1. **INTRODUCTION**

In today's world, the practice of sharing resources has become increasingly significant, providing an alternative to the traditional model of ownership. This shift not only helps individuals save money but also promotes sustainability by reducing waste and the demand for new products. Inspired by these principles, we developed NeighbourShare, a mobile application designed to facilitate the sharing and renting of items within local communities.

NeighbourShare is built using React Native and Expo, ensuring a smooth and responsive user experience across different mobile devices. Firebase serves as the backbone of the application, providing robust solutions for user authentication and database management through its Firestore service. The core idea behind NeighbourShare is simple yet powerful: to create a platform where users can list items they are willing to rent out and find items they need, all within their neighborhood.

The application starts with a secure authentication process, requiring users to sign up or log in using their Gmail accounts. This step not only simplifies the login process but also enhances security by leveraging Gmail’s built-in verification features. Once authenticated, users need to verify their Gmail accounts to access the full functionality of the app, ensuring that all users are genuine and traceable.

One of the critical features of NeighbourShare is its location-based service, which requires users to grant location permissions. This allows the app to display items available for rent within the user’s vicinity, making it convenient for users to find and share items locally. By focusing on nearby transactions, NeighbourShare reduces the logistical challenges and costs associated with borrowing and lending items.

Upon entering the app, users are greeted by the home page, which serves as the central hub for all sharing activities. Here, users can browse through a comprehensive list of items available for rent as well as rent requests posted by other users. The home page is designed to be intuitive, making it easy for users to find items of interest quickly. Items are categorized and displayed with relevant details, ensuring that users can make informed decisions.

NeighbourShare also features a range slider, allowing users to filter items based on distance from their current location. This functionality enhances user convenience by ensuring that the displayed items are within a manageable distance, making the sharing process more practical and efficient.

The application supports robust item management capabilities. Users can add their own items to the list of available rentals, providing details such as the item name, description, and rental terms. Additionally, users can post rent requests if they need specific items, creating a dynamic and interactive community marketplace. This dual functionality caters to both those who have items to share and those who are looking to borrow, fostering a balanced ecosystem of sharing.

To facilitate easy communication and transactions, NeighbourShare displays the contact details of the item owner on the item detail page. Users can directly contact the owner via the provided mobile number to discuss terms and arrange for the sharing of the item. This direct line of communication helps establish trust and ensures smooth transactions, making the sharing experience seamless and reliable.

Technically, NeighbourShare is a modern application that leverages the strengths of React Native for cross-platform compatibility and Expo for an efficient development and testing environment. Firebase's integration provides a secure and scalable backend solution, with Firestore offering fast and reliable data storage and retrieval capabilities.

NeighbourShare aims to promote sustainable living by encouraging the sharing of resources within local communities. The application not only helps users save money and reduce waste but also fosters a sense of community and cooperation. This documentation serves as a guide to understanding and utilizing the full potential of NeighbourShare, from installation and configuration to everyday use. We hope that users and developers alike will find this documentation helpful and that NeighbourShare will make a positive impact on communities everywhere.

1. **LITERATURE SURVEY**

**2.1 SIGNIFICANCE**

* **Promoting Sustainable Practices:** Encouraging the sharing of resources reduces unnecessary consumption and waste, contributing to environmental sustainability and conservation efforts.
* **Economic Benefits:** NeighbourShare offers a cost-effective alternative to purchasing items outright, saving users money while promoting responsible consumption habits.
* **Enhancing Accessibility:** Through its user-friendly interface and location-based connectivity, NeighbourShare makes it easier for individuals to access resources they may not have readily available, thereby increasing overall accessibility within communities.

**2.2 SEARCH STRATEGIES**

* Search Strategy Implementation: For our search, we chose specific keywords like "neighborhood sharing app" and "peer-to-peer rental platforms" to find similar platforms to NeighbourShare. We focused on platforms that share similar features, like borrowing and lending based on location, good user reviews, and easy accessibility.
* We also looked at various frameworks and tools to build this app and choose React Native to build this app as the development with it is easy with the help of JavaScript and component-based architecture.
  1. **SOURCES OF LITERATURE**
* **Online Platforms and Play Store:** Examination of online platforms and app stores, such as the Google Play Store and Apple App Store, allowed us to analyze existing sharing apps like Olx. These platforms provided insights into successful implementations and user feedback.
* **Documentation of React Native:** We went through the official documentation of React Native and other platforms for guidance on mobile application development. The React Native documentation offered insights into app structure, component development etc.

|  |  |  |  |
| --- | --- | --- | --- |
| **Study** | **Methodology** | **Data Source** | **Key Findings** |
| Vasquez et al. (2017) | User Behavior Analysis, Social Network Analysis | Online Item-Sharing Platforms | Analyzed user behavior and engagement patterns on item-sharing platforms. Identified key factors influencing user participation and the impact of social networks on sharing frequency. |
| Jones et al. (2018) | Case Study, Surveys | Community Sharing Programs | Investigated community-based sharing programs. Highlighted the environmental benefits of resource sharing and the challenges faced in fostering community trust and participation. |
| Kim et al. (2019) | Mobile App Usability Testing | Prototype Sharing Apps | Conducted usability testing on mobile sharing apps. Identified critical design elements that enhance user experience and engagement. Provided recommendations for improving app interfaces to increase adoption. |
| Singh et al. (2020) | Sustainability Assessment | Urban Sharing Initiatives | Assessed the sustainability impacts of urban sharing initiatives. Found significant reductions in waste and resource consumption. Emphasized the role of technology in facilitating sustainable practices. |
| Chen et al. (2021) | Machine Learning, Sentiment Analysis | Online Marketplaces | Applied machine learning and sentiment analysis to understand user reviews on sharing platforms. Identified key factors that contribute to positive and negative user experiences, providing insights for improving service quality |

Table 2.1: Analysis of Item-Sharing Platforms and Related Technologies

1. **SYSTEM ANALYSIS**

System analysis is a critical phase in the project management and development process. It provides a solid foundation for project planning, design, development, and implementation, helping to ensure that the final system meets the intended objectives and user needs. This analysis aims to gather detailed information about the existing system and its disadvantages, proposed system and its attributes like its advantages and advancements.

**3.1 EXISTING SYSTEM**

In today's market, multiple avenues exist for individuals seeking to borrow or rent items. These methods range from traditional practices to modern technological solutions, each with its own set of features and limitations.

1. **Traditional Methods:** Borrowing items from friends or relatives has been a long-standing practice. This method relies heavily on personal relationships and trust, limiting the pool of available items to one's immediate social circle. While it eliminates the need for monetary transactions, it is often inconvenient and restricts the variety and availability of items.
2. **Buying Used Items:** Platforms such as eBay, Craigslist, olx, and Facebook Marketplace facilitate the purchase of used items. These platforms provide a broader range of items and are often cost-effective. However, purchasing used items may not be suitable for temporary needs, leading to unnecessary ownership and potential waste.
3. **Online Item-Sharing Platforms:** Modern item-sharing platforms like Fat Llama and ShareGrid are designed specifically for renting items. These platforms allow users to list and rent a wide variety of items, ranging from electronics to household goods. Common features of these platforms include:

* **User Authentication:** Basic sign-up and login functionalities, often without stringent verification processes.
* **Location-Based Services:** Some platforms use location data to help users find items nearby, though the accuracy and efficiency of these services can vary.

#### 3.1.1 DISADVANTAGES OF EXISTING SYSTEM

Despite the functionalities offered by traditional methods and modern item-sharing platforms, several disadvantages persist, which NeighbourShare aims to address.

1. **Limited Verification and Security:** Many platforms lack robust verification processes, resulting in trust issues and potential fraud. Basic authentication methods do not sufficiently ensure the identity and reliability of users.
2. **Inefficient Search and Match Algorithms:** Existing search functionalities are often rudimentary, leading to poor user experience. Users may struggle to find the most relevant items, resulting in missed opportunities and dissatisfaction.
3. **Outdated User Interfaces:** Some platforms suffer from outdated or unintuitive interfaces, making navigation cumbersome and deterring user engagement. An intuitive design is crucial for a positive user experience.
4. **Lack of Real-Time Data:** Many platforms do not provide real-time updates on item availability, leading to booking conflicts and frustration for both renters and owners.
5. **Inaccurate Location Services:** Location-based features can be inaccurate or poorly implemented, making it difficult for users to find nearby items. Effective use of geolocation data is essential for convenience.
6. **Limited Scalability:** Existing systems may not be designed to handle a growing user base effectively, leading to performance issues and potential downtime as the platform expands.
7. **Environmental Impact Neglect:** Current platforms often fail to emphasize the environmental benefits of item sharing, missing an opportunity to educate users and promote sustainable practices.
8. **Dependency on Personal Networks:** Traditional methods of borrowing items are restricted to one's personal network, limiting the variety and availability of items.
9. **Inconvenience in Buying Used Items:** While buying used items can be cost-effective, it is not always practical for temporary needs and contributes to ownership clutter and waste.

**3.2 PROPOSED SYSTEM**

NeighbourShare aims to overcome the limitations of existing item-sharing platforms by introducing a comprehensive, user-friendly, and environmentally conscious solution. Our platform is designed to facilitate the sharing and renting of items in a way that is efficient, secure, and beneficial to the community.

1. **Enhanced Authentication:** Utilizing Firebase Authentication, NeighbourShare ensures secure and reliable user verification. This includes email verification, providing an additional layer of security and trust for all users.
2. **Modern User Interface:** Designed using React Native and Expo, the user interface is sleek, intuitive, and accessible. This modern design ensures that users can easily navigate the app, find items, and complete transactions with minimal effort.
3. **Real-Time Data Updates:** Integration with Firebase Firestore enables real-time updates on item availability and user activity. This minimizes the risk of booking conflicts and ensures that users have the most current information.
4. **Accurate Location Services:** NeighbourShare uses precise geolocation data to accurately display nearby items, making it easier for users to find what they need close to their location.
5. **Scalable Architecture:** Leveraging Firebase's cloud infrastructure, NeighbourShare is built to scale efficiently. This ensures that the platform can handle a growing user base without performance degradation.
6. **Environmental Focus:** NeighbourShare emphasizes the environmental benefits of sharing items. By reducing the need for new purchases and promoting the reuse of existing items, the platform helps decrease waste and resource consumption.

**3.2.1 ADVANTAGES OF PROPOSED SYSTEM**

NeighbourShare's proposed system offers several advantages over existing item-sharing platforms, addressing their limitations and providing additional benefits.

1. **Increased Trust and Security:** Enhanced authentication processes, including email verification, build a safer environment for all users. This reduces the risk of fraud and increases user confidence in the platform.
2. **Improved User Experience:** The modern, intuitive interface, combined with advanced search algorithms, makes it easier for users to find and rent items. This leads to higher user satisfaction and engagement.
3. **Real-Time Availability:** Real-time data updates ensure that users always have the most current information about item availability. This minimizes booking conflicts and enhances the reliability of the platform.
4. **Better Location Accuracy:** Accurate geolocation services help users find items close to them, making the process more convenient and efficient.
5. **Scalability:** The scalable architecture ensures that NeighbourShare can grow without performance issues, effectively handling an increasing number of users and items.
6. **Environmental Impact:** By promoting the environmental benefits of item sharing, NeighbourShare encourages sustainable practices and helps reduce waste. This environmental focus not only benefits the planet but also appeals to eco-conscious users.
7. **SYSTEM STUDY**

System study involves collecting and analyzing information to understand the requirements, identify potential problems, and decompose the system into its components. It is a critical phase in the development process, aiming to improve system efficiency and ensure that all components work effectively to achieve their goals.

For NeighbourShare, the system study process involves gathering insights into user needs, platform requirements, and potential challenges. By conducting a thorough system study, we can lay the foundation for a successful and user-friendly item-sharing platform.

**4.1 FEASILIBILITY STUDY**

The feasibility study assesses the viability of the project and presents a business proposal with a preliminary plan and cost estimates. It helps determine whether the proposed system is feasible and financially viable for implementation.

**4.1.1 ECONOMICAL FEASIBILITY**

Economic feasibility examines the financial impact of the system on the organization. It evaluates whether the proposed system is financially viable and within budget constraints. In the case of NeighbourShare, the following considerations are taken into account:

* **Cost Analysis:** Assessing the expenses associated with developing and maintaining the platform, including software development, server costs, and marketing.
* **Budget Allocation:** Ensuring that the project remains within the allocated budget and that expenditures are justified.
* **Cost-Benefit Analysis:** Analyzing the potential benefits of the system in relation to its costs, considering factors such as revenue generation and cost savings.

The use of readily available technologies and minimal customizations helps keep the project economically feasible.

**4.1.2 TECHNICAL FEASIBILITY**

Technical feasibility evaluates whether the proposed system can be implemented with the available technical resources. It ensures that the system's technical requirements are realistic and manageable. Key considerations for NeighbourShare include:

* **Technology Stack:** Assessing the suitability of chosen technologies, such as React Native, Firebase, and geolocation services, for building the platform.
* **Resource Requirements:** Determining the technical resources needed for system development, deployment, and maintenance.
* **Scalability:** Ensuring that the platform can accommodate a growing user base and handle increased traffic without compromising performance.

