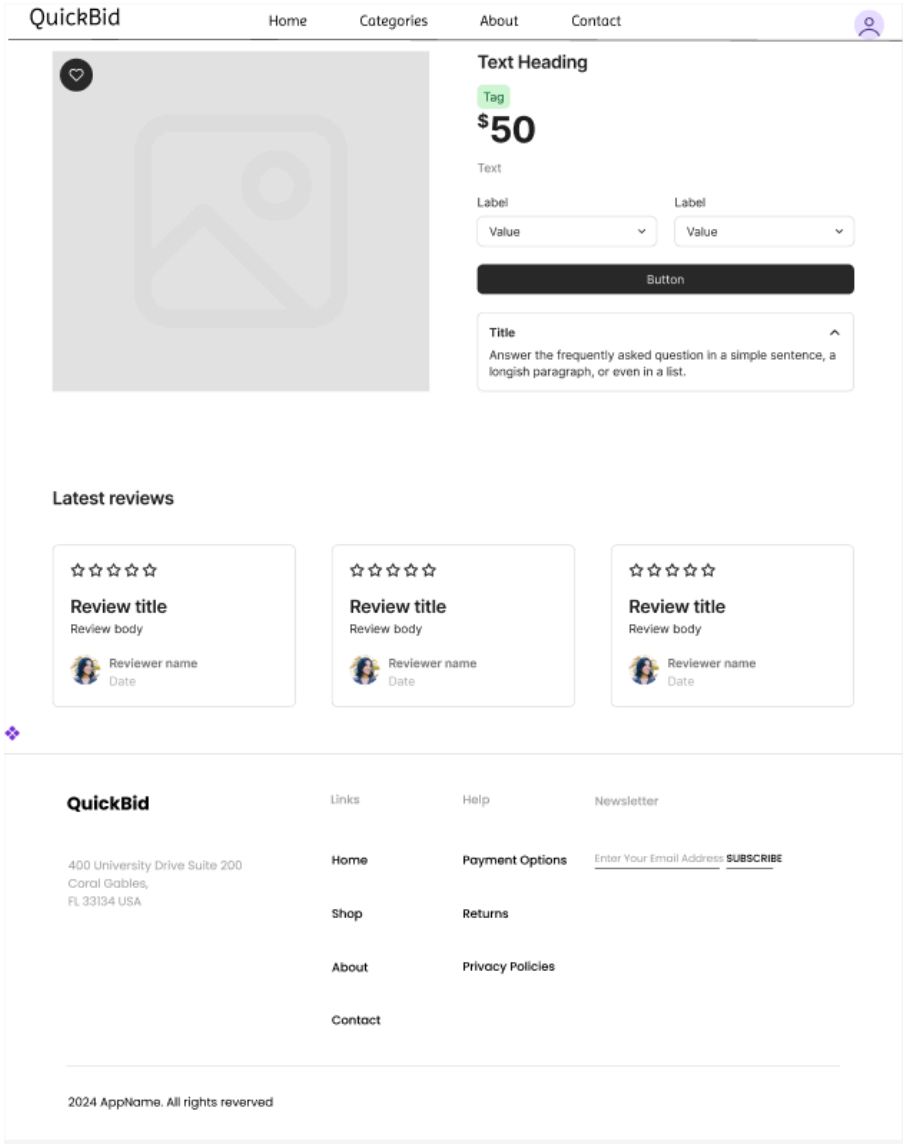
When the user clicks either the Login or Sign Up button on the landing page, they are directed to the authentication page. Here, they can securely log in to their account or register as a new user.

5.4 Shopping Cart Page



The shopping cart page provides visualization for clients that allows them to review and manage their selected items before finalizing their purchase. The design includes a "Select All Items" option for easy selection of all items, and each item is displayed with its title, image placeholder, quantity selector, and icon options for adjustments. A summary order section on the right-hand side allows the user to quickly see the total cost of all selections that exist in their cart. Additionally, the bottom page will highlight a section showcasing "Top Sellers This Month," effectively capturing client's attention for further potential purchases if interested.

5.5 Selected Item Page



6. Group Meetings

6.1 Meeting Logs

Date	Attendance	Discussion
9/11/2024	Ivan, Azim, Arihant, Tej, Rahat	Self-introduction and identified strengths, weaknesses and preferences (frontend or backend)
9/13/2024	Ivan, Azim, Arihant, Tej, Rahat	Identified the programming languages and its frameworks to utilize for frontend and backend. For the frontend we decided to use ReactJS and ExpressJS for the backend.
9/14/2024	Ivan, Azim, Arihant, Tej, Rahat	This day was relatively easy as the team was engaged with other tasks, but we shared the link for Google Docs and Figma. We used Google Docs as a means for any ideations and Figma for developing wireframes.
9/22/2024	Ivan, Azim, Arihant, Tej, Rahat	Before drafting our initial report, we chose to ideate a project that truly piqued our interest. We transformed these ideas into a tangible design on Figma, using its powerful features to create detailed wireframes and bring our concept to life visually.
9/23/2024	Ivan, Azim, Arihant, Tej, Rahat	Everybody worked on the phase 1 report together by using the Google Docs and Figma shared.
9/27/2024	Ivan, Azim, Arihant, Tej, Rahat	Finished phase 1 report
10/17/2024	Ivan, Azim, Arihant, Tej, Rahat	We finalized the phase 1 report by thoroughly reviewing it to ensure there were no mistakes + submission
10/25/2024	Ivan, Azim, Arihant, Tej, Rahat	Met up to discuss our approach on phase 2 report, which included diagrams, system screens, and pseudocode. We also finalized task assignments to ensure clear responsibilities and an efficient workflow for the report's completion.

10/26/2024	Ivan, Azim, Arihant, Tej, Rahat	Met up for a short progress check on each member before leaving for our own tasks.
11/04/2024	Ivan, Azim, Arihant, Tej, Rahat	The team remained in the meeting working on their assigned tasks. We made sure to provide support to one another whenever issues arose. By the end of the meeting, all diagrams were successfully completed by using Lucidchart. We scheduled our next meeting for 11/6/2024, aiming to finalize the rough draft.
11/06/2024	Ivan, Azim, Arihant, Tej, Rahat	Finalized the rough draft by finishing the pseudocode and major GUI screens.
11/12/2024	Ivan, Azim, Arihant, Tej, Rahat	Revisions and a final check to ensure no mistakes were made + submission

6.2 Work Distribution

Software requirements Specifications (Phase 1):

- Ivan:
    - Introduction
    - Purpose
    - Scope
    - Group Meetings

- Tej:
    - Definitions
    - Acronyms
    - Abbreviations and References
    - Overview
    - Group Meetings

- Rahat:
    - Overall Description
    - Use-Case Model Survey
    - Group Meetings

- Azim:
    - Assumptions and Dependencies
    - Specific Requirements
    - Use-Case Reports
    - Group Meetings

- Arihant:
    - Supplementary Requirements
    - Supporting Information
    - Group Meetings

Design Report (Phase 2):

- Arihant
    - ER Diagram
    - Pseudocode
    - Group Meetings

- Ivan:
    - Use case scenarios
    - Petri Nets
    - Group Meetings

- Azim:
    - Collaboration Class Diagrams
    - Pseudocode
    - Group Meetings

- Tej:
    - Major GUI Screens
    - Pseudocode
    - Group Meetings

- Rahat:
    - Major GUI Screens
    - Pseudocode
    - Group Meetings

7. GitHub

]https://github.com/notprowler/quickbid. We plan on using the “Issues” feature (Instead of JIRA) of github to correctly organize tasks, keep everyone accountable and collaborate effectively on all features.