The use of scalable technologies and minimal technical dependencies contribute to the technical feasibility of NeighbourShare.

**4.1.3 SOCIAL FEASIBILITY**

Social feasibility evaluates the level of acceptance of the system by users and stakeholders. It focuses on user acceptance, training, and confidence-building measures. For NeighbourShare, social feasibility considerations include:

**User Acceptance:** Assessing users' willingness to adopt and use the platform, considering factors such as usability, trust, and perceived value.

**Training and Education:** Implementing user training programs to familiarize users with the platform's features and functionalities.

**Feedback Mechanisms:** Establishing channels for user feedback and constructive criticism to continuously improve the platform based on user insights

1. **HARDWARE AND SOFTWARE REQUIREMENTS**

The hardware and software requirements for NeighbourShare play a crucial role in ensuring the platform's optimal functionality and performance. These requirements encompass a variety of components and technologies necessary for both development and operation, catering to the diverse needs of users and developers alike.

* 1. **HARDWARE REQUIREMENTS**

NeighbourShare demands hardware components that not only provide sufficient processing power and storage capacity but also offer reliability and compatibility with the platform's features and functionalities.

|  |  |
| --- | --- |
| Processor | I5 |
| RAM | 4 GB (minimum) |
| Hard Disk | 20 GB |
| Key Board | Standard Windows Keyboard |
| Mouse | Two or Three Button Mouse |
| Operating System | Android 8 and iOS 12 (minimum) |

**Table 5.1 Hardware requirements**

**5.2 SOFTWARE REQUIREMENTS**

NeighbourShare relies on various software components and technologies to facilitate its development, deployment, and operation. These software requirements encompass a diverse array of tools and frameworks tailored to meet the platform's unique needs and requirements.

**5.2.1 DEVELOPMENT ENVIRONMENT**

During the development phase, NeighbourShare leverages a comprehensive set of tools and environments to streamline the development process and enhance collaboration among team members.

* **Integrated Development Environment (IDE):** NeighbourShare developers use Visual Studio Code (VS Code) as their IDE for coding, debugging, and editing.
* **Android Virtual Device (AVD):** For testing on Android, NeighbourShare utilizes Android Studio's AVD Manager to create and manage virtual devices.

**5.2.2 PROGRAMMING LANGUAGE AND FRAMEWORKS**

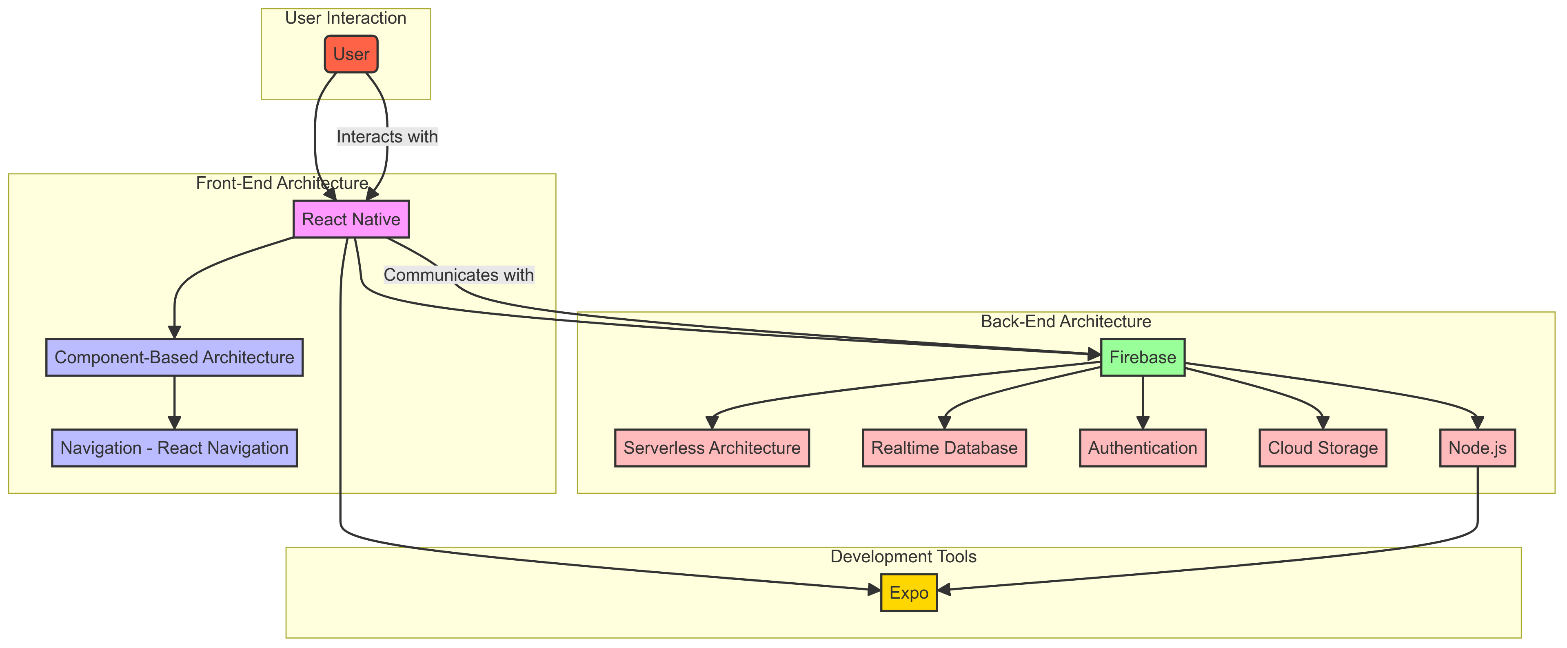
|  |  |
| --- | --- |
| Coding Language | JavaScript |
| Front-End | React Native |
| Back-End | Node JS, Firebase |
| Designing | React Style Sheet |
| Data Base | Cloud Firestore |

NeighbourShare is built using a combination of programming languages and frameworks chosen for their versatility, performance, and compatibility with mobile app development.

**Table 5.2 Software Requirements**

* **Front-End Development:** NeighbourShare's front-end is built with React Native, offering a seamless experience on both Android and iOS. React Native's single codebase speeds up development, enabling rich user interfaces.
* **Back-End Development:** For the back-end, Node.js powers NeighbourShare's infrastructure. Its asynchronous architecture handles concurrent user requests efficiently. Firebase complements Node.js, providing user authentication, real-time database management, and cloud storage for scalability and reliability.

1. **SYSTEM ARCHITECTURE**

The architecture of NeighbourShare is designed to ensure scalability, reliability, and performance while providing a seamless user experience. It encompasses both the front-end and back-end components, along with the interactions between them, to deliver a robust and responsive item-sharing platform.

**Fig 6.1 Architecture Diagram**

**6.1 FRONT-END ARCHITECTURE**

NeighbourShare's front-end architecture is built on the principles of modularity, reusability, and responsiveness, enabling the creation of a rich and intuitive user interface across multiple platforms.

* **React Native:** The front-end of NeighbourShare is developed using React Native, a JavaScript framework for building mobile applications. React Native allows for the creation of native-like user interfaces using familiar web development technologies such as HTML, CSS, and JavaScript.
* **Component-Based Architecture:** NeighbourShare's front-end follows a component-based architecture, where each UI element is encapsulated within a reusable component. This approach promotes code reusability, maintainability, and scalability, allowing developers to efficiently build and manage complex user interfaces.
* **Navigation:** NeighbourShare utilizes navigation libraries such as React Navigation to implement navigation between different screens and sections of the app. This enables users to navigate intuitively through the app and access its various features with ease.

**6.2 BACK-END ARCHITECTURE**

NeighbourShare's back-end architecture is designed to support real-time data synchronization, user authentication, and seamless communication with the front-end.

* **Firebase:** NeighbourShare's back-end infrastructure is built on Firebase, a comprehensive platform provided by Google. Firebase offers a suite of backend services, including authentication, real-time database, cloud storage, and hosting, which are integral to NeighbourShare's functionality.
* **Serverless Architecture:** NeighbourShare leverages Firebase's serverless architecture, eliminating the need for managing servers or infrastructure. This allows developers to focus on building features and functionality without worrying about server maintenance or scalability issues.
* **Realtime Database:** NeighbourShare utilizes Firebase Realtime Database to store and synchronize data in real-time across clients. This enables instant updates and notifications, ensuring that users have access to the latest information without delay.
* **Authentication:** NeighbourShare implements user authentication using Firebase Authentication, which supports various authentication methods such as email/password, social login, and phone authentication. This ensures secure access to the platform and protects user data from unauthorized access.
* **Cloud Storage**: NeighbourShare utilizes Firebase Cloud Storage to store user-uploaded files and media, such as item images and user profiles. Cloud Storage provides scalable and reliable storage solutions, ensuring efficient handling of user-generated content.
* **Node.js:** NeighbourShare uses Node.js for executing backend commands and running scripts, facilitating server-side operations and API management.
* **Expo:** Expo is used to streamline the development and deployment process, providing tools and services that enhance the React Native development experience.

1. **MODULES**

NeighbourShare is built with several key modules, each designed to handle specific functionalities within the app. These modules work together to provide a seamless experience for users, from user authentication to item management and location services.

**7.1 USER AUTHENTICATION MODULE**

The User Authentication Module ensures secure access to NeighbourShare, managing user sign-ups, logins, and authentication processes.

1. **Sign-Up/Registration:** Users can create a new account using their email and password. The module handles the collection and storage of user credentials securely.
2. **Email Verification:** After registration, users must verify their email address via a verification link sent to their email. This step ensures that only legitimate users gain access to the platform.
3. **Login:** Users can log in using their registered email and password. The module authenticates the credentials and grants access to the app.
4. **Password Recovery:** Provides functionality for users to reset their password if they forget it, ensuring they can regain access to their account.

**7.2 ITEM MANAGEMENT MODULE**

The Item Management Module handles the creation, modification, and display of items available for rent or requested by users.

1. **Add Item:** Users can list items they want to rent out by providing details such as item name, description, rental price, and images.
2. **View Available Items:** Displays a list of items available for rent. Users can browse, filter, and search for items based on their preferences.
3. **Add Rental Request:** Allows users to request specific items they need but are not currently available. Other users can see these requests and offer their items.
4. **Delete Items:** Users can remove items they have listed for rent, ensuring they can manage their inventory effectively.

**7.3 LOCATION SERVICES MODULE**

The Location Services Module integrates location-based functionalities to enhance the user experience.

1. **Location Permissions:** Requests and manages user permissions to access location data, ensuring user privacy and compliance with regulations.
2. **Range Slider:** Allows users to set a range for item searches based on their location. Users can find items available for rent within a specified distance.

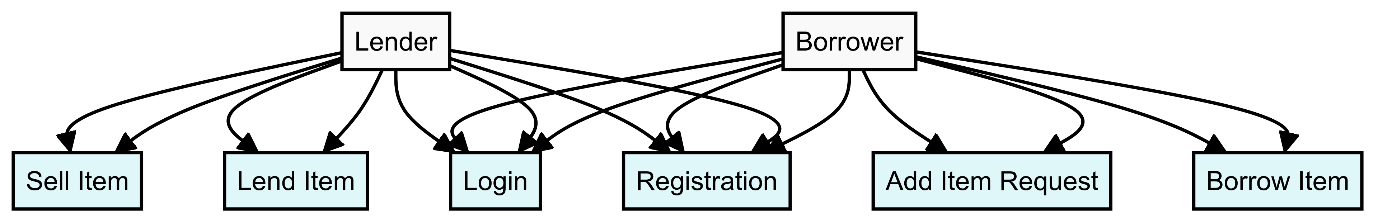
**7.4 COMMUNICATION MODULE**

The Communication Module facilitates direct interaction between users who wish to rent items and those offering items.

**Contact Information:** Displays the contact details (such as mobile numbers) of users who have listed items, enabling direct communication to negotiate and finalize rentals.

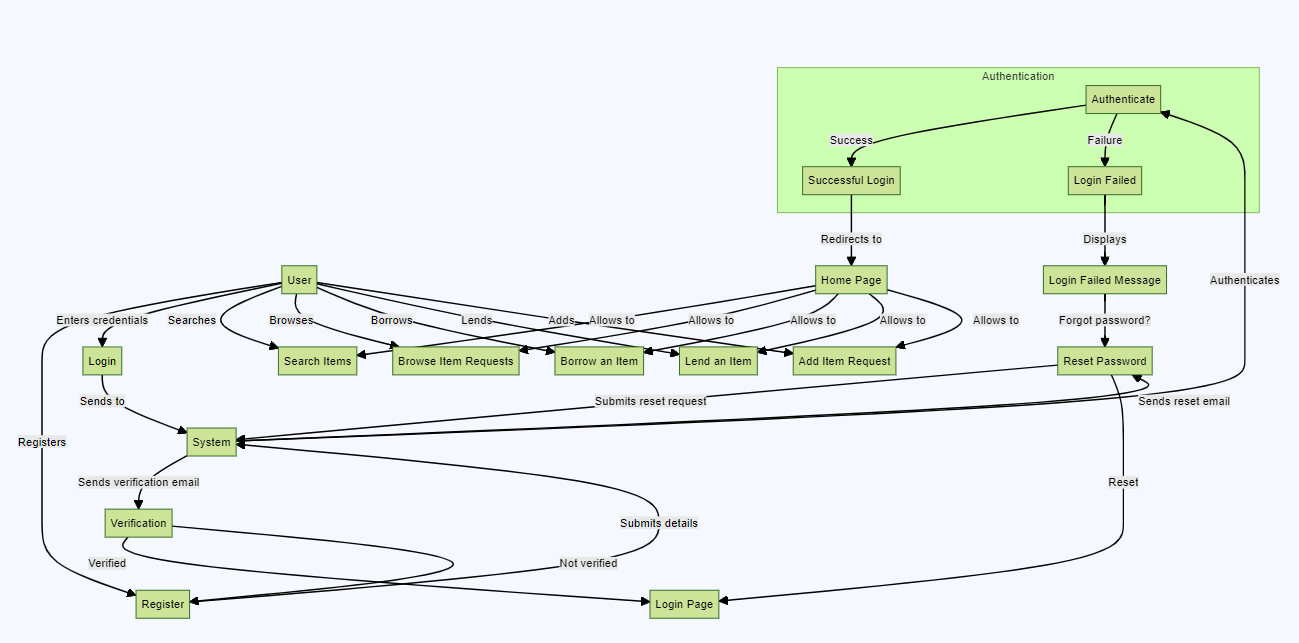
1. **DIAGRAMS**

**8.1 USE CASE DIAGRAM**



**Fig 8.1 Use Case Diagram**

* **Lender:** A user who offers items to be borrowed or sold.
* **Borrower:** A user who requests to borrow items or needs specific items.
* **Sell Item:** Enables a lender to post an item for sale.
* **Lend Item**: Allows a lender to offer an item for borrowing.
* **Login:** Provides a secure way for users to log into their accounts.
* **Registration:** Allows new users to create an account on the platform.
* **Add Item Request**: Lets a borrower post a request for specific items they need.
* **Borrow Item**: Allows borrowers to browse and borrow items offered by lenders.

**8.2 ACTIVITY DIAGRAM**

**Fig 8.2 Activity Diagram Diagram**

The Activity Diagram for NeighbourShare outlines the key interactions between users and the system, focusing on the primary functionalities of the application. This diagram highlights the main use cases and how users interact with them:

1. **User**

* **Enters credentials:** The user provides login information to access their account.
* **Searches:** The user searches for items available for borrowing.
* **Browses:** The user browses through various item requests and available items.

1. **Login**

* **Authenticate:** The system verifies the user's credentials.
* **Success:** If the credentials are correct, the user is redirected to the home page.
* **Failure:** If the credentials are incorrect, a login failed message is displayed, and the user can opt to reset the password.

1. **System**

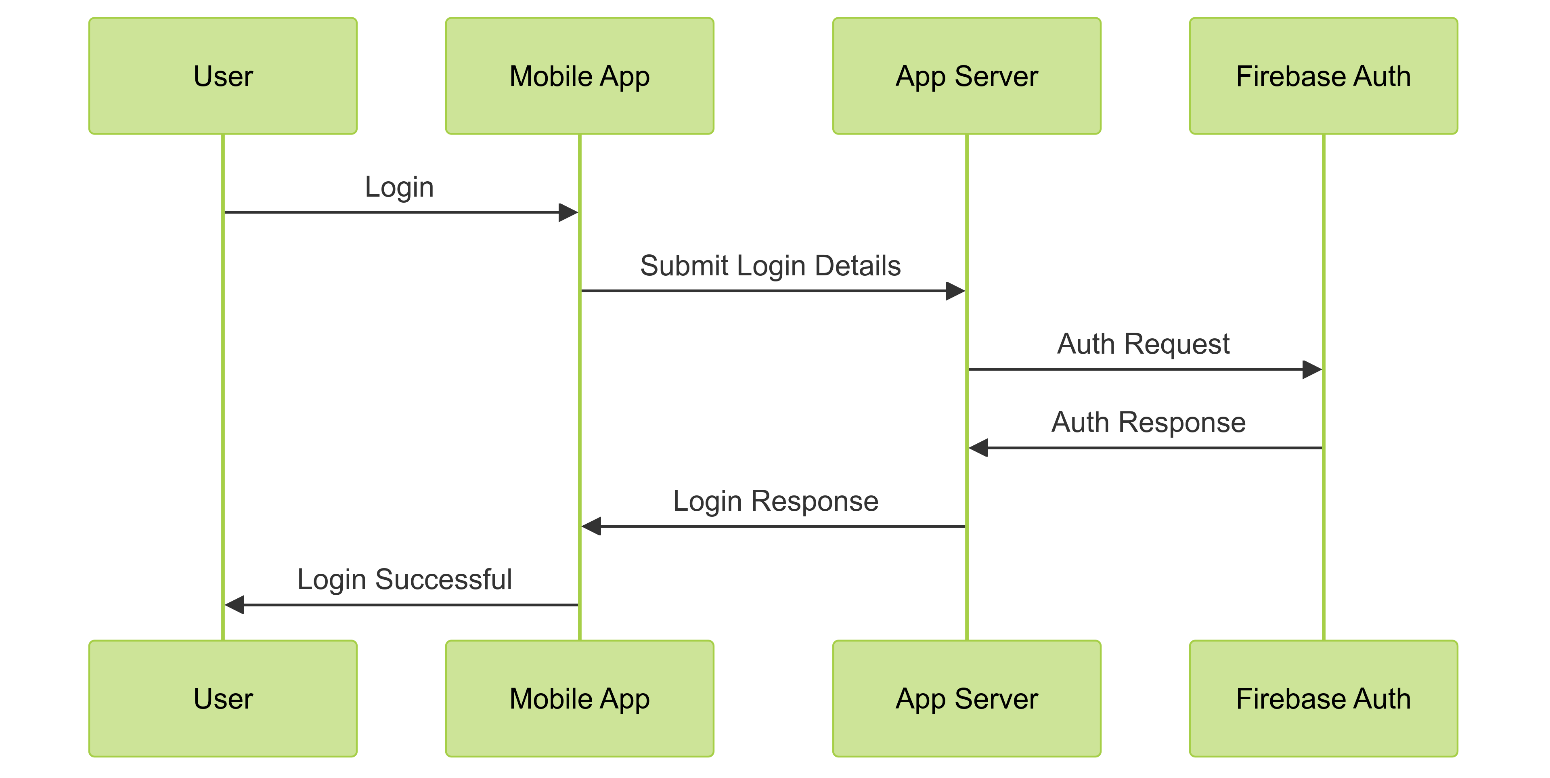
* **Register:** The system handles user registration by sending a verification email.
* **Verification:** Once the email is verified, the user's registration is completed.

1. **Home Page**

* **Search Items:** Users can search for items they wish to borrow.
* **Browse Item Requests:** Users can view requests made by others.
* **Borrow an Item:** Users can borrow items available on the platform.
* **Lend an Item:** Users can lend items they own to others.
* **Add Item Request:** Users can request items they need.

1. **Reset Password**

* **Forgot Password?:** Users can reset their password if they forget it.
* **Reset:** The system sends a reset email to the user, allowing them to set a new password.

**8.3 SEQUENCE DIAGRAM**

**Fig 8.3 Sequence Diagram**

**User Registration:**

1. **User**:
   * Initiates the registration process by entering credentials (e.g., email, password) and submitting them through the mobile app.
2. **Mobile App**:
   * Receives the user’s credentials and sends a registration request to the app server.
3. **App Server**:
   * Forwards the registration request to Firebase Authentication.
   * Waits for the authentication response from Firebase.
4. **Firebase Authentication**:
   * Processes the registration request.
   * If successful, it creates a new user record and sends a confirmation response back to the app server.

**User Login:**

1. **User**:
   * Initiates the login process by entering their credentials (e.g., email, password) and submitting them through the mobile app.
2. **Mobile App**:
   * Receives the user’s credentials and sends a login request to the app server.
3. **App Server**:
   * Forwards the login request to Firebase Authentication.
   * Waits for the authentication response from Firebase.
4. **Firebase Authentication**:
   * Verifies the user’s credentials.
   * If successful, it sends an authentication token and user details back to the app server.
5. **App Server**:
   * Receives the authentication token and user details from Firebase Authentication.
   * Sends a login success message and user details back to the mobile app.
6. **Mobile App**:
   * Displays the login success message and redirects the user to the home page or dashboard.
7. **IMPLEMENTATION**

Implementation is a critical phase in the software development lifecycle where the design is transformed into a functional system through coding and integration. This phase involves writing the actual code, integrating various components, and ensuring the system meets the specified requirements. In the context of NeighbourShare, the implementation phase covers several key aspects:

**9.1 SETTING UP THE DEVELOPMENT ENVIRONMENT**

Properly setting up the development environment is crucial for ensuring smooth and efficient development. Below are the detailed steps for setting up the necessary software tools and mobile emulators:

**9.1.1 SOFTWARE TOOLS**

1. **Visual Studio Code (VS Code):**

* **Download:** Visit the Visual Studio Code website and download the appropriate installer for your operating system (Windows, macOS, or Linux).
* **Install:** Run the installer and follow the on-screen instructions to complete the installation.
* **Extensions:** Install useful extensions such as ESLint, Prettier, and React Native Tools to enhance your development workflow. These can be found in the Extensions view (Ctrl+Shift+X).

1. **Node.js:**

* **Download:** Go to the Node.js website and download the recommended LTS version for your operating system.
* **Install:** Run the installer and follow the installation instructions. This will also install npm (Node Package Manager) which is essential for managing project dependencies.

1. **Expo CLI:**

* **Install:** Open your terminal or command prompt and run the following command to install Expo CLI globally:

npm install -g expo-cli

* **Verify:** After installation, verify that Expo CLI is installed correctly by running

expo –version

**9.1.2 MOBILE EMULATORS**

**1. Android Virtual Device (AVD):**

* **Install Android Studio:** Download and install Android Studio which includes the AVD Manager.
* **Configure AVD:**
  + Open Android Studio and navigate to Configure > AVD Manager.
  + Click Create Virtual Device.
  + Select a device from the list (e.g., Pixel 3) and click Next.
  + Choose a system image (e.g., Android 10.0) and click Next.
  + Configure the AVD settings as needed and click Finish.
* **Start AVD:** Launch the AVD by clicking the play button next to the created virtual device in the AVD Manager.

**9.1.3 ADDITIONAL CONFIGURATIONS**

**1. React Native Environment:**

* **Initialize Project:** Create a new React Native project using Expo CLI by running:

expo init NeighbourShare

* **Start Project:** Start the Expo development server:

expo start

**9.2 SOURCE CODE**

**9.2.1 App.js**

import React from 'react';

import Home from './screens/Home';

import Login from './screens/Login.js';

import Register from './screens/Register';

import RootStack from './navigators/RootStack.js';

function App() {

return (

<RootStack />

);

}

export default App;

**9.2.2 RootStack.js**

import { createStackNavigator } from '@react-navigation/stack';

import { NavigationContainer } from '@react-navigation/native';

import { firebase } from '../firebaseConfig';

import React, { useState, useEffect } from 'react';

import Home from './../screens/Home';

import Login from './../screens/Login';

import Register from './../screens/Register';

import DetailedItem from './../screens/DetailedItem';

import AddItem from './../screens/AddItem'

const Stack = createStackNavigator();

const Colors = {

primary: "#ffffff",

secondary: "#ED5B2D",

teritiary: "#291B25",

brand: "#ff8316",

blue: "#569fa0",

gray: "#f9e8d1"

};

const RootStack = () => {

const [initializing, setInitializing] = useState(true);

const [user, setUser] = useState();

const onAuthStateChanged = (user) => {

setUser(user);

if (initializing) setInitializing(false);

}

useEffect(() => {

const currrentUser = firebase.auth().onAuthStateChanged(onAuthStateChanged);

return currrentUser;

}, [])

if (initializing) return null;

return (

<NavigationContainer>

<Stack.Navigator screenOptions={{

headerStyle: {

backgroundColor: "transparent"

},

headerTintColor: Colors.teritiary,

headerTransparent: true,

headerTitle: '',

headerLeftContainerStyle: {

paddingLeft: 20

}

}}

initialRouteName={user ? 'Home' : 'Register'}>

{!user ? (

<>

<Stack.Screen name="Login" component={Login} />

<Stack.Screen name="Register" component={Register} />

</>

) : (

<>

<Stack.Screen name="Home" component={Home} />

<Stack.Screen name="DetailedItem" component={DetailedItem} />

<Stack.Screen name="AddItem" component={AddItem} />

</>

)}

</Stack.Navigator>

</NavigationContainer>

)

}

export default RootStack;

**9.2.3 Register.js**

import React, { useState } from 'react';

import { ScrollView, View, Text, TextInput, TouchableOpacity, Image, StyleSheet } from 'react-native';

import { Octicons, Ionicons, AntDesign } from '@expo/vector-icons';

import Constants from 'expo-constants';

import { firebase } from '../firebaseConfig';

const statusBarHeight = Constants.statusBarHeight;

const Colors = {

primary: "#ffffff",

secondary: "#ED5B2D",

teritiary: "#291B25",

brand: "#ff8316",

blue: "#569fa0",

gray: "#f9e8d1"

};

const Register = ({ navigation }) => {

const [fullName, setFullName] = useState('');

const [email, setEmail] = useState('');

const [password, setPassword] = useState('');

const [confirmPassword, setConfirmPassword] = useState('');

const [isPasswordVisible, setIsPasswordVisible] = useState(false);

const handleRegister = () => {

if (password !== confirmPassword) {

alert("Passwords do not match");

return;

}

firebase.auth().createUserWithEmailAndPassword(email, password)

.then(() => {

const userId = firebase.auth().currentUser.uid;

return firebase.firestore().collection('users').doc(userId).set({

fullName: fullName,

email: email

});

})

.then(() => {

const user = firebase.auth().currentUser

user.sendEmailVerification().then(() => {

alert("Verification email has been sent");

navigation.navigate("Home")

}).catch((err) => {

console.log(err)

})

})

.catch((error) => {

alert(error.message);

});

};

return (

<ScrollView style={{ backgroundColor: "#fff" }}>

<View style={styles.scrollContainer}>

<View style={styles.innerContainer}>

<Image

style={styles.pageLogo}

source={require("../assets/LogoTopBottom.png")}

resizeMode="contain"

/>

<Text style={styles.pageTag}>Don't just buy stuff on a whim!!</Text>

<Text style={styles.secondHeader}>-SIGN UP-</Text>

<View style={styles.formArea}>

<MyTextInput

label="Name:"

icon="person"

placeholder="Enter Your Name"

onChangeText={setFullName}

value={fullName}

/>

<MyTextInput

label="Email Address:"

icon="mail"

placeholder="Enter Your Email"

onChangeText={setEmail}

value={email}

keyboardType="email-address"

/>

<MyTextInput

label="Password:"

icon="lock"

placeholder="Enter Your Password"

onChangeText={setPassword}

value={password}

secureTextEntry={!isPasswordVisible}

isPassword={true}

isPasswordVisible={isPasswordVisible}

setIsPasswordVisible={setIsPasswordVisible}

/>

<MyTextInput

label="Confirm Password:"

icon="lock"

placeholder="Confirm Your Password"

onChangeText={setConfirmPassword}

value={confirmPassword}

secureTextEntry={!isPasswordVisible}

isPassword={true}

isPasswordVisible={isPasswordVisible}

setIsPasswordVisible={setIsPasswordVisible}

/>

<Text style={styles.msgBox}>...</Text>

<TouchableOpacity style={styles.styledButton} onPress={handleRegister}>

<Text style={styles.buttonText}>Sign Up</Text>

</TouchableOpacity>

<View style={styles.line} />

<View style={styles.extraView}>

<Text style={styles.extraText}>Already have an account? </Text>

<TouchableOpacity onPress={() => navigation.navigate("Login")} style={styles.textLink}>

<Text style={styles.textLinkContent}>Sign in</Text>

</TouchableOpacity>

</View>

</View>

</View>

</View>

</ScrollView>

);

};

const MyTextInput = ({ label, icon, isPassword, isPasswordVisible, setIsPasswordVisible, ...props }) => {

return (

<View>

<View style={styles.leftIcon}>

<Octicons name={icon} size={30} color={Colors.brand} />

</View>

<Text style={styles.styledInputLabel}>{label}</Text>

<TextInput style={styles.styledInput} {...props} />

{isPassword && (

<TouchableOpacity style={styles.rightIcon} onPress={() => setIsPasswordVisible(!isPasswordVisible)}>

<Ionicons size={30} name={isPasswordVisible ? "eye" : "eye-off"} color="#f99984" />

</TouchableOpacity>

)}

</View>

);

};

const styles = StyleSheet.create({

scrollContainer: {

paddingTop: statusBarHeight + 12,

},

innerContainer: {

width: '100%',

alignItems: 'center',

justifyContent: 'center',

},

pageLogo: {

width: 250,

height: 200,

margin: 0,

paddingTop: 50,

},

pageTag: {

fontSize: 12,

marginTop: 0,

textAlign: 'center',

fontWeight: 'bold',

padding: 0,

color: Colors.teritiary,

fontFamily: 'Roboto',

},

secondHeader: {

fontSize: 18,

marginTop: 0,

textAlign: 'center',

fontWeight: 'bold',

padding: 0,

margin: 20,

color: Colors.teritiary,

fontFamily: 'Roboto',

},

formArea: {

width: '85%',

},

styledInput: {

padding: 18,

paddingLeft: 50,

fontSize: 16,

backgroundColor: Colors.gray,

borderWidth: 3,

borderColor: Colors.secondary,

borderRadius: 12,

color: Colors.teritiary,

marginVertical: 4,

marginBottom: 10,

},

styledInputLabel: {

color: Colors.teritiary,

fontSize: 12,

textAlign: 'left',

padding: 5,

},

leftIcon: {

position: 'absolute',

top: 47,

left: 10,

zIndex: 1,

},

rightIcon: {

position: 'absolute',

top: 47,

right: 10,

zIndex: 1,

},

styledButton: {

backgroundColor: Colors.blue,

color: Colors.teritiary,

padding: 15,

alignContent: 'center',

borderRadius: 12,

marginVertical: 5,

justifyContent: 'center',

height: 60,

width: '60%',

alignSelf: 'center',

},

buttonText: {

color: Colors.primary,

textAlign: 'center',

fontSize: 16,

padding: 5,

},

msgBox: {

textAlign: 'center',

fontSize: 12,

},

line: {

height: 1,

width: '100%',

backgroundColor: '#44361b',

marginVertical: 5,

},

extraView: {

justifyContent: 'center',

flexDirection: 'row',

alignItems: 'center',

padding: 10,

},

extraText: {

justifyContent: 'center',

alignContent: 'center',

alignItems: 'center',

color: Colors.teritiary,

fontSize: 14,

},

textLink: {

justifyContent: 'center',

alignItems: 'center',

},

textLinkContent: {

backgroundColor: Colors.primary,

color: Colors.secondary,

fontSize: 14,

},

});

export default Register;

**9.2.4 Login.js**

import React, { useState } from 'react';

import { View, Text, TextInput, TouchableOpacity, Image, StyleSheet } from 'react-native';

import { Octicons, Ionicons, AntDesign } from '@expo/vector-icons';

import Constants from 'expo-constants';

import { firebase } from '../firebaseConfig';

//import { useNavigation } from '@react-navigation/native';

const statusBarHeight = Constants.statusBarHeight;

const Colors = {

primary: "#ffffff",

secondary: "#ED5B2D",

teritiary: "#291B25",

brand: "#ff8316",

blue: "#569fa0",

gray: "#f9e8d1"

};

const Login = ({navigation}) => {

const [email, setEmail] = useState('');

// const navigation = useNavigation();

const [password, setPassword] = useState('');

const [isPasswordVisible, setIsPasswordVisible] = useState(false);

const handleLogin =async () => {

// Add your login logic here

try {

await firebase.auth().signInWithEmailAndPassword(email, password);

navigation.navigate("Home");

} catch(error) {

alert(error.message);

}

};

const resetPassword = async () => {

if (!email) {

alert("Enter a valid email");

return

}

firebase.auth().sendPasswordResetEmail(email)

.then(() => {

alert("Password reset email has been sent!")

})

.catch((error) => {

alert(error.message)

});

}

return (

<View style={styles.container}>

<View style={styles.innerContainer}>

<Image

style={styles.pageLogo}

source={require("../assets/LogoTopBottom.png")}

resizeMode="contain"

/>

<Text style={styles.pageTag}>Don't just buy stuff on a whim!!</Text>

<Text style={styles.secondHeader}>-LOGIN-</Text>

<View style={styles.formArea}>

<MyTextInput

label="Email Address:"

icon="mail"

onChangeText={setEmail}

placeholder="Enter Your Email"

value={email}

keyboardType="email-address"

/>

<MyTextInput

label="Password:"

icon="lock"

placeholder="Enter Your Password"

onChangeText={setPassword}

value={password}

secureTextEntry={!isPasswordVisible}

isPassword={true}

isPasswordVisible={isPasswordVisible}

setIsPasswordVisible={setIsPasswordVisible}

/>

<Text style={styles.msgBox}>...</Text>

<TouchableOpacity style={styles.styledButton} onPress={handleLogin}>

<Text style={styles.buttonText}>Login</Text>

</TouchableOpacity>

<View style={styles.line} />

<View style={styles.extraView}>

<Text style={styles.extraText}>Don't have an account already? </Text>

<TouchableOpacity onPress={() => navigation.navigate("Register")} style={styles.textLink}>

<Text style={styles.textLinkContent}> Signup</Text>

</TouchableOpacity>

</View>

<View style={styles.extraView}>

<Text style={styles.extraText}>Forgot you Password? Don't worry </Text>

<TouchableOpacity onPress={resetPassword} style={styles.textLink}>

<Text style={styles.textLinkContent}> Reset Password</Text>

</TouchableOpacity>

</View>

</View>

</View>

</View>

);

};

const MyTextInput = ({ label, icon, isPassword, isPasswordVisible, setIsPasswordVisible, ...props }) => {

return (

<View>

<View style={styles.leftIcon}>

<Octicons name={icon} size={30} color={Colors.brand} />

</View>

<Text style={styles.styledInputLabel}>{label}</Text>

<TextInput style={styles.styledInput} {...props} />

{isPassword && (

<TouchableOpacity style={styles.rightIcon} onPress={() => setIsPasswordVisible(!isPasswordVisible)}>

<Ionicons size={30} name={isPasswordVisible ? "eye" : "eye-off"} color="#f99984" />

</TouchableOpacity>

)}

</View>

);

};

const styles = StyleSheet.create({

container: {

flex: 1,

paddingTop: statusBarHeight + 12.5,

backgroundColor: Colors.primary,

},

innerContainer: {

width: '100%',

alignItems: 'center',

justifyContent: 'center',

},

pageLogo: {

width: 250,

height: 200,

margin: 0,

paddingTop: 50,

},

pageTag: {

fontSize: 12,

marginTop: 0,

textAlign: 'center',

fontWeight: 'bold',

padding: 0,

color: Colors.teritiary,

fontFamily: 'Roboto',

},

secondHeader: {

fontSize: 18,

marginTop: 0,

textAlign: 'center',

fontWeight: 'bold',

padding: 0,

margin: 20,

color: Colors.teritiary,

fontFamily: 'Roboto',

},

formArea: {

width: '85%',

},

styledInput: {

padding: 18,

paddingLeft: 50,

fontSize: 16,

backgroundColor: Colors.gray,

borderWidth: 3,

borderColor: Colors.secondary,

borderRadius: 12,

color: Colors.teritiary,

marginVertical: 4,

marginBottom: 10,

},

styledInputLabel: {

color: Colors.teritiary,

fontSize: 12,

textAlign: 'left',

padding: 5,

},

leftIcon: {

position: 'absolute',

top: 47,

left: 10,

zIndex: 1,

},

rightIcon: {

position: 'absolute',

top: 47,

right: 10,

zIndex: 1,

},

styledButton: {

backgroundColor: Colors.blue,

color: Colors.teritiary,

padding: 15,

alignContent: 'center',

borderRadius: 12,

marginVertical: 5,

justifyContent: 'center',

height: 60,

width: '60%',

alignSelf: 'center',

},

buttonText: {

color: Colors.primary,

textAlign: 'center',

fontSize: 16,

padding: 5,

},

msgBox: {

textAlign: 'center',

fontSize: 12,

},

line: {

height: 1,

width: '100%',

backgroundColor: '#44361b',

marginVertical: 5,

},

extraView: {

justifyContent: 'center',

flexDirection: 'row',

alignItems: 'center',

padding: 10,

},

extraText: {

justifyContent: 'center',

alignContent: 'center',

alignItems: 'center',

color: Colors.teritiary,

fontSize: 14,

},

textLink: {

justifyContent: 'center',

alignItems: 'center',

},

textLinkContent: {

backgroundColor: Colors.primary,

color: Colors.secondary,

fontSize: 14,

},

});

export default Login;

**9.2.5 Home.js**

import React, { useState, useEffect } from 'react';

import { View, StyleSheet, Text, ScrollView, Image, TouchableOpacity, ActivityIndicator } from 'react-native';

import { Ionicons } from '@expo/vector-icons';

import ItemCard from '../components/ItemCard';

import { firebase } from '../firebaseConfig';

import Slider from '@react-native-community/slider';

import \* as Location from "expo-location";

const getDistance = (latitude1, longitude1, latitude2, longitude2) => {

const toRadian = n => (n \* Math.PI) / 180;

const R = 6371; // km

const dLat = toRadian(latitude2 - latitude1);

const dLon = toRadian(longitude2 - longitude1);

const lat1 = toRadian(latitude1);

const lat2 = toRadian(latitude2);

const a = Math.sin(dLat / 2) \* Math.sin(dLat / 2) +

Math.cos(lat1) \* Math.cos(lat2) \* Math.sin(dLon / 2) \* Math.sin(dLon / 2);

const c = 2 \* Math.atan2(Math.sqrt(a), Math.sqrt(1 - a));

return R \* c;

};

const Colors = {

primary: "#ffffff",

secondary: "#ED5B2D",

tertiary: "#291B25",

brand: "#ff8316",

blue: "#569fa0",

gray: "#f9e8d1"

};

const Home = ({ navigation }) => {

const [name, setFullName] = useState('');

const [selectedSection, setSelectedSection] = useState('availableItems');

const [selectedSubsection, setSelectedSubsection] = useState('byOthers');

const [availableItems, setAvailableItems] = useState([]);

const [rentRequests, setRentRequests] = useState([]);

const [loading, setLoading] = useState(true);

const [isEmailVerified, setIsEmailVerified] = useState(false);

const [locationPermissionGranted, setLocationPermissionGranted] = useState(false);

const [checkingEmailVerification, setCheckingEmailVerification] = useState(true);

const [currentLocation, setCurrentLocation] = useState(null);

const [distanceFilter, setDistanceFilter] = useState(20);

const [maxDistance, setMaxDistance] = useState(20);

const fetchUser = async () => {

try {

const user = firebase.auth().currentUser;

const userSnapshot = await firebase.firestore().collection('users').doc(user.uid).get();

if (userSnapshot.exists) {

const userData = userSnapshot.data();

setFullName(userData.fullName);

setIsEmailVerified(user.emailVerified);

} else {

alert("User does not exist");

}

} catch (error) {

console.error("Error fetching user data: ", error);

}

};

const getLocation = async () => {

let { status } = await Location.requestForegroundPermissionsAsync();

if (status !== "granted") {

console.log("Permission to access location was denied");

setLocationPermissionGranted(false);

setLoading(false); // Stop loading if permission is denied

return;

}

setLocationPermissionGranted(true);

let location = await Location.getCurrentPositionAsync({});

setCurrentLocation(location.coords);

setLoading(false);

};

const fetchItems = async () => {

try {

const itemsSnapshot = await firebase.firestore().collection('items').get();

const items = itemsSnapshot.docs.map(doc => {

const item = doc.data();

const distance = currentLocation ? getDistance(

currentLocation.latitude,

currentLocation.longitude,

item.location.latitude,

item.location.longitude

) : null;

return { id: doc.id, ...item, distance };

});

setAvailableItems(items);

} catch (error) {

console.error("Error fetching items: ", error);

}

};

const fetchRentRequests = async () => {

try {

const requestsSnapshot = await firebase.firestore().collection('rentRequests').get();

const requests = requestsSnapshot.docs.map(doc => {

const request = doc.data();

const distance = currentLocation ? getDistance(

currentLocation.latitude,

currentLocation.longitude,

request.location.latitude,

request.location.longitude

) : null;

return { id: doc.id, ...request, distance };

});

setRentRequests(requests);

} catch (error) {

console.error("Error fetching rent requests: ", error);

}

};

const fetchData = async () => {

setLoading(true);

await fetchUser();

if (firebase.auth().currentUser.emailVerified) {

setIsEmailVerified(true);

await getLocation();

} else {

setIsEmailVerified(false);

setCheckingEmailVerification(false);

}

};

useEffect(() => {

fetchData();

}, []);

useEffect(() => {

if (currentLocation) {

fetchItems();

fetchRentRequests();

setLoading(false);

setCheckingEmailVerification(false);

}

}, [currentLocation]);

const handleRefresh = async () => {

setLoading(true);

await fetchItems();

await fetchRentRequests();

setLoading(false);

};

const renderItems = () => {

const filteredItems = availableItems.filter(item => {

if (selectedSubsection === 'byYou') {

return item.ownerId === firebase.auth().currentUser.uid;

} else {

return item.ownerId !== firebase.auth().currentUser.uid && (distanceFilter === maxDistance || item.distance <= distanceFilter);

}

});

return (

<View style={styles.itemContainer}>

{filteredItems.length === 0 ? (

<Text style={styles.noItemsText}>No items available</Text>

) : (

filteredItems.map(item => (

<ItemCard

key={item.id}

image={item.image}

itemId={item.id}

itemName={item.itemName}

rentPerHour={item.rentPerHour}

description={item.description}

mode={selectedSection}

subMode={selectedSubsection}

phoneNumber={item.mobile}

location={item.location}

distance={item.distance}

currentLocation={currentLocation}

handleRefresh={handleRefresh}

ownerId={item.ownerId}

/>

))

)}

</View>

);

};

const renderRequests = () => {

const filteredRequests = rentRequests.filter(request => {

if (selectedSubsection === 'byYou') {

return request.ownerId === firebase.auth().currentUser.uid;

} else {

return request.ownerId !== firebase.auth().currentUser.uid && (distanceFilter === maxDistance || request.distance <= distanceFilter);

}

});

return (

<View style={styles.itemContainer}>

{filteredRequests.length === 0 ? (

<Text style={styles.noItemsText}>No rent requests available</Text>

) : (

filteredRequests.map(request => (

<ItemCard

key={request.id}

image={request.image}

itemId={request.id}

itemName={request.itemName}

rentPerHour={request.rentPerHour}

mode={selectedSection}

ownerId={request.ownerId}

handleRefresh={handleRefresh}

subMode={selectedSubsection}

description={request.description}

phoneNumber={request.mobile}

distance={request.distance}

currentLocation={currentLocation}

location={request.location}

/>

))

)}

</View>

);

};

return (

<View style={{ flex: 1 }}>

<ScrollView contentContainerStyle={{ flexGrow: 1 }}>

<View style={styles.container}>

<Image source={require("../assets/LogoLeftRight.png")} style={styles.logo} resizeMode="contain" />

{isEmailVerified ? (

locationPermissionGranted ? (

loading ? (

<View style={styles.loadingContainer}>

<ActivityIndicator size="large" color={Colors.blue} />

</View>

) : (

<>

<View style={styles.welcomeContainer}>

<Text style={styles.text}>Welcome {name}</Text>

<View style={styles.iconContainer}>

<TouchableOpacity style={styles.iconButton} onPress={handleRefresh}>

<Ionicons name="refresh" size={26} color={Colors.blue} />

</TouchableOpacity>

<TouchableOpacity style={styles.iconButton} onPress={() => { firebase.auth().signOut().then(() => navigation.navigate("Login")) }}>

<Ionicons name="log-out-outline" size={26} color={Colors.blue} />

</TouchableOpacity>

</View>

</View>

<View style={styles.buttonGroupPrimary}>

<TouchableOpacity

style={[styles.sectionPrimaryButton, selectedSection === 'availableItems' && styles.activePrimaryButton]}

onPress={() => setSelectedSection('availableItems')}

>

<Text style={[styles.primaryButtonText, selectedSection === 'availableItems' && styles.activePrimaryButtonText]}>Available Items</Text>

</TouchableOpacity>

<TouchableOpacity

style={[styles.sectionPrimaryButton, selectedSection === 'rentRequests' && styles.activePrimaryButton]}

onPress={() => setSelectedSection('rentRequests')}

>

<Text style={[styles.primaryButtonText, selectedSection === 'rentRequests' && styles.activePrimaryButtonText]}>Rent Requests</Text>

</TouchableOpacity>

</View>

<View style={styles.buttonGroup}>

<TouchableOpacity

style={[styles.sectionButton, selectedSubsection === 'byOthers' && styles.activeButton]}

onPress={() => setSelectedSubsection('byOthers')}

>

<Text style={styles.buttonText}>By Others</Text>

</TouchableOpacity>

<TouchableOpacity

style={[styles.sectionButton, selectedSubsection === 'byYou' && styles.activeButton]}

onPress={() => setSelectedSubsection('byYou')}

>

<Text style={styles.buttonText}>By You</Text>

</TouchableOpacity>

</View>

{selectedSubsection === 'byOthers' && (

<View style={styles.sliderContainer}>

<Text>Filter by distance: {distanceFilter === maxDistance ? "Unlimited" : `${distanceFilter} km`}</Text>

<Slider

style={{ width: 200, height: 40 }}

minimumValue={1}

maximumValue={maxDistance + 1}

step={1}

value={distanceFilter}

onValueChange={(value) => setDistanceFilter(value === maxDistance + 1 ? maxDistance : value)}

minimumTrackTintColor={Colors.blue}

maximumTrackTintColor="#000000"

/>

</View>

)}

{selectedSection === 'availableItems' ? renderItems() : renderRequests()}

</>

)

) : (

<View style={styles.verificationContainer}>

<Text style={styles.text}>Please grant location permission to continue.</Text>

<TouchableOpacity style={styles.loginButton} onPress={getLocation}>

<Text style={styles.loginButtonText}>Grant Location Permission</Text>

</TouchableOpacity>

</View>

)

) : (

<View style={styles.verificationContainer}>

<Text style={styles.text}>Please verify your email to continue.</Text>

<Text style={styles.text}>Check your inbox and try logging in again.</Text>

<TouchableOpacity style={styles.loginButton} onPress={() => { firebase.auth().signOut().then(() => navigation.navigate("Login")) }}>

<Text style={styles.loginButtonText}>Go to Login</Text>

</TouchableOpacity>

</View>

)}

</View>

</ScrollView>

{isEmailVerified && locationPermissionGranted && (

<TouchableOpacity style={styles.addButton} onPress={() => navigation.navigate("AddItem", { currentLocation })}>

<Ionicons name="add" size={36} color={Colors.primary} />

</TouchableOpacity>

)}

</View>

);

}

const styles = StyleSheet.create({

container: {

flex: 1,

alignItems: 'center',

paddingVertical: 20,

},

logo: {

width: 150,

height: 70,

marginVertical: 5,

},

text: {

fontSize: 18,

marginVertical: 10,

textAlign: 'center',

color: Colors.tertiary,

},

buttonGroupPrimary: {

flexDirection: 'row',

justifyContent: 'space-evenly',

margin: 5,

alignItems: 'center'

},

sectionPrimaryButton: {

padding: 15,

marginHorizontal: 5,

},

primaryButtonText: {

color: "#ccc",

fontSize: 22,

fontWeight: 'bold'

},

activePrimaryButtonText: {

color: Colors.brand,

},

activePrimaryButton: {

borderBottomWidth: 3,

borderBottomColor: Colors.brand,

},

sliderContainer: {

flexDirection: 'column',

alignItems: 'center',

marginVertical: 10,

},

buttonGroup: {

flexDirection: 'row',

marginVertical: 10,

},

sectionButton: {

padding: 10,

backgroundColor: '#ccc',

borderRadius: 5,

marginHorizontal: 5,

},

activeButton: {

backgroundColor: Colors.blue,

},

buttonText: {

color: Colors.primary,

fontWeight: 'bold',

fontSize: 16,

},

buttonTextDark: {

color: Colors.tertiary,

},

welcomeContainer: {

flexDirection: 'row',

width: "100%",

justifyContent: 'space-between',

alignItems: 'center',

paddingHorizontal: 25,

},

iconContainer: {

flexDirection: 'row',

},

iconButton: {

marginLeft: 10,

},

itemContainer: {

flexDirection: 'row',

flexWrap: 'wrap',

margin: 5,

padding: 5,

justifyContent: 'space-between',

},

logoutButton: {

borderWidth: 3,

borderColor: Colors.blue,

padding: 15,

borderRadius: 45,

marginTop: 20,

alignItems: 'center',

marginLeft: 10,

width: '30%',

},

noItemsText: {

fontSize: 18,

color: 'gray',

textAlign: 'center',

marginTop: 20,

},

addButton: {

position: 'absolute',

bottom: 20,

right: 20,

backgroundColor: Colors.blue,

width: 60,

height: 60,

borderRadius: 30,

justifyContent: 'center',

alignItems: 'center',

shadowColor: '#000',

shadowOffset: { width: 0, height: 2 },

shadowOpacity: 0.3,

shadowRadius: 2,

elevation: 5,

},

loadingContainer: {

flex: 1,

justifyContent: 'center',

alignItems: 'center',

},

verificationContainer: {

alignItems: 'center',

justifyContent: 'center',

marginVertical: 20,

},

loginButton: {

marginTop: 20,

backgroundColor: Colors.blue,

paddingVertical: 10,

paddingHorizontal: 20,

borderRadius: 5,

},

loginButtonText: {

color: Colors.primary,

fontSize: 18,

},

});

export default Home;

**9.2.6 ItemCard.js**

import React from 'react';

import { View, Text, Image, TouchableOpacity, StyleSheet, Alert } from 'react-native';

import { useNavigation } from '@react-navigation/native';

import { firebase } from '../firebaseConfig'; // Ensure firebaseConfig is correctly set up

const Colors = {

primary: "#ffffff",

secondary: "#ED5B2D",

teritiary: "#291B25",

brand: "#ff8316",

blue: "#569fa0",

gray: "#f9e8d1"

};

const ItemCard = ({ image, itemName, rentPerHour, description, phoneNumber,currentLocation, location, mode,distance, subMode, ownerId, itemId,handleRefresh }) => {

const navigation = useNavigation();

//console.log(distance + "asdf")

//console.log("Submode " + mode +" " + ownerId)

let buttonTextDisplay;

let displayRed = false;

if (mode === "availableItems") {

if (subMode === "byYou") {

buttonTextDisplay = "Remove Item";

displayRed = true;

} else {

buttonTextDisplay = "Request Item";

}

} else if (mode === "rentRequests") {

if (subMode === "byYou") {

buttonTextDisplay = "Remove Rent Request";

displayRed = true;

} else {

buttonTextDisplay = "Contact Person";

}

}

let costDisplay = mode === "availableItems" ? "Cost per Hour: " : "Expected cost/Hour: ";

const handlePress = async () => {

if (subMode !== 'byYou') {

//console.log(location)

navigation.navigate('DetailedItem', {

item: { image, itemName, rentPerHour, description, phoneNumber, location, mode, ownerId,currentLocation,distance }

});

} else {

// Handle deletion

const collec = mode === "availableItems" ? "items" : "rentRequests";

Alert.alert(

"Confirm Deletion",

"Are you sure you want to remove this item?",

[

{ text: "Cancel", style: "cancel" },

{

text: "Yes",

onPress: async () => {

try {

console.log(`Attempting to delete item with ID: ${itemId}`);

await firebase.firestore().collection(collec).doc(itemId).delete();

console.log("Item successfully deleted.");

Alert.alert("Item removed", "The item has been removed successfully.");

handleRefresh();

} catch (error) {

console.error("Error removing item: ", error);

Alert.alert("Error", "There was an error removing the item. Please try again.");

}

}

}

]

);

}

};

return (

<View style={styles.container}>

<Image source={{ uri: image }} style={styles.image} />

<Text style={styles.itemName}>{itemName}</Text>

<Text style={styles.rentPerHour}>{costDisplay} {rentPerHour}</Text>

<TouchableOpacity onPress={handlePress} style={[styles.rentButton, displayRed && styles.redButton]}>

<Text style={styles.rentButtonText}>{buttonTextDisplay}</Text>

</TouchableOpacity>

</View>

);

};

const styles = StyleSheet.create({

container: {

width: '45%',

marginBottom: 20,

borderWidth: 3,

borderColor: '#ddd',

backgroundColor: Colors.primary,

borderRadius: 10,

padding: 10,

margin: 5,

},

image: {

width: '100%',

height: 150,

resizeMode: 'cover',

borderRadius: 8,

},

itemName: {

fontSize: 18,

fontWeight: 'bold',

marginTop: 10,

},

rentPerHour: {

fontSize: 12,

marginTop: 5,

},

description: {

fontSize: 14,

marginTop: 5,

color: 'gray',

},

redButton: {

backgroundColor: '#f43e11',

},

rentButton: {

backgroundColor: Colors.blue,

borderRadius: 5,

marginTop: 10,

paddingVertical: 8,

alignItems: 'center',

},

rentButtonText: {

color: 'white',

fontWeight: 'bold',

},

});

export default ItemCard;

**9.2.7 DetailedItem.js**

import React, { useState, useEffect } from 'react';

import { View, Text, Image, StyleSheet, ScrollView, ActivityIndicator } from 'react-native';

import { firebase } from '../firebaseConfig';

const Colors = {

primary: "#ffffff",

secondary: "#ED5B2D",

tertiary: "#291B25",

brand: "#ff8316",

blue: "#569fa0",

gray: "#f9e8d1"

};

const DetailedItem = ({ route }) => {

const { item } = route.params;

const currentLocation = item.currentLocation;

const [ownerName, setOwnerName] = useState('');

const [loading, setLoading] = useState(true);

const distance = item.distance;

console.log(distance)

useEffect(() => {

const fetchOwnerName = async () => {

try {

const userDoc = await firebase.firestore().collection('users').doc(item.ownerId).get();

if (userDoc.exists) {

const userData = userDoc.data();

setOwnerName(userData.fullName);

} else {

console.log("No such document!");

}

} catch (error) {

console.error("Error fetching owner details: ", error);

} finally {

setLoading(false);

}

};

fetchOwnerName();

}, [item.ownerId, currentLocation, item.location]);

if (loading) {

return (

<View style={styles.loadingContainer}>

<ActivityIndicator size="large" color={Colors.brand} />

</View>

);

}

return (

<ScrollView contentContainerStyle={styles.container}>

<Text style={styles.header}>Detailed Item Info</Text>

<Image source={{ uri: item.image }} style={styles.image} />

<View style={styles.detailsContainer}>

<Text style={styles.itemName}>{item.itemName}</Text>

<Text style={styles.rentPerHour}>Cost per Hour: {item.rentPerHour}</Text>

<Text style={styles.description}>{item.description}</Text>

<View style={styles.contactContainer}>

<Text style={styles.contactLabel}>Contact:</Text>

<Text style={styles.phoneNumber}>{item.phoneNumber}</Text>

</View>

<View style={styles.distanceContainer}>

<Text style={styles.distanceLabel}>Distance:</Text>

<Text style={styles.distance}>

{distance < 1 ? "Less than 1 km" : `${distance.toFixed(2)} km`}

</Text>

</View>

<View style={styles.ownerContainer}>

<Text style={styles.ownerLabel}>Owner:</Text>

<Text style={styles.ownerName}>{ownerName}</Text>

</View>

</View>

</ScrollView>

);

};

const styles = StyleSheet.create({

header: {

fontSize: 28,

fontWeight: 'bold',

marginBottom: 16,

},

container: {

flexGrow: 1,

padding: 16,

backgroundColor: Colors.primary,

justifyContent: 'center',

alignItems: 'center',

}, distanceContainer: {

marginBottom: 16,

},

distanceLabel: {

fontSize: 16,

fontWeight: 'bold',

color: Colors.tertiary,

marginBottom: 4,

},

distance: {

fontSize: 16,

color: Colors.tertiary,

},

image: {

width: '100%',

height: 250,

borderRadius: 8,

marginBottom: 16,

},

detailsContainer: {

padding: 16,

backgroundColor: Colors.gray,

borderRadius: 8,

width: '100%',

shadowColor: '#000',

shadowOffset: { width: 0, height: 2 },

shadowOpacity: 0.1,

shadowRadius: 8,

elevation: 5,

},

itemName: {

fontSize: 24,

fontWeight: 'bold',

color: Colors.tertiary,

marginBottom: 8,

},

rentPerHour: {

fontSize: 18,

color: Colors.tertiary,

marginBottom: 8,

},

description: {

fontSize: 16,

color: Colors.secondary,

marginBottom: 16,

},

contactContainer: {

marginBottom: 16,

},

contactLabel: {

fontSize: 16,

fontWeight: 'bold',

color: Colors.tertiary,

marginBottom: 4,

},

phoneNumber: {

fontSize: 16,

color: Colors.tertiary,

},

locationContainer: {

marginBottom: 16,

},

locationLabel: {

fontSize: 16,

fontWeight: 'bold',

color: Colors.tertiary,

marginBottom: 4,

},

location: {

fontSize: 16,

color: Colors.tertiary,

},

ownerContainer: {

marginBottom: 16,

},

ownerLabel: {

fontSize: 16,

fontWeight: 'bold',

color: Colors.tertiary,

marginBottom: 4,

},

ownerName: {

fontSize: 16,

color: Colors.tertiary,

},

loadingContainer: {

flex: 1,

justifyContent: 'center',

alignItems: 'center',

backgroundColor: Colors.primary,

},

});

export default DetailedItem;

**9.2.8 AddItem.js**

import React, { useState, useEffect } from 'react';

import { View, StyleSheet, Text, TextInput, TouchableOpacity, ScrollView, Alert } from 'react-native';

import FontAwesome6 from '@expo/vector-icons/FontAwesome6';

import { Octicons } from '@expo/vector-icons';

import { Picker } from '@react-native-picker/picker';

import { firebase } from '../firebaseConfig';

const Colors = {

primary: "#ffffff",

secondary: "#ED5B2D",

teritiary: "#291B25",

brand: "#ff8316",

blue: "#569fa0",

gray: "#f9e8d1"

};

const AddItem = ({ route, navigation }) => {

const { currentLocation } = route.params;

const [selectedType, setSelectedType] = useState('item');

const [itemName, setItemName] = useState('');

const [imageUrl, setImageUrl] = useState('');

const [description, setDescription] = useState('');

const [rentPerHour, setCostPerHour] = useState('');

const [location, setLocation] = useState(currentLocation || null);

const [mobile, setMobile] = useState('');

const handleAddItem = async () => {

if (!itemName || !imageUrl || !description || !rentPerHour || !location || !mobile) {

Alert.alert("Validation Error", "Please fill in all the fields.");

return;

}

try {

const newItem = {

itemName,

image: imageUrl,

description,

rentPerHour,

location,

mobile,

type: selectedType,

ownerId: firebase.auth().currentUser.uid

};

const collectionName = selectedType === 'item' ? 'items' : 'rentRequests';

await firebase.firestore().collection(collectionName).add(newItem);

navigation.navigate('Home', { refresh: true });

} catch (error) {

Alert.alert("Error", error.message);

}

};

useEffect(() => {

if (!currentLocation) {

const requestLocationPermission = async () => {

let { status } = await Location.requestForegroundPermissionsAsync();

if (status !== 'granted') {

console.log('Permission to access location was denied');

return;

}

let location = await Location.getCurrentPositionAsync({});

setLocation(location.coords);

};

requestLocationPermission();

}

}, [currentLocation]);

return (

<ScrollView style={styles.scrollContainer} contentContainerStyle={styles.scrollContent}>

<View style={styles.innerContainer}>

<Text style={styles.header}>Add New {selectedType === 'item' ? 'Item' : 'Rent Request'}</Text>

<View style={styles.formArea}>

<View style={styles.pickerWrapper}>

<Picker

selectedValue={selectedType}

onValueChange={(itemValue) => setSelectedType(itemValue)}

style={styles.picker}

>

<Picker.Item label="Add Item" value="item" />

<Picker.Item label="Add Rent Request" value="rentRequest" />

</Picker>

</View>

<MyTextInput

label="Item Name:"

icon="tag"

onChangeText={setItemName}

placeholder="Enter Item Name"

value={itemName}

/>

<MyTextInput

label="Image URL:"

icon="link"

onChangeText={setImageUrl}

placeholder="Enter Image URL"

value={imageUrl}

/>

<MyTextInput

label="Description:"

icon="info"

onChangeText={setDescription}

placeholder="Enter Description"

value={description}

/>

<MyTextInput

label="Cost per Hour:"

icon="money"

onChangeText={setCostPerHour}

placeholder="Enter Cost per Hour"

value={rentPerHour}

/>

<MyTextInput

label="Mobile:"

icon="device-mobile"

onChangeText={setMobile}

placeholder="Enter Mobile Number"

value={mobile}

keyboardType="phone-pad"

/>

<TouchableOpacity style={styles.styledButton} onPress={handleAddItem}>

<Text style={styles.buttonText}>Add {selectedType === 'item' ? 'Item' : 'Rent Request'}</Text>

</TouchableOpacity>

{location && (

<Text style={styles.locationText}>

Location: {location.latitude}, {location.longitude}

</Text>

)}

</View>

</View>

</ScrollView>

);

};

const MyTextInput = ({ label, icon, ...props }) => {

return (

<View>

<View style={styles.leftIcon}>

{icon === "money" ? (

<FontAwesome6 name="money-bill" size={30} color={Colors.brand} />

) : (

<Octicons name={icon} size={30} color={Colors.brand} />

)}

</View>

<Text style={styles.styledInputLabel}>{label}</Text>

<TextInput style={styles.styledInput} {...props} />

</View>

);

};

const styles = StyleSheet.create({

scrollContainer: {

flex: 1,

backgroundColor: Colors.primary,

},

scrollContent: {

flexGrow: 1,

alignItems: 'center',

justifyContent: 'center',

paddingTop: 30,

},

header: {

fontSize: 24,

fontWeight: 'bold',

marginBottom: 16,

color: Colors.teritiary,

},

formArea: {

width: '85%',

},

styledInput: {

padding: 18,

paddingLeft: 50,

fontSize: 16,

backgroundColor: Colors.gray,

borderWidth: 3,

borderColor: Colors.secondary,

borderRadius: 12,

color: Colors.teritiary,

marginVertical: 4,

marginBottom: 10,

},

styledInputLabel: {

color: Colors.teritiary,

fontSize: 12,

textAlign: 'left',

padding: 5,

},

leftIcon: {

position: 'absolute',

top: 47,

left: 10,

zIndex: 1,

},

styledButton: {

backgroundColor: Colors.blue,

color: Colors.teritiary,

padding: 15,

alignContent: 'center',

borderRadius: 12,

marginVertical: 5,

justifyContent: 'center',

height: 60,

width: '60%',

alignSelf: 'center',

},

buttonText: {

color: Colors.primary,

textAlign: 'center',

fontSize: 16,

padding: 5,

},

pickerWrapper: {

borderWidth: 2,

borderColor: Colors.blue,

borderRadius: 12,

marginBottom: 20,

overflow: 'hidden',

},

picker: {

height: 50,

width: '100%',

},

});

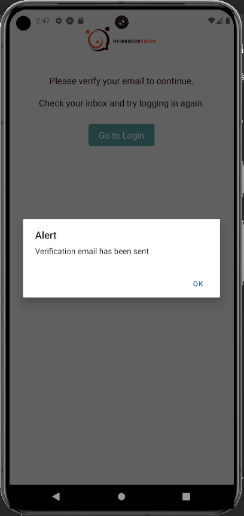
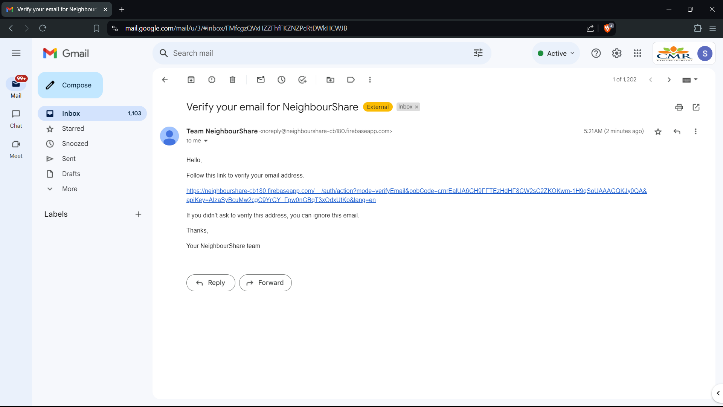
export default AddItem;

**10. SCREENSHOTS**

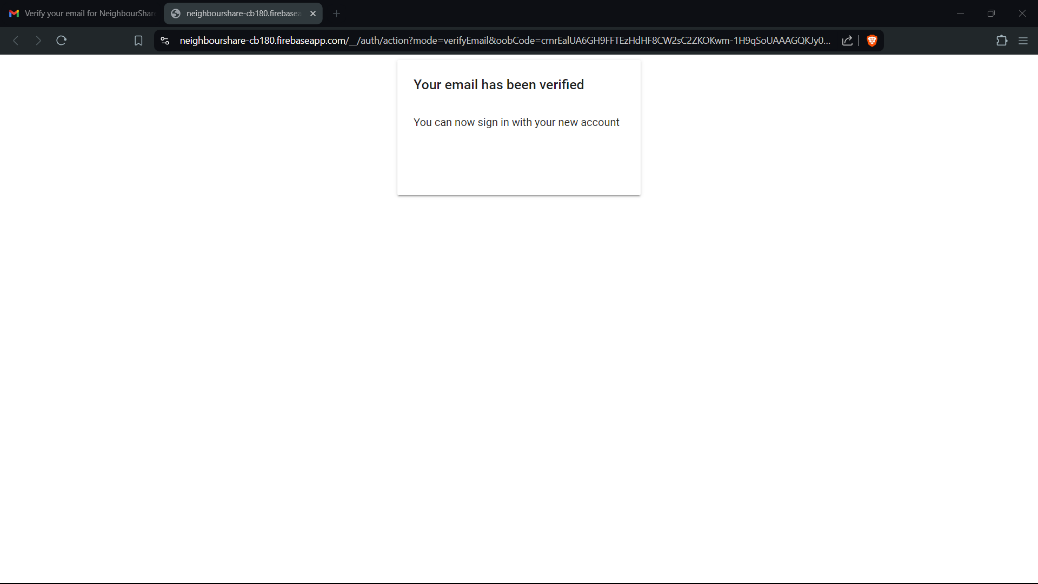
**10.1 Register**



**Fig 10.1.1 Register Screen with hidden pass Fig 10.1.2 Register Screen shown Pass**

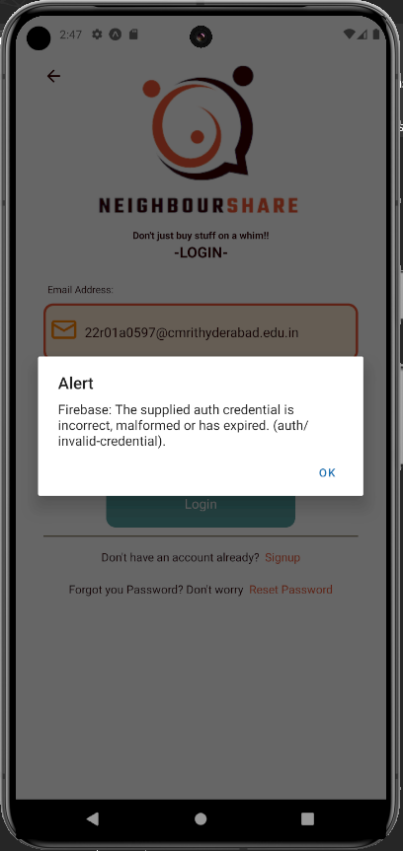
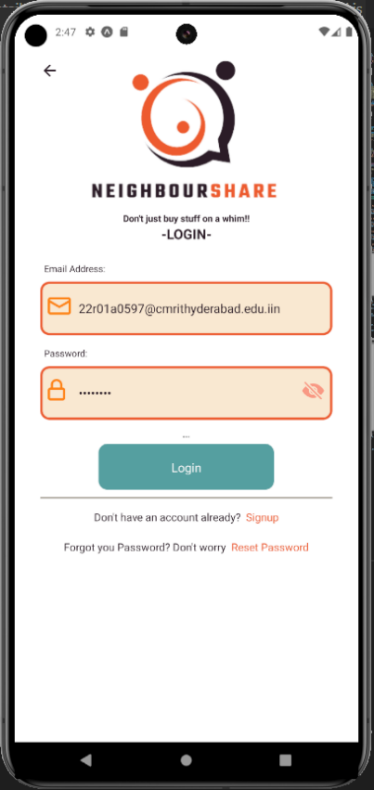
**10.2 Verification**

**Fig 10.2.1 App sent verification mail Fig 10.2.2 Verification mail**

****

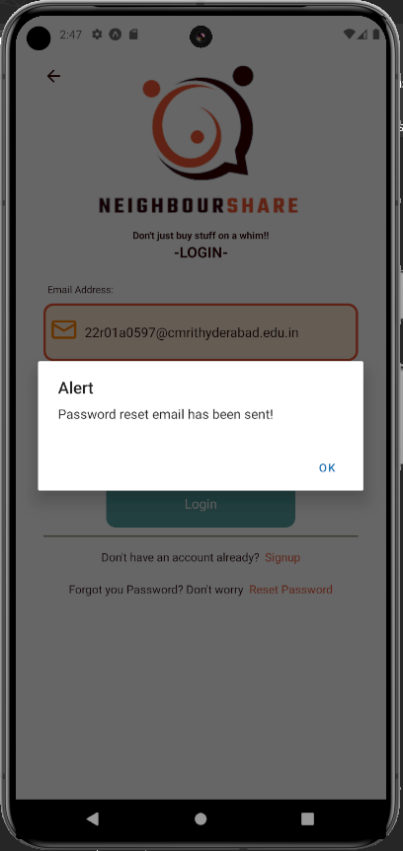
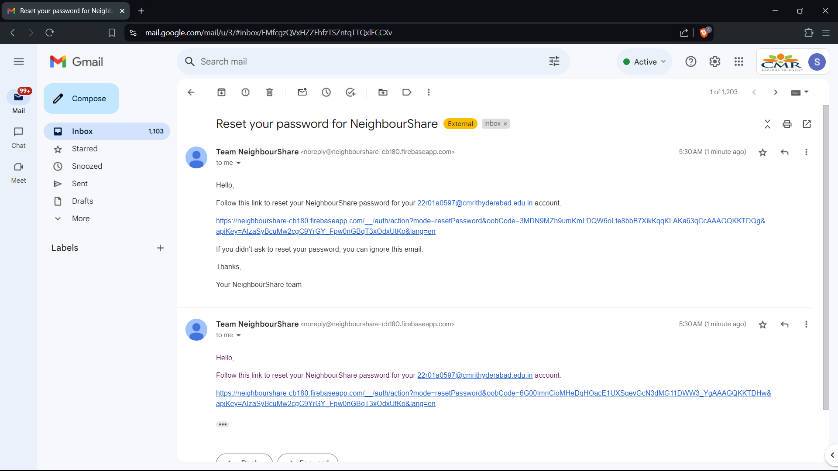
**Fig 10.2.3 Account Verified**

**10.3 Login**

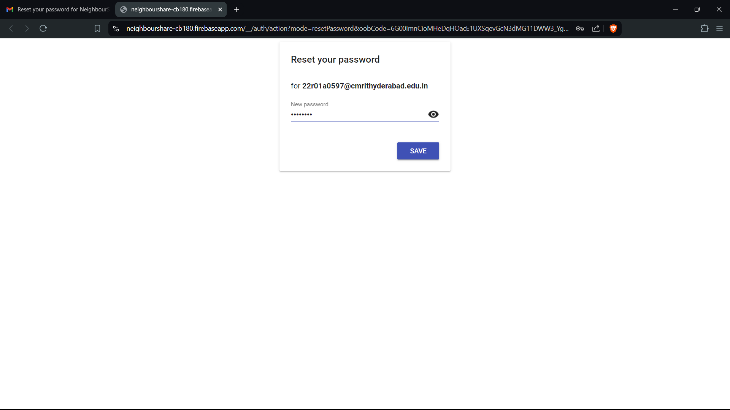
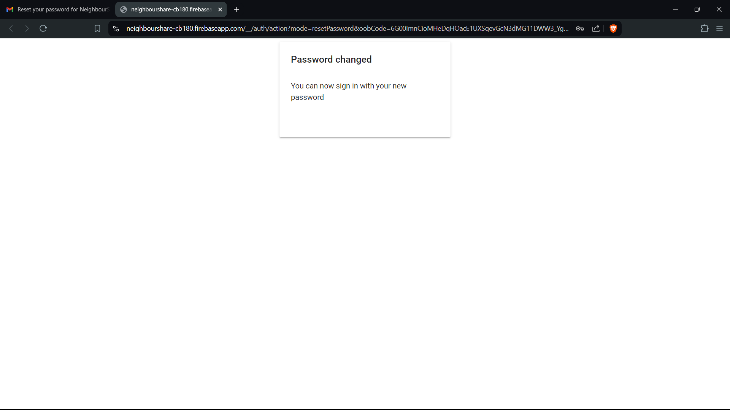
****

**Fig 10.3.1 Login Screen Fig 10.3.2 Invalid Password**

**10.4 Password Reset**

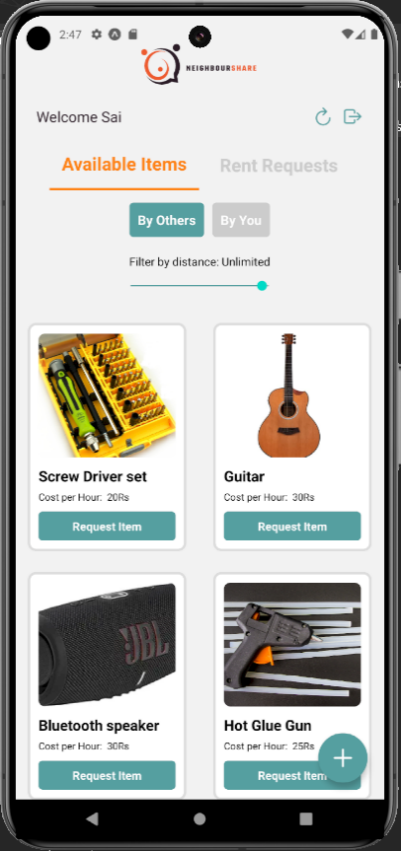
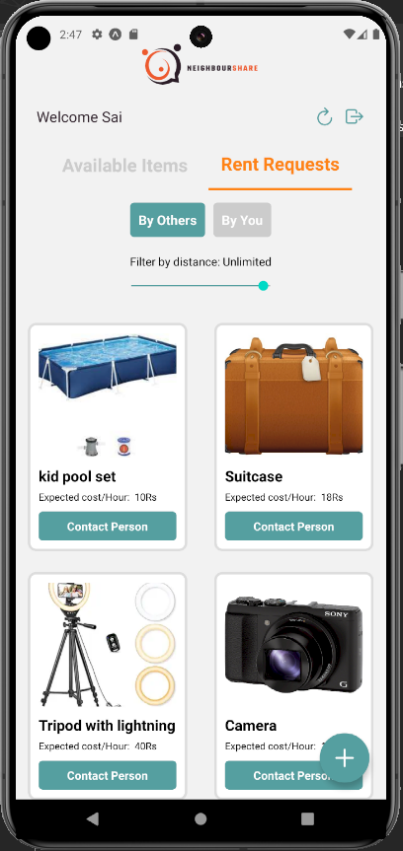
****

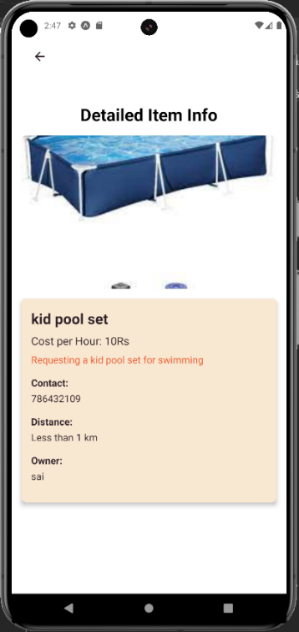
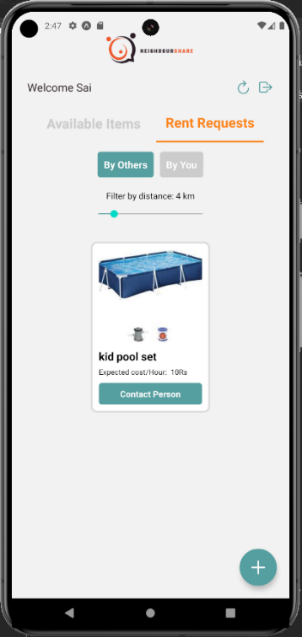
**Fig 10.4.1 Reset mail sent Fig 10.4.2 Reset Mail**

****

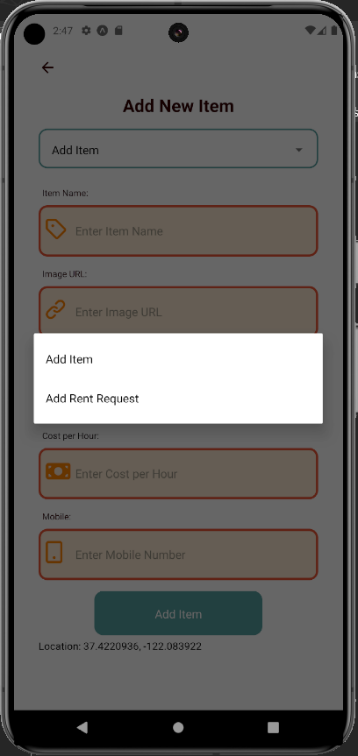
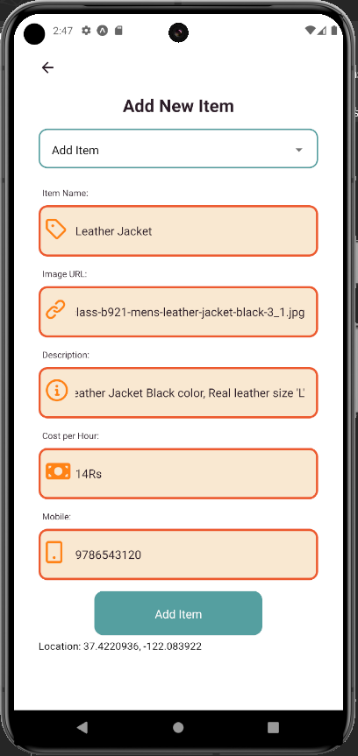
**Fig 10.4.3 Password reset screen Fig 10.4.4 Password reset successful**

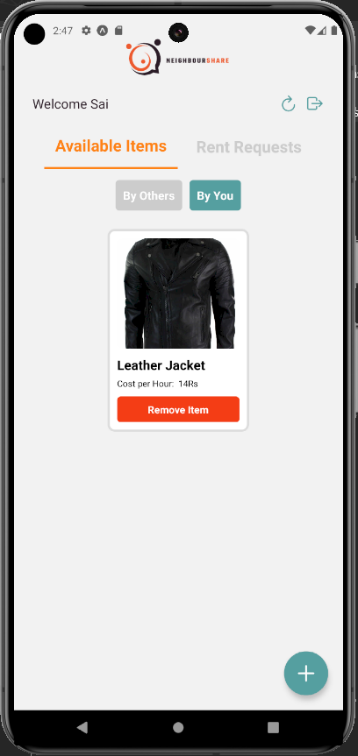
1. **5 Home Screen**

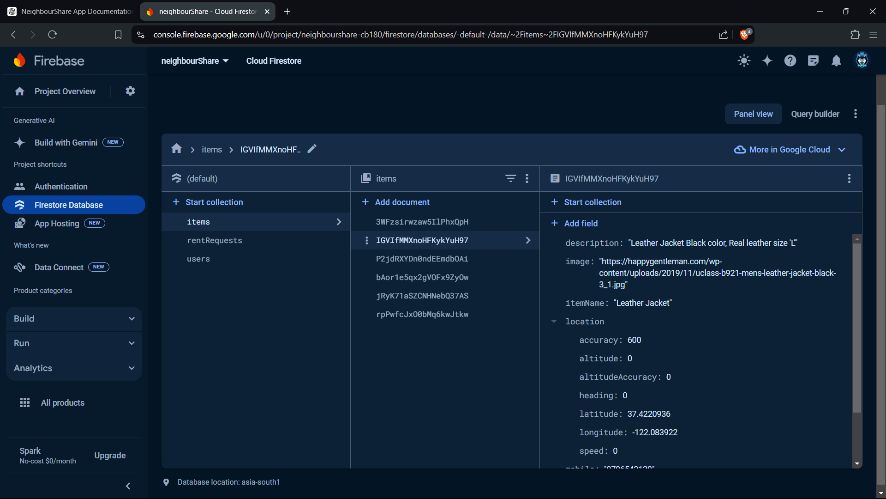
****

** Fig 10.5.1 Available Items Fig 10.5.2 Rent Requests**

**Fig 10.5.3 Filtered Requests Fig 10.5.4 Item/Contact Info**

**10.6 Adding Items**

** Fig 10.6.1 Add Item/Request Fig 10.6.2 Adding Item**



**Fig 10.6.3 Your Item Displayed Fig 10.6.4 Item added in Firestore DB**

**11. TESTING**

The purpose of testing is to discover errors. Testing is the process of trying to discover every conceivable fault or weakness in a work product. It provides a way to check the functionality of components, sub-assemblies, assemblies, and/or a finished product. It ensures that the software system meets its requirements and user expectations and does not fail in an unacceptable manner. Various types of tests address specific testing requirements.

**11.1 TYPES OF TESTS**

**11.1.1 UNIT TESTING**

Unit testing involves designing test cases that validate that the internal program logic functions properly and that program inputs produce valid outputs. All decision branches and internal code flow should be validated. It is the testing of individual software units of the application. It is done after the completion of an individual unit before integration. This structural testing relies on knowledge of its construction and is invasive. Unit tests perform basic tests at the component level and test a specific business process, application, and/or system configuration. Unit tests ensure that each unique path of a business process performs accurately to the documented specifications and contains clearly defined inputs and expected results. Unit testing is usually conducted as part of a combined code and unit test phase of the software lifecycle, although it is not uncommon for coding and unit testing to be conducted as two distinct phases.

**Test Strategy and Approach:** Field testing will be performed manually, and functional tests will be written in detail.

**Test Objectives:**

* All field entries must work properly.
* Pages must be activated from the identified link.
* The entry screen, messages, and responses must not be delayed.
* Features to be Tested:
* Verify that the entries are of the correct format.
* No duplicate entries should be allowed.
* All links should take the user to the correct page.

**11.1.2 INTEGRATION TESTING**

Integration tests are designed to test integrated software components to determine if they run as one program. Testing is event-driven and is more concerned with the basic outcome of screens or fields. Integration tests demonstrate that although the components were individually satisfactory, as shown by successful unit testing, the combination of components is correct and consistent. Integration testing is specifically aimed at exposing problems that arise from the combination of components. Software integration testing is the incremental integration testing of two or more integrated software components on a single platform to produce failures caused by interface defects. The task of the integration test is to check that components or software applications, e.g., components in a software system or software applications at the company level, interact without error.

**Test Results:** All the test cases mentioned above passed successfully. No defects encountered.

**11.1.3 FUNCTIONAL TESTING**

Functional tests provide systematic demonstrations that functions tested are available as specified by the business and technical requirements, system documentation, and user manuals.

**Functional Testing is Centered on the Following Items:**

1. **Valid Input:** Identified classes of valid input must be accepted. Invalid Input: Identified classes of invalid input must be rejected.
2. **Functions:** Identified functions must be exercised.
3. **Output:** Identified classes of application outputs must be exercised.
4. **Systems/Procedures:** Interfacing systems or procedures must be invoked.

**Organization and Preparation:**

Preparation for functional tests is focused on requirements, key functions, or special test cases. In addition, systematic coverage pertaining to identifying business process flows, data fields, predefined processes, and successive processes must be considered for testing. Before functional testing is complete, additional tests are identified, and the effective value of current tests is determined.

**11.1.4 SYSTEM TESTING**

System testing ensures that the entire integrated software system meets requirements. It tests a configuration to ensure known and predictable results. An example of system testing is the configuration-oriented system integration test. System testing is based on process descriptions and flows, emphasizing pre-driven process links and integration points. Software, once validated, must be combined with other system elements (e.g., hardware, people, database). System testing verifies that all the elements are proper and that overall system function performance is achieved. It also tests to find discrepancies between the system and its original objective, current specifications, and system documentation.

**11.2 USER TRAINING**

Whenever a new system is developed, user training is required to educate users about the system's working so that it can be used efficiently. The normal working of the project was demonstrated to prospective users. Its working is easily understandable, and since the expected users are people with good knowledge of computers, using this system is very easy.

**11.3 MAINTENANCE**

Maintenance covers a wide range of activities, including correcting code and design errors. To reduce the need for maintenance in the long run, user requirements were more accurately defined during system development. Depending on the requirements, this system was developed to satisfy needs to the largest possible extent. With technological advancements, it may be possible to add many more features based on future requirements. The coding and design are simple and easy to understand, making maintenance easier.

**11.4 TESTING STRATEGY**

A strategy for system testing integrates system test cases and design techniques into a well-planned series of steps resulting in the successful construction of software. The testing strategy must cooperate with test planning, test case design, test execution, and resultant data collection and evaluation. A strategy for software testing must accommodate low-level tests necessary to verify that a small source code segment has been correctly implemented, as well as high-level tests that validate major system functions against user requirements. Software testing is a critical element of software quality assurance and represents the ultimate review of specification design and coding. Testing represents an interesting anomaly for the software. Thus, a series of tests are performed for the proposed system before the system is ready for user acceptance testing.

**12. CONCLUSION**

NeighbourShare has been successfully developed to foster community sharing by allowing users to request and lend items efficiently. The application was built using modern technologies such as React Native for cross-platform mobile development and Expo for a streamlined development experience. This ensures that NeighbourShare is accessible and user-friendly on both Android and iOS devices.

Throughout the development process, we prioritized robust functionality and user experience. The implementation of secure authentication, real-time item requests, and geolocation features ensures that users can easily find and share items within their local communities. Our use of Node.js for backend processes and Firebase for data storage and notifications ensures scalability and reliability.

Comprehensive testing was conducted, including unit testing, integration testing, and system testing, to validate that the application meets all specified requirements and performs reliably under various conditions. The testing phase confirmed that NeighbourShare operates as intended, providing a seamless and efficient platform for item sharing.

NeighbourShare demonstrates the potential of leveraging technology to build stronger, more connected communities by promoting resource sharing and reducing waste. This project highlights the importance of intuitive design, robust functionality, and thorough testing in developing applications that meet user needs and expectations.

Future enhancements could include expanding the feature set to support more diverse item categories, improving the user interface based on user feedback, and integrating advanced analytics to provide insights into sharing trends within communities. Additionally, exploring partnerships with local organizations and community groups could further enhance the platform's reach and impact.

NeighbourShare not only meets its initial goals but also sets a foundation for future growth and development, contributing to the larger goal of fostering a more sustainable and cooperative society.

**13. REFERENCES**

**Books and Journal Articles**

[1] J. Smith and A. Brown, "Developing Community-Oriented Mobile Applications," J. Mobile Comput., vol. 14, no. 3, pp. 123-134, June 2020.

[2] S. Patel, "A Comprehensive Guide to Cross-Platform Mobile Development," Int. J. Comput. Sci., vol. 22, no. 2, pp. 112-118, April 2021.

[3] A. Kumar and M. Shukla, "Implementing Authentication Systems in Mobile Apps," Softw. Eng. Pract., vol. 30, no. 5, pp. 210-221, May 2019.

**Websites**

[4] Expo. (n.d.). "Expo Documentation," [Online]. Available: https://docs.expo.dev/

[5] Node.js. (n.d.). "Node.js Documentation," [Online]. Available: https://nodejs.org/en/docs/

[6] Firebase. (n.d.). "Firebase Documentation," [Online]. Available: https://firebase.google.com/docs

[7] React Native. (n.d.). "React Native Documentation," [Online]. Available: https://reactnative.dev/docs/getting-started

[15] Android Developers. (n.d.). "Run Apps on the Android Emulator," [Online]. Available: https://developer.android.com/studio/run/emulator